Designing Emergence: Automatic Extraction of Stigmergic Algorithms From Lattice Structures

James G. Adam

A thesis submitted for the degree of Doctor of Philosophy

Department of Computer Science
University of Essex

2005

Abstract

The complex behaviour of relatively-simple creatures in nature, such as that of social insects, has often inspired research into how such intricate networks of cooperation operate, and how such cooperation can be implemented to solve other problems. Stigmergic construction techniques attempt to demonstrate how complex structures may be built without explicit communication between agents.

An abstract model of sematectonic stigmergic construction was presented in 1995 by Bonabeau & Theraulaz[158, 156], capable of reproducing some of the complex structures observed in social insect research. The investigation into this model was motivated by the potential application of stigmergic techniques, by engineers and designers, as the basis of as system of artificial construction.

In this thesis, their model and assertions are critically reviewed, and the concepts provided by Bonabeau & Theraulaz[158, 156, 22, 19] shown to be insufficient for use in arbitrary construction tasks. A new software implementation and new concepts for considering the behaviour of stigmergic algorithms – *post-rules* – are devised and presented as a more concrete basis for investigation. Computational constraints surrounding Bonabeau & Theraulaz's approach, which prohibit further consideration using their techniques, are then presented.

In an attempt to address the motivation which inspired the original work, a novel process is presented, which allows the automatic generation of stigmergic algorithms from arbitrary existing structures. A new algorithm is presented which reduces the *complexity* of a stigmergic rule set. Limitations of this approach are discussed, and several refinements which improve the quality of the extracted algorithm are then described in detail, along with experimental results.

Finally, our ability to derive stigmergic algorithms of the same *quality* as those noted in nature is investigated, and the limitations of sematectonic stigmergy for arbitrary architecture construction are investigated and shown in detail.

Acknowledgements

I would like to express my greatest thanks to my supervisor Owen Holland for his continual support and guidance throughout this project, without whom I would not have reached this point. I would also like to thank my family and friends for providing an anchor for my sanity whilst walking through the philosophical quicksand surrounding 'emergence'. I would like to express my particular gratitude to Murray Steele and Catriona Macdonald for their kind assistance in proof-reading this work, despite its constant revision from under their feet, and to Siobhán Mitchell, for her constant support despite the many long and frustrating hours required.

My thanks also go to Guy Theraulaz, Kjerstin Easton and Joe Andrieu at CalTech for their support in making the *Nest-2.11.1* software available for this project, and also in kindly answering my queries regarding its implementation.

Dedicated to my grandparents.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "[Engineers], in their attempts to design distributed artificial multi-agent systems (cellular and reactive robots, mobile automata), are facing the problem of finding simple behavioural algorithms at the individual level so as to produce a collective performance. The study of collective building processes in social insects seems to be a path to follow in order to discover new kinds of such distributed behavioural algorithms." [156]

1.1 Dial M For Mars

One of the most significant ecological events in the recent history of Earth is not the advent of global warming, nor the depletion of the ozone layer, but simply the number of people currently living on the planet itself at any given moment:

> "It took from the beginning of time until about 1927 to put the first 2 billion people on the planet; less than 50 years to add the next 2 billion people (by 1974); and just 25 years to add the next 2 billion (by 1999). In the most recent 40 years, the population doubled."[37]

As the increase in the human population of our planet continues to accelerate, it is now virtually certain, without the advent of significant population management, that will be *forced* to make homes on other worlds within the Solar System.

It is perhaps with this inevitability in mind, rather than purely the quest for knowledge, that many scientists and engineers are focused on the exploration of nearby planets which might be capable of supporting life. Over the last 50 years exploration into space has changed from a scientific frontier to a nearly common-place event. However, while probes have been dispatched and reported on most of the significant physical objects in our galactic neighbourhood, the only object which has been visited by humans over all this time is our nearest neighbour, the Moon.

1.1.1 At Home in the Universe

While relatively close to the Earth, our celestial partner is not capable of supporting life without a huge investment of external resources. For instance, to control even the temperature of an installation on the Moon, systems which smooth the fluctuation from $-147°C$ in the shade to $+100°C$ in direct sunlight[1] must be developed and installed in whichever location humans wish to inhabit.

In contrast, Mars maintains a temperature of between $-112°C$ and $-8°C$. While this is certainly an extreme environment, it is not unlike temperatures found in polar regions on Earth, and would certainly be far more manageable than the extremes of hot and cold on the Lunar surface. The presence of water (in the form of ice) confirms Mars as the most likely candidate for a 'home away from home' for any human on the look out for extra-planetary property investment.

Despite the apparent feasibility of establishing an off-world presence on Mars, the distance between Mars and the Earth means a trip to Mars will take between 6 and 8 months. It would be of huge benefit to any such mission if a base of some kind were *already present* on the Mars surface, available to house and support the astronauts while they are present on the planet, and provide supplies for the return journey. It has been suggested[90] that the best way to achieve this would be dispatching some robotic systems ahead of a manned mission, which could then assemble the structures and mechanisms required to support human life, prior to the arrival of any astronauts.

[1] Comparisions of surface temperature on Earth, Mars and the Moon can be found at http://www.asi.org/adb/02/05/01/surface-temperature.html

1.1.2 Remote Robotic Construction

Robotic exploration of space is an endeavour marked as often by its failures as it is by its success. A undisputedly significant characteristic of space exploration which contributes to this chequered track record is the *remoteness* of the locations in which these automated systems must operate. Because these robots are operating at distances which must be measured in terms of the time it takes light to travel them[2], they *cannot* be controlled remotely by engineers on Earth. They must accomplish their task fully autonomously.

Furthermore, since these robots are distant from their maintainers, the possibility of a failure in one or more components must be seriously considered, and measures taken to ensure that any such failure does not compromise the achievement of the system's goal. This fault-tolerance will be of particularly importance if the lives of a number of en-route astronauts are relying on the successful assembly of their planetary life support mechanism to survive on arrival.

Finally, the system must be robust in the face of unexpected situations. It is almost certain that whatever environment the construction system is deployed in, unexpected conditions, environmental configurations or chance events will be encountered, many of which may not have been specifically considered by the system's engineers. It is therefore vital that the robotic agents can adapt their behaviour to accommodate these deviations and continue to fulfil their objective.

1.1.3 Construction on Mars: A Summary

While the construction of a habitat on Mars is a somewhat futuristic example, similar problems exist on this planet which share the constraints described above. Often a task must be accomplished in locations where humans cannot directly intervene. These situations might be *inhospitable*, such as the site of nuclear disaster, or in the high-pressure environment of the deep sea. Other situations may simply be *inaccessible*, such as the assembly required on microscopic scales in 'nanotechnology'[53, 164]. In order to satisfactorily address such a

[2]Depending on the relative distance between Mars and Earth as a result of their orbits, it can take up to 40 minutes for a radio signal to be sent from Earth to Mars and back.

problem, any artificial system must exhibit the following traits:

- The system must be capable of operating autonomously, without the instruction (or supervision, to an extent) of humans.
- It must be robust in the presence of unexpected environmental conditions, or when the outcomes of its actions lead to unexpected results.
- Finally, it must be fault-tolerant; if some component fails, the goal should not be automatically compromised.

In the following section, one particular class of solutions based on the observed behaviour of social insects will be considered as a candidate system which claims to fulfil these important criteria.

1.2 The Rise of the Insects

Many of the most spectacular achievements seen in nature are witnessed at the level of its smallest inhabitants. The feats of habitat engineering and coordination performed by social insects, whose equivalent might require months of planning and consideration by humans, seem to coalesce almost without effort or even communication between those who are involved in the construction.

For instance, several species of ants build cone-shaped nests, each over a metre height and housing hundreds of thousands of workers[87]. The equivalent size of this structure for a human would be a forty-floor skyscraper, relative to body size. Similarly, bees and social wasps build intricate nest structures to house and maintain their swarms[95, 94, 17]. These constructions can house many thousands of swarm members, and supports the gestation and development of new generations.

1.2.1 Emergence and Collective Intelligence

What is perhaps most surprising is that the coordination between the many individuals which participate in these construction processes might seem to be invisible to the naked

eye. There is no 'foreman', whose human counterpart would ensure that each task proceeds in order, referring to a plan of the finished structure when necessary.

Instead, the simple building behaviours of each individual, acting alone, combine together and result in the construction of a coherent, functional structure. There is no single intelligence responsible for the coordination of activity required to arrive at this result, but rather a "collective" or "swarm intelligence", present only when considering the colony as a whole. It is also an example of an "emergent behaviour" - while the individual actions of each insect can be described very simply, when many individuals are working simultaneously in the same environment, the behaviour of the system as a single entity is seen to be the construction of a complex structure. This behaviour is said to "emerge" from the individual actions of each insect. These terms will be discussed further in Chapter 2.

1.2.2 Why Use Swarm Intelligence?

The wealth of examples of emergent solutions in nature, and the success of the 'swarm' societies we find in the domain of social insects, is undeniable. However, if such mechanisms are to be useful in the design of other multiagent systems, this type of system must bring with it significant benefits. A selection of the potential benefits of employing swarm intelligence to arbitrary multiagent problems are outlined below.

Simple entities

Given the homogeneous and simple nature of the actors within a biological swarm, artificial swarm agents should share these characteristics. If each agent is both simple, and identical to every other agent, the agents are ideal candidates for *mass production*. Production in bulk also means that large numbers of agents can be produced relatively *cheaply*.

Robustness

As Wavish[161] states with regards to the generation of 'emergent' or collective behaviour:

> "Stable emergent behaviour is valuable both because it can be produced much

> more economically than explicitly programmed behaviour, and because it is typically very robust."[161]

While expected situations, environmental conditions, or abnormal behaviour from within an internal component itself might hinder the performance of a traditional system, or cause it to fail completely. On the other hand, the emergent behaviours exhibited by swarm intelligence can often capitalise on variation in the environment:

> "Randomness or fluctuations... far from being harmful, may in fact greatly enhance the sytem's ability to explore new behaviors and find new 'solutions"'[17]

Built-in Redundancy

Homogeneity allows agents within the system to be *interchangeable*; no agent is tied by its specific design to any particular task. A further consequence of this is that to some extent each agent is also *disposable*. The lack of specialisation in each agent means that if an agent were to fail while the system is in operation, its role may be easily filled by any other agent with the swarm. In other words, if one or more members of the swarm are destroyed or somehow incapacitated, other members should automatically pick up the slack.

As a contrast, if a component fails in a hierarchical system it must be replaced, and the operation of all components 'under' that which failed make no contribution to the performance of the rest of the system. Any failed components must be repaired or replaced before the system can continue to operate properly.

1.2.3 Swarm Intelligence for Autonomous Construction

Each of these characteristics – cheap component agents, redundancy and fault-tolerance – are features which would be highly valuable in *any* system, natural or artificial, regardless of that systems origin or context. It is clear that there is significant motivation for applying swarm intelligence to engineering problems.

There is now a significant motivation to adapting the swarm behaviours of social insects towards solving the arbitrary problems which human engineers and designers are faced with.

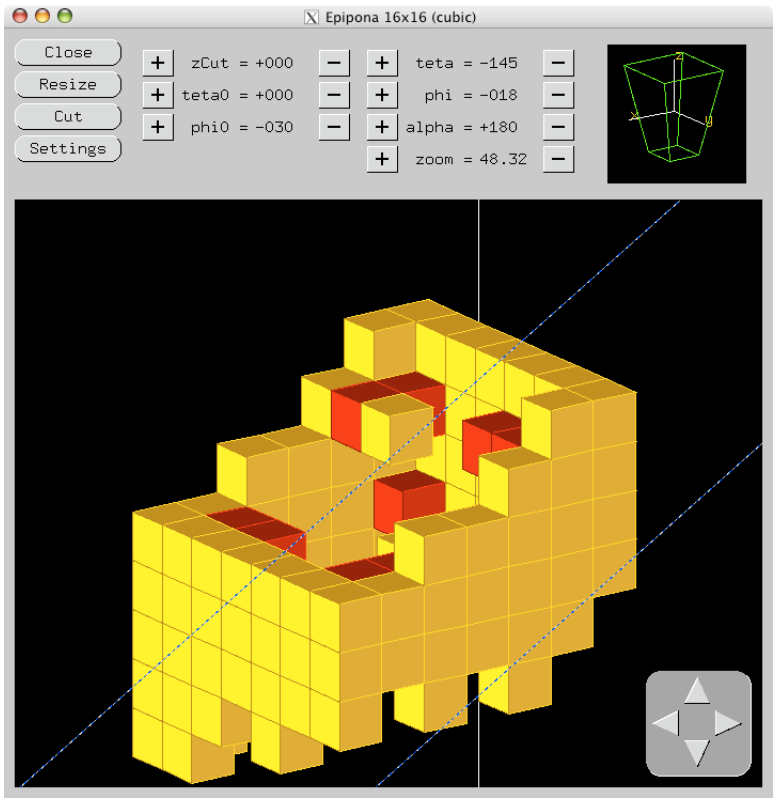


Figure 1.1: An architecture built by the *Nest-2.11.1* software (described fully in Chapter 3).

The parallels between construction in social insects and the remote construction problem described above are self-evident. As Bonabeau and Theraulaz noted in the quotation at the head of this chapter: "the problem of finding simple behavioural algorithms at the individual level so as to produce a collective performance."[156].

1.3 The *Nest* Model of Swarm Construction

The model presented by Bonabeau and Theraulaz[158, 156], and implemented with their *Nest-2.11.1* software, is an example of an extremely simple system which is capable of demonstrating some of the construction behaviour seen in social insects. An example of the type of structure which can be built using this simple system is shown in Figure 1.1.

The *Nest* model employs a form of coordination called 'sematectonic stigmergy' (see Section 2.3.4) to ensure that each agent activities work towards the construction of a single, coherent structure. The mechanisms underlying this system will be subject of further dis-

cussion and investigation in the following sections of this thesis.

1.3.1 *Nest* in Action

During simulation, the *Nest* system operates as follows. Virtual 'bee' agents move within a discrete representation of space, divided into geometrically-shaped (hexagonal or cubic) cells. Cells may be empty, or may contain a piece of matter, otherwise known as a *brick*. A brick may also have a colour, so that different building materials can be distinguished from each other by the agents.

At any point in time an agent may occupy a region of the lattice whose local arrangement of bricks and empty cells corresponds to a pre-determined **stimulating configuration**. The agent has access to a list of such configurations, otherwise known as *rules*. The list of rules is also called a **stigmergic algorithm** or **stigmergic script** (consiting of 'stigmergic rules', or rules which operate using stigmergy; see Section 2.3). If an agent encounters a local configuration which matches one of the rules within the algorithm, a brick (the colour of which is also determined by the rule) is placed in the agent's current location.

1.3.2 *Nest* – A Candidate for Autonomous Construction

It would appear, given the architecture seen in Figure 1.1, that interesting and potentially-desirable structure can be created using this simple model. Furthemore, it was shown in [158, 156] that structures which bear remarkable similarities to biological nests can also be produced, given the correct set of stigmergic rules. The potential of this system as the basis of a solution to the practical problems described above must be investigated in detail; this describes the general aim of the work presented here. The specific questions and goals of this research are discussed in the following section.

1.4 Designing Emergence and Swarm Construction

When Bonabeau and Theraulaz (hereafter B&T) introduced their *Nest* model[158] (described in detail as Chapter 3 of this thesis), the construction complex architectures using only very

simple agents was clearly demonstrated. The obvious hope inspired by this demonstration is eventual use of these techniques by engineers and designers to solve problems just like the Mars construction example described above.

However, despite the demonstrations of complex construction in [158, 156, 22, 19], any individual wishing to use swarm intelligence techniques to solve their personal construction problems is left with little more than the notion that it *might* be possible using the model B&T proposed.

This therefore becomes the aim of the investigations presented in this thesis. While B&T's model is clearly capable of producing interesting structures, the application of their techniques to novel problems has never been demonstrated:

- Is it **possible** to use sematectonic stigmergy to build arbitrary structures?

The possibility of applying stigmergic construction to arbitrary construction tasks is useful only if some methodology can be devised which will guide engineers and designers as they apply the general principles involved in stigmergic construction to their own, unique requirements:

- If it *is* possible to use stigmergy for arbitrary tasks, **how** can it be used? In other words, what steps must be taken in order to generate a stigmergic algorithm which will build a given structure?

Finally, if some methodology can be devised to assist in the creation of targeted stigmergic systems, it would seem natural to attempt to codify those guidelines into and algorithmic process:

- Can the application of stigmergy to arbitrary construction problems be **automated** using some algorithm process, and if so, what are the algorithms required?

These are the questions which have driven the research described in the rest of this work, and their solution represents the ultimate goal of this investigation in emergent construction. In addressing these questions, several other questions will also necessarily be encountered:

- What are the **limitations**, if any, of the *Nest* model (and the underlying sematectonic stigmergy which it is built upon)?

- If any limitations exist, can they be **overcome** and if so, how? In other words, where sematectonic stigmergy is ultimately insufficient to solve a particular problem, what must be added to the model to enable the creation of an acceptable solution?

The answers to these questions will provide a greater understanding of the capabilities and suitability of stigmergy as a practical tool.

1.5 Thesis Outline

This section gives an overview of the rest of this thesis. Following this introduction, Chapter 2 provides a more detailed survey of existing related work in other areas, and in particular the areas of emergent multiagent behaviour and stigmergic systems.

Once the background has been presented, Chapter 3 introduces the *Lattice Swarm* model and the *Nest-2.11.1* system developed originally by Bonabeau, Theraulaz et al. [158, 156, 22, 19]. The assertions and experimental results obtained using this system are also critically presented.

In Chapter 4 a new stigmergic simulation system implementation is presented, which improves upon the original work done by Bonabeau et al. and makes significant strides to overcoming some of the artificial restrictions present in their system. The architecture and implementation of the system is described, along with suggestions for further development and experimentation.

Chapter 5 contains a more detailed critical discussion of the assertions presented in the original work, with particular emphasis on the the effectiveness of these results as tools enabling us to exploit stigmergic systems in a practical manner. A new set of tools is then developed – *post-rules* – to both frame and attempt to understand the behaviour of stigmergic simulations. This framework may be used within mechanisms of automatically exploring the behavioural space of these systems. Finally, the computational limitations of this approach are discussed.

In the light of such limitations, Chapter 6 approaches the goal of harnessing stigmergic behaviour from an alternative, 'top-down' perspective. The new framework presented in

Chapter 5 is exploited to extract stigmergic algorithms from pre-existing structures. A simple algorithmic process is developed to solve this problem, consisting of two distinct stages: *ordering* and *state assignment*. The generality of this approach is also shown.

A new state assignment algorithm is then presented in Chapter 7 which minimises the brick state complexity – the number of distinct brick colours – of the extracted algorithm.

Chapter 8 continues this exploration, considering the optimisation of the ordering process. Similar problems from the field of combinatorial analysis are discussed, and a genetic algorithm approach found useful in that field is then applied to the present ordering problem. The limitations of this process are then presented, and the notions of repetition and modularity in abstract stigmergic systems are then explored in Chapter 9.

Finally, Chapter 10 considers issues relating to his investigation which are not directly focused on the practical design of stigmergic algorithms and outlines interesting avenues for further investigation. Chapter 11 concludes this thesis by reiterating the achievements made within this study, and providing answers where possible the the questions presented above.

Chapter 2

Related Work

The work presented in this thesis takes inspiration from wide and varied fields of research. In this section, these links are highlighted along with significant related research which has contributed to the current understanding of stigmergic and emergent multiagent systems.

2.1 Emergence

Scholarly consideration of **emergence** can be traced as far back as Aristotle's *Metaphysica*, where it was noted that "the whole [may be] more than the sum of its parts". In more recent years, the study of this type of phenomenon, where something apparently-new *emerges* from collection of component parts (a system) has increased dramatically. The fascination with emergence is in part due to the sheer quantity and variety of examples of **emergent systems** identifiable to the casual observer.

These examples range from physical systems[13, 86], to chemical interactions such as the famous Belousov-Zhabotinsky reaction[143], to the production of biological structure and patterning[133], extending up to the behaviour of human social structures[38]; from structure within turbulence[153], to the social coordination within ant-colonies[29, 17, 87], to the appearance of behaviours within the global economy. The demonstrated existence of emergence is pervasive and seemingly inescapable.

Perhaps even more so than the 'intelligence' in *Artificial Intelligence*, 'emergence' is the subject of intense scrutiny and debate, in fields ranging from social science to philosophy.

Only a flavour of the depth of this discourse can be given here, in so much as it is relevant to the proceedings which follow thereafter; whenever possible the reader is directed to sources of further information of far greater depth.

2.1.1 Defining Emergence

Many attempts have been made to define 'emergence'. While attempts to *precisely* define the term remain the subject of some debate (see Section 2.1.3 below), many more general formulations have been posited. Baranger suggests that

> "Emergence happens when you switch the focus of attention from one scale to the coarser scale above it."[10]

In his book entitled 'Emergence', Holland's definition is presented in the form of a constraint:

> "The [emergent] behavior of the overall system *cannot* be obtained by *summing* the behaviors of its constituent parts."[85]

Emergence has also been characterised by the requirement for new descriptive categories[149, 150] or vocabularies[161] to describe the behaviour of the system, which were not necessary to describe the behaviour of the individual components:

> "...a designer has a vocabulary of behaviour descriptions $V1$ with which the behaviour of a system can be described, and a much smaller subset $V2$ which is actually used in the written texts describing and ultimately generating the behaviour of the system. *Emergent* behaviour is that behaviour which is produced by behaviour describable by $V2$, but which is itself not describable by $V2$, although it is describable by $V1$."[161]

Crutchfield[41] asserts that

> "A feature emerges when the system puts some effort into creating it."

All these encapsulate the notion of some interesting novel feature which was either not previously present, or is unexpected within the given context. However, like the concept of 'intelligence', there is as yet no universally acceptable definition of emergence, nor a test to indicate the presence or lack of any emergent phenomena within a system.

Classification of Definitions

In the absence of such a general definition, a number of classifications and approaches currently used to understand emergence and emergent phenomena have been outlined. Cariani[30] outlines three distinct approaches to the generation and study of emergence:

Computational Emergence views global emergent properties as arising solely from the underlying deterministic computational interactions. Perhaps the most popular demonstration of computational emergence appears within the cellular automata investigated within the field of artificial life.

Thermodynamic Emergence explains the global behaviour of a system in terms of concepts from dynamical systems, such as attractors, generators and autocatalytic cycles.

Emergence-relative-to-a-model recognises the role of the observer within the emergence-exhibiting system as a whole. In this approach, emergent behaviour is seen as a deviation in behaviour from that predicted by the observer's internal model of the system, thus forcing the observer to modify their model in order to accommodate the new behaviour(s). The relationships between the system, the observer and the measuring mechanisms used by the observer determine what (if any) emergent behaviour is produced.

Types of Emergence

In contrast to these approaches to studying and understanding emergence, Castelfranchi[32] outlines a number of different types of emergence. The first, *diachronic* emergence, is claimed to occur when new behaviours or properties appear over time or through some evolutionary mechanism. *Descriptive* emergence, on the other hand, occurs when a complex system of many interacting components can be described concisely at the level of the system as a whole. This description may involve new concepts and ideas which were not represented at the level of the individual components of the system, and because of this features of the system described by these new terms can be labelled as emergent.

Castelfranchi then outlines a number of other forms of emergence, including *cognitive* emergence, where some implicit knowledge, rule or fact becomes explicit within a mind, and *gestalt* emergence where the interaction between individual elements occurs only within the mind of the observer (such as the emergent 'structure' of a constellation of stars).

Subjectivity

Castelfranchi[32] also highlights one of the major problems encountered when attempting to classify emergent phenomena – the subjectivity of emergence when defined relative to the observer. This problem exists because such a definition of emergence is dependent on the observer to detect and label candidate properties as emergent. Different observers may have different internal models[8, 85], and thus while one observer may declare a given phenomenon as emergent, a second observer may not.

This discrepancy may occur because of a relative deficiency in the cognitive or modelling abilities of the first observer, whereas to the more-enlightened second observer, the emergent behaviour is 'obvious' and was already apparent within their internal model of the system. The very act of modelling may induce such a difference, as it forces the modeller to choose which aspects of the system are relevant and which can be regarded as 'noise'[42]. Different modellers will make different choices, and may miss the crucial measurement which indicates emergence[124].

Cariani[30] acknowledges this by suggesting that the observer/measurer must be considered a part of the system itself. The relationship between an observer's ability to understand and explain (i.e. model) is also a common theme[8, 44, 42, 118].

2.1.2 Second-Order Emergence

While most study of emergence takes place using systems of extremely simple components, when more complex (and even cognitive) components (agents) are present, a second type of emergence becomes increasingly important – so-called "second-order emergence"[72]. Second-order emergence implies that while the global (emergent) properties of a system are generated by the interactions of the individual components, the components somehow be-

come 'aware' of these properties and their individual behaviours are as a result influenced by them.

This downward causal link is of particular importance to social science, as the organisations that an individual exists within influence the behaviour of that individual. This is similar to the concept of "social functions"[32] – where the global emergent behaviour reinforces the micro-behaviours which generated it. It is also reflected in the simulated experiments of Wavish[161] (wherein high-level behaviours are defined in terms of emergent behaviours as detected by the agent itself).

Both Castelfranchi and Gilbert argue that some amount of sophistication within the elements of the system is required to produce such effects. However, it seems unclear why this is necessary – given that the system itself forms the environment with which each component interacts, changes in this environment will necessarily affect the way the component behaves, regardless of its internal complexity. While the 'macro-micro' link may be more apparent in systems containing cognitive agents, it should not be exclusive to them.

2.1.3 Strong and Weak Emergence

Bedau proposes that in order to select a precise and *useful* notion of emergence, definitions should be considered 'strong', or 'weak', paralleling the enduring 'strong' versus 'weak' debate surrounding classical artificial intelligence[1]. The distinction between weak and strong emergence is considered below.

Strong Emergence

The definition of 'strong' emergence given by Bedau is based on a conception of emergence defended by O'Conner[126]. Reformatted for clarity, it states:

> "Property P is an emergent property of a (mereologically-complex[2]) object O iff:
>
> 1. P supervenes on properties of the parts of O,

[1] "...according to strong AI, the computer is not merely a tool in the study of the mind; rather, the appropriately programmed computer really is a mind."[144]

[2] An object composed of several parts.

2. P is not had by any of the object's parts,
3. P is distinct from any structural property, and
4. P has a direct ("downward") determinative influence on the pattern of behaviour involving O's parts."[13]

The key aspect of 'strong' emergence is shared in common with "second-order" emergence as described above – the emergent behaviour exerts a downward influence onto the behaviours of a system's component parts. What is made explicit in this definition of 'strong' emergence, which is either missing or implicit in second-order emergence[72], is the *supervenience*[3] of the emergent property from the activities of the components of the system. This requirement causes some distress:

> "[Strong emergence] is uncomfortably like magic. How does an irreducible but supervenient downward causal power arise, since by definition it cannot be due to the aggregation of the micro-level potentialities?"[13]

The strong separation between the emergent property, and the component activities of the system which must be generating that property, seems to defy logic.

Weak Emergence

Bedau counters strong emergence with a definition of 'weak' emergence, as follows:

> "Macrostate P of [system] S with microdynamic D is *weakly emergent* iff P can be derived from D and S's external conditions but only by simulation."[13]

The external conditions referred to in this definition may be simply the initial state, in the case of a deterministic system. Most importantly, this 'weak emergence' definition carries with it the notion of *unpredictability*, a commonly encountered aspect of complex systems and chaos theory[43]. This conceptualisation of emergence also fits well with those provide by Holland ("Emergence occurs in systems that are generated."[85] p.225), Crutchfield ("A feature emerges when the system puts some effort into creating it"[41]) and Darley ("A true emergent phenomenon is one for which the optimal means of prediction is simulation."[44]).

[3]The appearance of something additional or extraneous.

Darley[44] illustrates this further with his definition of emergence:

> "Let $s(n)$ be the amount of computation required to simulate a system $[n]$...
>
> Our deeper level of understanding of the symmetries of the system... has allowed us to perform a creative analysis and deduce the future state whilst, we hope, circumventing most of the previous-required computation. Let $u(n)$ be the amount of computation required to arrive at the result by this method. [If:]
>
> $u(n) < s(n) \Rightarrow$ the system is non-emergent.
>
> $u(n) \geq s(n) \Rightarrow$ the system is emergent."[44]

Computational Irreducibility

"The system making the prediction must be able to outrun the system is is trying to predict. If the system is capable of universal computation, it cannot be outrun."[14]

The reasoning for this statement is as follows: if a system is capable of universal computation, then there are many questions regarding the behaviour of the system that are *undecidable*[159, 166]. It is therefore not possible to 'outrun' such a system, since a system's behaviour may be infinite.

For example, the simple cellular automata (CA) system studied in depth by Wolfram[165, 166] – 'Rule 30' – generates random structure when simulated, without falling into any cycle of behaviour, indefinitely. It is not possible to determine the configuration the the CA at time n without considering the system's state at each time period before n. In other words, it is not possible to 'short-cut' the behaviour of this system.

2.1.4 Detecting and Measuring Emergence

Despite the absence of a clear definition of emergence, a number of methods for detecting it have been proposed. The nature of these methods varies from statistical testing of various aspects of the system[167, 130], to attempting to map the dynamical envelope of a simulation[155], to openly subjective tests of 'surprise' in the observer[139]. Unfortunately, each of these methods has significant problems.

Nicolis demonstrates that emergent behaviour is

> "associated with the spontaneous emergence of lang-range spatial and/or temporal coherence among the variables of the (organised) system."[124]

Consequently, choosing the appropriate parameters of a system to model and measure is vitally important when trying to demonstrate – and detect – emergent phenomena[1]. Statistical methods of detecting emergence rely on appropriate choices of parameters, and these choices are typically made by the designer/observer. As Forrest notes:

> "[Emergence is] interpreted by the perceptual system of the person running the experiment. Thus, when conducting a cellular automaton experiment, researchers typically rely on graphics-based simulations to reveal the phenomena of interest."[61], p.3

Purely subjective tests of emergence such as the 'Design, Observation, Surprise!' method proposed in [139] suffer the same failings as subjective definitions of emergence. Differences in the abilities and choices of observers yield differing results, and improvements of the understanding of systems threaten any claimed emergence with 'demotion'[72]. It seems there is great difficultly in removing the requirement for an *active* observer from any means of measuring emergent behaviour. The interesting relationship between the behaviour of a system and and observer is presented in [102]:

> "In other words, the computational complexity [of a system] cannot be an intrinsic property of a physical system: it emerges from the interaction of a system state dynamics and measurement as established by an observer."

This relationship can be simply characterised by the common conundrum "if a tree falls in the woods, and no person is near, does it make a sound?", whose counterpart in the study of complex systems might be "if a system elicits some behaviour and no person measures is, is that behaviour still emergent?". The notion of 'sound' describes an experience produced in an observer who is measuring, biologically, changes in air pressure. Similarly, an emergent behaviour may only *exist* when a particular observable aspect of the system is measured.

2.1.5 Emergence and Self-Organisation

Systems which exhibit 'emergent properties' are often also characterised with the term **self-organised**. A concise definition of what constitutes self-organisation is given by Camazine et al.:

> "Self organization is a process in which pattern at the global level of a system emerges solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern."[29]

There seems little to differentiate this definition from a typical definition of 'emergence'. However, the perspective associated with self-organisation draws from a tradition of dynamical systems analysis[125], and as such draws on terminology from that field. The key characteristics which should be identifiable in a self-organising systems are[29, 17]:

Positive Feedback – as the individual components produce some pattern, other components respond to its presence and cause the original pattern to be reinforced.

Negative Feedback – other aspects of a system might cause the structure or behaviour to disperse. The presence of both positive and negative feedback results in semi-stable behaviour or structure.

Feedback leads to Amplification of Fluctuations – often the behaviour of a system is initially random, but over time certain random fluctations are reinforced (using positive feedback) whilst others are discarded (by negative feedback). In this manner, the system tends towards focusing its activity on a smaller number of behaviours or structures.

Multistable – there are typically a number of stable states for a self-organising system. Changes in the environment or other measurable parameters of the system may, if large enough, push the system from one stable state to another. The appearance of a qualitative change in the behaviour of a system when some measurable parameter of

the system is modified is termed a *bifurcation*, and particularly stable states are known as *basins of attraction*.

Local Information Flow – since the organising behaviour of the system is emergent, it must be a result of the interactions of the components within that system. As stated in [29] and above, in self-organising systems these interactions are typically local.

A Simple Example of Self-Organisation

Most examples of self-organised systems are taken from nature, such as pattern-formation on shells[119], or the coordination of social insects during construction[17, 28, 156, 94, 132, 76] or foraging[58, 64, 63]. For instance, the formation of paths by ants[58] (or ant-like robots[149, 54]; see below) features the creation of a self-organising path structure through the deposit of pheromones (chemical markers detectable by other individuals). Initially, individuals disperse through the environment, but once a food source has been located, the returning insect deposits a trail of pheromone back towards the nest. When other colony members encounter this trail (*local interactions*, via the environment; see **stigmergy** below), they follow it towards the location of the food, and strengthen the path markers by depositing further pheromone (*positive feedback*, *amplification*).

It is possible that many paths may be formed. However, shorter paths (those which can be traversed in the least time) will be reinforced more often, leaving longer paths unattended. While the pheromone trail on these longer routes is not being strengthened by continual deposits, it is also evaporating, and eventually long routes will disappear (*negative feedback*). Without direction, the colony has organised itself and selects the nearest food source for exploitation.

Further Reading

Self-organisation in biological and biologically-inspired systems is discussed in great depth in [29], to which the interested reader is directed for further examples and analysis.

2.1.6 Emergence in Abstract Systems

In an attempt to attack the 'problem' of emergence in a pure form, without contending with issues such as physical interactions and environmental modelling, a great deal of work has been undertaken which studies the emergent behaviour of very simple, abstract systems.

Boolean Networks and Cellular Automata

For example, Kauffman[100] and Solé[146, 147] both examine the emergent behaviour in boolean and multi-state networks. A 'boolean network', in this context, is simple a collection of boolean 'gates' which operate on the output value of other gates within the same network. It was found that while some networks (i.e. the values held by the gates) rapidly converged to a static state, other networks seemed to oscillate and produce patterns of 'behaviour'.

Wolfram [166, 165] and Langton[110, 109]'s investigations into the behaviours of simple, abstract systems show similar complex behaviour exhibited by simple, one-dimensional cellular automata (CA), whose low-level behaviour can be specified even more simply that Kauffman's networks. Patterns of behaviour within CAs are also often intuitively recognisable, given the spatial, graphic nature which is typically used to display them. Adamatzky and Holland[2] demonstrate the emergent formation of patterns in 'excitable media' using cellular automata. Adamatzky and Holland then [3], demonstrate similar pattern formation using models based on agents rather than passive cells. This type of emergence and the relationship between agents and excitable media have also been discussed recently by Bonabeau[18].

Frameworks and Relationships Between Abstract Emergent Systems

In his book entitled 'Emergence'[85], John Holland proposed constrained generative proceedures (CGP), simple production-rule systems for investigating the nature of emergence. Forrest [60] demonstrates emergent behaviour in the classifier systems, which are very close to the CGPs Holland describes. Forrest also describes a mapping between classifier systems and simpler Boolean networks, demonstrating the equivalence between these two simple state machines.

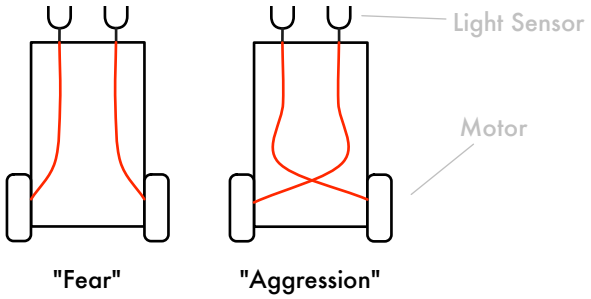


Figure 2.1: Two 'Braitenberg Vehicles'[25]. The speed of each 'motor' is directly proportional to the amount of light detected by the connected 'sensor'.

Earlier work by this author[1] has similarly compared multi-state networks (networks with gates whose set of output states is larger than two) with multiagent systems. Through the comparison between forms of these two systems, a mapping from abstract multiagent systems to multi-state networks has been demonstrated. It was also shown that the converse mapping is not possible

Emergence in Simple Agent Systems

Emergence has also been shown in systems with extremely simple agents. For instance, Channon and Damper[33], Ray[135] and Rucker[140] all demonstrate emergent behaviour in populations of abstract 'machine-code' organisms, whose intelligence amounts to little more than a few instructions. The Tierra simulation[135] in particular is a fascinating example of a 'digital ecology', in which organisms compete for computing resources of a computer system directly, rather than a simulation of a physical, tangible environment.

In contrast to the stark, almost-alien nature of Tierra agents, Braitenberg[25] describes a set of idealised, imaginary robotic animals, and demonstrates a wide variety of emergent behaviours – perceived by (naive) observers as 'aggression', 'love' and even 'optimism' – by making small modifications to the mediation between sensory information motor action with these 'Vehicles'. Internally, these creatures are nothing more than simple variations on the connections between primitive 'sensors' and 'motors', as shown in Figure 2.1. The speed of each 'motor' is directly proportional to the amount of light detected by the connected 'sensor'.

When the two vehicles shown are placed in an environment with an undirected light source, the first will initially move towards, and then rapidly turn away from the light. This behaviour is often perceived by observers (see Section 2.1.1 above) as "fear" (or "run-away"). The second vehicle behaves similarly at a distance from the light source, but close proximity causes the vehicle to move with increasing speed directly toward the stimulus: "aggression" (or "chase"). These simple agents clearly demonstrate the structured behaviours of light avoidance and light following, but there is no specification of such behaviour within the 'implementation' (Braitenberg conceived his 'vehicles' as thought-experiments). Furthermore, the appearance of high-level motives and 'emotions' can only be emergent *and* subjective to the observer, as there is certainly no specification of such behaviour within the specification of the agents themselves.

The *Nest* Model

The final example of emergent, abstract agent systems presented here is the *Nest* system developed by Bonabeau and Theraulaz [158, 156], which will become the basis for the research presented here. As such, the *Nest* model is described below, and again at length in the following chapter. While this system is based on existing biological systems, and can be discussed using significant physical metaphors (pheromones, building material, excavation and construction), the core of the system is very similar to the classifier systems or cellular automata described above. When stripped of concepts such as "agent" and "brick", the resulting model is a lattice network of cells which change state to form patterns within the spatial representation of the lattice, similar to cellular automata[4].

The *Nest* model is a rare example of abstract simplicity combined with a grounding in real physical systems[158, 156, 17, 29]. While it may be far from trivial to determine the practical advantages of a clearer understanding of cellular automata[166], the insights gained through analysis of this system will at least be applicable to physical (i.e. robotic) implementations of this emergent multiagent system.

[4]While the *Nest* model can appear similar to cellular automata when abstracted from its biological context, only a single site within the lattice is active at any single time. This would be classed a 'mobile automaton' according to Wolfram[166], or an 'asynchronous cellular automata' (see Section 3.6.2).

2.2 Emergent Multiagent Systems: Swarm Intelligence

While many examples of emergence are present in the most simple systems (such as cellular automata[166, 165] described above), emergent behaviour has also been exploited within the fields of multiagent systems and robotics. Within this context, the active entities are typically autonomous agents situated within either the real environment (in the case of robotics), or in in a simulated environment which mimics some or many aspects of physical reality.

The increased interest in emergent, collective behaviour as a flexible means of coordinating group activity is reminiscent of the introduction of behaviour-based robotics[26, 27] as a control mechanism for individual agents. These approaches are highly complementary, each relying on the specification of simple sub-units of activity, and relying on the interactions between these units and the resulting system output to provide the behaviour desired.

2.2.1 Swarm Intelligence

The most prevalent demonstrations of emergence in multiagent systems are currently limited to the reproduction of collective behaviours seen in nature, and in particular within the world of social insects. Several of the most prominent of these examples of emergent multiagent behaviour are discussed below. Additionally, excellent collections by Bonabeau et al.[17], Camazine et. al[29] and a survey by Steels[150] which discuss these systems and others in greater depth are also available to the interested reader.

2.2.2 Foraging

One of the most often cited demonstrations of emergent behaviour in multiagent systems is the 'Mars Explorer' system devised by Steels[149]. The scene is set:

> "The objective is to explore a distant planet, more concretely to collect samples of a particular type of rock. The location of the rock samples is unknown in advance but they are typically clustered in certain spots."[149]

Multiple agents must work together in an unknown environment, gathering these 'rock samples' in an efficient manner. Because the system is located on a remote planet, it must operate entirely autonomously; the distances are so vast that remote control is limited by the speed of signal transmission itself.

Path Formation

In Steels's simulated system, the swarm of collecting robots wanders randomly around the environment. If an agent detects a rock sample (this could equally be anything which must be collected) the agent returns to 'base' by following the increasing gradient of a homing signal. As the agent returns to base, it deposits 'crumbs', two at a time, along the path it takes to the base. Once an agent returns to base, it drops the sample and begins random-wandering again.

If an agent encounters a trail of crumbs, it will follow the trail, collecting a single crumb from each pair until it encounters the cluster of samples. After picking one up, the agent then returns to base, depositing pairs of crumbs in a similar manner to the agent which first discovered the cluster. In this manner, the path is reinforced whilst there remain samples to be collected, but once the samples cluster is fully eroded, the path will similarly be collected by agents following in anticipation of a sample. The path maintained by 'Mars Explorers' agents is an example of a dissipative structure[125], in which both positive feedback (agents reinforcing the path as they return with samples) and negative feedback (agents causing the path to 'dissipate' as they follow the path in search of samples) can be seen.

In experiments[149, 150], swarms of agents which used path-formation techniques collected all samples within the environment within a much shorter space of time that systems which employed only random wandering. This single model of foraging[58] contains many of the hallmark characteristics of emergent behaviour[29, 17, 150].

Agent Chain Formation

This foraging technique has been extended by Drogoul[54] by allowing agents to notify nearby agents if they are carrying samples, and allowing the transfer of collected material from one

agent to another. Drogoul's algorithm allows the agents to form *chains*, passing samples towards the base. This chain structure is far faster to adapt to changing environmental conditions, most notably the clearing of a cluster of samples, and therefore affords even greater increases in system efficiency.

2.2.3 Cooperative Transport

Ants can often be seen transporting prey larger than any of the individuals carrying it would be able to lift alone[63, 64, 106, 87]. To achieve this, multiple ants must coordinate their separate movements such that they can mutually transport the item, whilst simultaneously negotiating obstacles and soliciting further assistance if necessary.

This task has become a popular challenge in the field of robotics, and the design of of distributed algorithms to control a group of robots performing this task has become a 'benchmark for swarm robotics'[16]. Currently, this task is realised as box-pushing by two or more robots. Kube et al.[104, 106, 105, 107] have developed robotic swarm systems in which individual agents effectively coordinate their movements to push boxes towards a goal.

Related recent work by Parker et al.[131] takes inspiration from the 'blind-bulldozing' behaviour observed in some species of ant, and demonstrates cooperative movement of environmental material by a group of robots in a manner complementary to Steels[149] or the construction systems[158, 156, 19] which form the foundation for the original research presented here.

2.2.4 Clustering and Sorting

The techniques employed by Parker et al.[131] are similar to those required to achieve *sorting* and *clustering*, another renowned example of collective behaviour. Sorting and clustering behaviour is seen in ants[64, 46] in the form of the aggregation of corpses into 'cemetaries' and the sorting of brood (eggs, larvae, pupae, etc.).

Deneubourg[46] proposed a general model which is capable of reproducing both these behaviours. Essentially, agents wander randomly through the environment, and if an item is encountered it may be picked up. The probability of an agent picking up a given item is

inversely proportional to the number of other items in the neighbourhood. Agents carrying items will deposit their items using a roughly-opposite probability; that is, if an agent encounters a high density of items, and is carrying one at that time, it is likely to deposit the item it is carrying at this position. As will be seen be seen below, sorting can be achieved if the behaviour of an agent depends on the local density a *type* of object, and the type of the object currently being carried.

Robotic Clustering

This mechanism has been implemented in several robotic systems. Beckers et al.[12, 46] implemented a very simple collection of robotic agents which push small pucks around using a C-shaped scoop connected to a lever. The lever rests on a switch which is triggered when the resistance (due to friction with the ground) of the items trapped in the scoop reaches a given threshold. In this manner, the agent can be tuned to detect when it encounters several pucks together in the environment, since a collection of several pucks will exhibit higher resistance than a single puck. The robot is also equipped with the ability to detect collision with obstacles in the environment.

The robots may only exhibit one of three behaviours at any time:

1. If the trigger is not activated, the robot moves forward in a straight line.

2. If an obstacle is detected, it turns away at a randomly-selected angle, and returns to behaviour 1.

3. If the switch is triggered (indicating the robot has encountered a cluster of pucks in the environment) the robot reverses briefly, rotates by a random amount, and returns to behaviour 1.

Once running the robots move around randomly. If a puck is encountered, it will be trapped within the scoop as the robot moves forward (behaviour 1). Once several pucks have been encountered and present resistance to the scoop, the lever is pushed back and the switch triggered, activating behaviour 3. The robot backs away, leaving the collection of pucks where they are, and continues collecting in a different direction.

If any agent then encounters this small cluster, it will push any pucks it is carrying into contact with the cluster, detect the increased resistance of a cluster, reverse and continue away. By reversing briefly in step three above, the robot ensures that its random turn does not drag any pucks away from the structure in its scoop.

As the system proceeds, multiple clusters will be aggregated by the random pushing behaviour of the robots. However, if a robot approaches an existing cluster at a tangent, its scoop may drag a small number of pucks away from the cluster. These pucks will stay with the robot until it encounters another cluster. Given enough time smaller clusters will be eroded by this 'puck theft' at their periphery and the a single, larger cluster will be created.

Sorting

The behavioural model described above only demonstrates the clustering of objects within the environment. If the collecting probability is increased (and the complementary dropping probability decreased) where other objects in the neighbourhood are of a different *type*, such systems also exhibit *sorting* behaviour – the resulting environment will contain piles of distinct object types.

Holland and Melhuish[86] have implemented a similar system to the clustering robots described above[46, 12]. This robotic swarm uses two different types of item (in this case black and yellow frisbee discs). By enabling the robots to distinguish between types of item in the environment, the swarm can cluster certain types of disc together, or in other words, sort the objects within the environment. This is illustrated clearly in the time-lapsed images shown in Figure 2.2, and the sorting behaviour can be seen most clearly in the final image.

By modifying the conditions in which an object will be picked up to depend on the local density of that type of object, the clustering behaviour described about can be adapted to produce a form of sorting. If an object is encountered and these is a high density of similar objects, if will be ignored. However, if there is a high density of dissimilar objects, then this 'stray' should be collected by the agent.

Likewise, if an agent is carrying an object of a particular type, and encounters a high density of objects of the same time, it should deposit its object in that location. If the

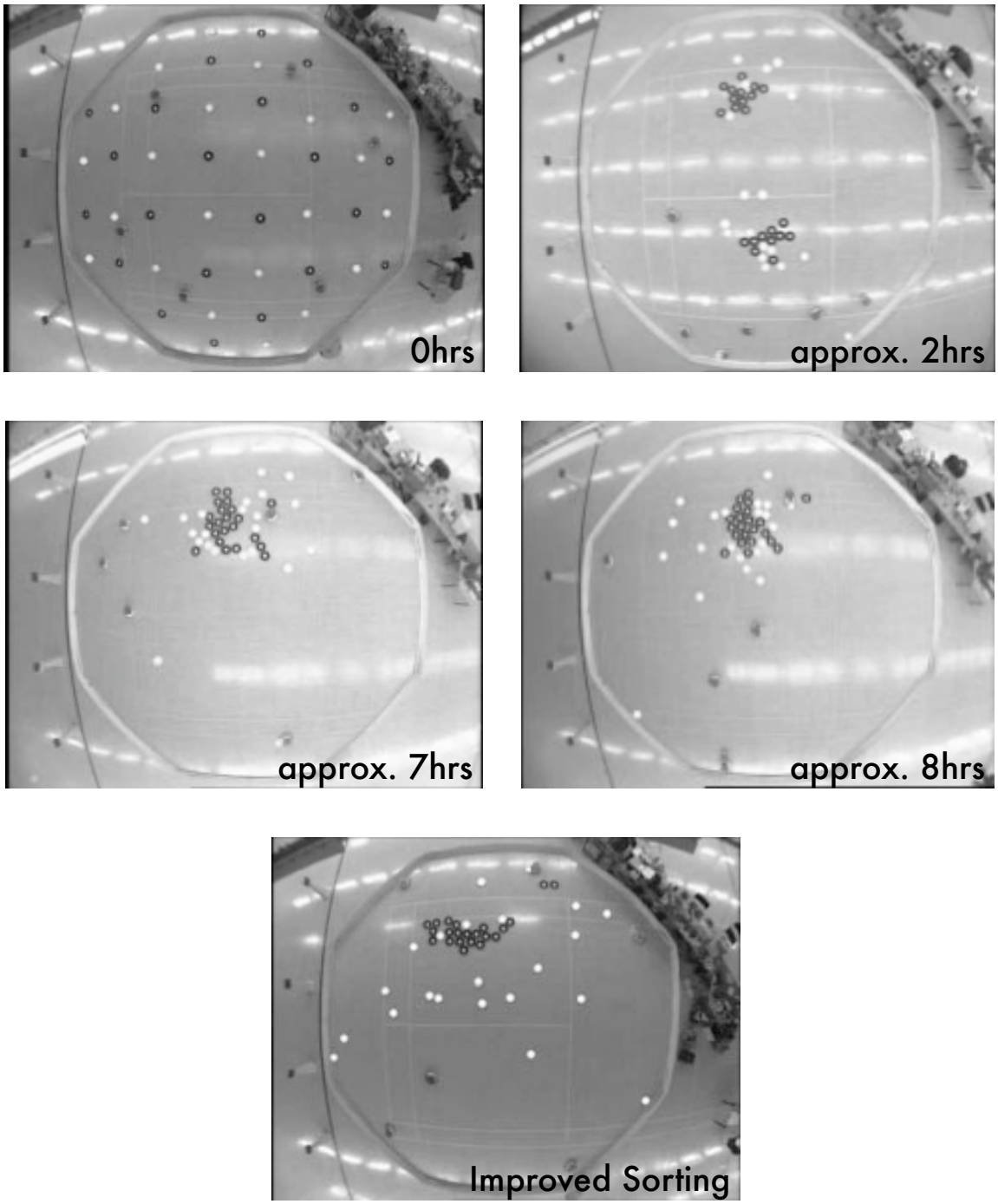


Figure 2.2: A robotic sorting implementation, using two colours of frisbee. Initially, several groups of objects are formed. Eventually, one cluster is eroded and a single cluster is formed. The lower image depicts the system after adjustment to more clearly demonstrate annular sorting; the black discs have been clustered in a central pile, and the yellow discs evenly spread in a surrounding 'aura'.

objects within the environment are of a different type, it should simply avoid this cluster and continue moving through the environment.

Other Applications of Clustering

The techniques described above have also been applied to abstract optimisation problems such as graph partitioning[113] and exploratory data analysis[108]. Further details of the more-abstract applications of this technique can be found in [17].

2.2.5 Construction and Coordinated Assembly

Perhaps the most striking example of coordinated activity in social insects is the construction of nest structure, which support the life cycle of often vast numbers of individuals[17, 64, 87]. The disparate actions of multitudes of individuals combine to produce complex a structure serving many functions, from housing the colony to protecting young to storing food supplies.

The *Nest* system implemented by Bonabeau and Theraulaz[158, 156, 22, 19, 29] (mentioned previously in Section 2.1.6) is the most prominent simulation of emergent swarm construction, where 'virtual wasps' interact to produce complex arrangements of bricks. Their model relies upon the segmentation of space into a lattice of discrete cells, which may be empty, or may contain a piece of matter, otherwise known as a *brick*. Agents move around within the lattice examining local arrangements of cells ('neighbourhoods') and comparing them against an internal set of stimulating configurations which might indicate the placement of a brick in the current location. Multiple types of brick (or 'colours', for simplicity of interpretation) can be placed, and recognised by agents during simulation.

Bonabeau, Theraulaz et al.[156, 22, 19] have shown that certain collections of these building rules will produce structures in space which appear complex and would typically require a great deal of coordination to produce using traditional planning mechanisms. Furthermore, within this simple framework it is possible to derive rule sets which will produce structures similar to those seen in nature, such as wasp nests.

While the *Nest* model is by far the most often-cited investigation into swarm construction, other notable work involving construction using simple agents includes investigations by

Karsai[95, 94, 96, 99, 97, 98], Mason[115], Parker[131], Coates[36] and Bowyer[23].

Self-Assembly

Construction by a swarm of agents shares much in common with the concept of self-assembly. In both, simple components arrange material within space, constrained by limited individual cognitive and perceptual abilities. The significant difference in the case of self-assembly is that the active components (the agents) and the material of construction are one and the same. Self-assembly is seen within insect colonies, most notably the formation of bridges by ants, linking their bodies together to bridge gaps[64, 17].

Self-assembly is of particular interest to the new areas of micro-mechanical components and 'nanotechnology'[53]. At such small scales, it becomes increasingly impractical to build machinery and artefacts using external forces. If structures could be caused to assemble themselves dynamically, in a manner similar to the bonding of molecules into compounds[164], the construction of nano-scale structures may become feasible.

Self-assembly can also be useful at macroscopic scales, and it has been demonstrated in a number of robotic systems[65]. Melhuish et al[120] demonstrate the formation of barriers by ant-like robots using stigmergic communication. Pamecha et al.[128], Murata et al.[122], Hosokawa et al.[89] and Yoshida et al.[170] demonstrate several mechanisms for self-assembling and self-reconfiguring robotic systems.

The recent work of Jones[92] on abstract self-assembling systems is very closely related to the *Nest* systems devised by Bonabeau and Theraulaz[158, 156]. The agents (or 'modules' in this case) arrange themselves within a regular lattice, and rules of construction based on stimulating local configurations are used to determine which locations around the structure modules can be placed as the simulation progresses.

2.2.6 Ant Colony Optimisation

Ants working together form trails to food sources. By use of pheromone deposits, the ant colony as a whole establishes the shortest (fastest) route between the nest and the location of the food.[87, 64, 17, 11, 163].

The trail-making behaviour of ants foraging for food has inspired one particularly notable application of swarm intelligence. The ability of a distributed colony of simple individuals to find optimal solutions to problems involving a 'journey' has obvious applications in the field of combinatorial optimisation[16]. Other problems to which the ant metaphor lends itself well include the dynamic optimisation of packet transmission in telecommunications networks. These examples of **Ant Colony Optimisation** (ACO) will be discussed briefly below.

The Travelling Ant

The 'travelling salesman' problem (TSP)[93, 112] is a classic NP-complete combinatorial problem[71]: given a finite number of 'cities' with some pre-determined cost of travelling between each pair, we must find the cheapest route which allows the salesman to visit all cities and return to his starting point. If links exist between all cities, then the total number of tours is given by $\frac{(n-1)!}{2}$, where n is the number of cities the salesman must visit. Further aspects of the TSP will become relevant to the research presented later in this thesis (see Section 8.2).

The similarities between this abstract shortest-path problem and the pathfinding behaviour of ant colonies within the natural world are self-evident. As a result, algorithms inspired by the behaviour of ants, and in particular the virtualisation of mechanisms which they use to optimise paths (pheromones) have enabled the development of high-performance ant-based routines for solving TSP problems[49, 48].

Network Routing with Ants

An extremely practical application of ACO appears in the form of telecommunications routing[20, 141, 69]. The problem within this domain is to maximise the routing performance of a (possibly heterogenous) network under a constantly varying traffic load. This problem differentiates itself from the TSP above in that the structure of the network (both in terms of connections, and the cost of transmission over those links) is dynamic, and therefore may be subject to change while the system is operating.

Schoonderwoerd et al.[141] developed a system named *ABC* (ant-based control) which models the routing behaviour of a typical telephone network. In this domain, communication between nodes is achieved by circuit-switching (rather than packet-switching, see below). 'Virtual ants' are periodically sent from a random node to a random destination node within the network, depositing virtual pheromone along each link in that journey. The amount 'deposited' at each node is inversely proportional to the amount of time the ant spent waiting in the network to reach that node. In this way, nodes which are congested will have little pheromone deposited, whilst nodes which have spare capacity will receive stronger deposits.

Using this mechanism, the routing tables for each node can be updated to determine, dynamically, which nodes in the network have the capacity to support new connections, and which are congested and should not be used in routing at that time. In comparisons with traditional routing algorithms, the ABC system has been shown to yield significantly superior performance[141, 17], especially when the load on the network is varying significantly.

Modern computer networks (such as the Internet, for example) employ packet-switching rather than circuit-switching in order to establish communications between two nodes. Di Caro and Dorigo's *AntNet*[69, 68] have adapted the 'virtual ant' mechanism in ABC for use in both types of network in the new system *AntNet*. When compared against algorithms currently used to route traffic through the internet, *AntNet*'s performances was shown to be at least as good as (and often significantly better than[68, 17]) other algorithms.

Summary

It is clear that the behaviour of ants can not only be the inspiration for novel solutions to abstract combinatorial optimisation problems[49, 48], but also be applied in practical situations whilst yielding excellent performance over traditional distributed optimisation methods[51, 69, 68, 141].

ACO along with many additional applications of swarm optimisation are discussed in detail by Di Caro and Dorigo in [69], and Bonabeau, Dorigo and Theraulaz in [17]. The reader is directed there[5] for more detailed information regarding its application and characteristics.

[5]Frequently updated information regarding ACO, including publications, bibliographies and conferences, can also be found at Dorigo's homepage: http://iridia.ulb.ac.be/dorigo/ACO/ACO.html

2.3 Stigmergy

All emergent multiagent systems feature individual entities which interact with each other to produce a collective behaviour. Interaction between individuals can be considered a form of *communication*, and may not be direct (agent to agent). Instead, interactions may occur indirectly by modifying the environment. In this section, we describe one common means of indirect communication in swarm intelligence systems, inspired by the interaction of social insects.

2.3.1 A Brief History

The term **stigmergy** was originally formulated by Pierre-Paul Grassé, as a means of explaining the observed coordination in nest construction by *Bellicositermes* termites[76]. Previous to this landmark study, two near-opposite schools of thought presided over the explanation of social insect behaviour. The **organicist** school considered insect colonies as examples of a single *super-organism* (based on Alfred Espinas' application[56] of the metaphor devised originally by Herbert Spencer[148]), by observing that the tendency for individuals to collect together into societies mirrors the tendency of cells to form multicellular organisms. These similarities were also noted by Wheeler in the famous paper *An ant colony as an organism*:

> "An organism is a complex, definitely coordinated and therefore individualized system of activities, which are primarily directed to obtaining and assimilating substances from an environment, to producing other systems, known as offspring, and to protecting the system itself and usually also its offspring from dangers emanating from the environment. The three fundamental activities enumerated in this definition, namely nutrition, reproduction and protection, seem to have their inception in what we know, from exclusively subjective experience, as feelings of hunger, affection and fear respectively."[163]

This was based on the observation that both individuals and animal societies share several common features, including acting as a single unit, exhibiting behaviours often characteristic of only that species, and undergoing cycles of growth and reproduction. In this manner,

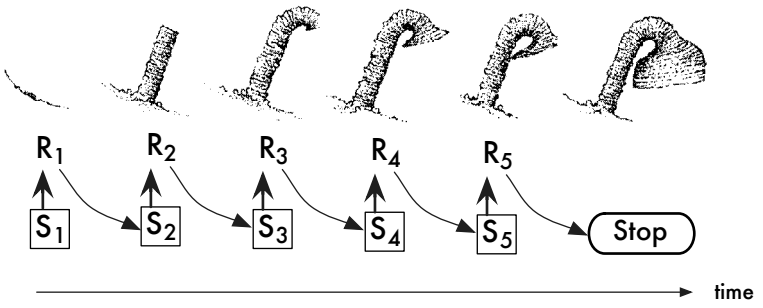


Figure 2.3: The sequential building activity of the wasp *Paralastor*. Adapted from [157].

the behaviour of an entire insect colony could be explained using the same techniques and motivations as if it were a single creature.

The alternative, termed here the **analytic** view, insists that what is observed as the behaviour of an entire animal society can *only* be the result of the behaviours of the individual entities within that collective:

> "... all these individuals are working... the "collective" work is only the juxtaposition of individual works... [T]he common work is no more than a side effect of the interattraction that gather[s] individuals together."[134]

Grassé's 'stigmergy' (coined as a conjunction between the Greek *stigma*, to sting, and *ergon*, to work) demonstrated a bridge between the actions of the individual and the actions of the insect colony as a whole. In the original termite study [76], it was shown that the coordination and regulation of the building behaviour of multiple termites was guided by the structure itself. Each individual's actions are determined by the current state of the local environment. A stimulating environmental configuration of material triggers building behaviour within the individual, which in turn transforms the environment, resulting in a new local configuration of material, which may (or may not) trigger further actions.

This behaviour is also clearly demonstrated in the study of the solitary wasp *Paralastor* by Smith[145], illustrated in Figure 2.3. During stigmergic construction, at the end of each building stage the resulting structure acts as the stimulus triggering construction of the subsequent building stage.

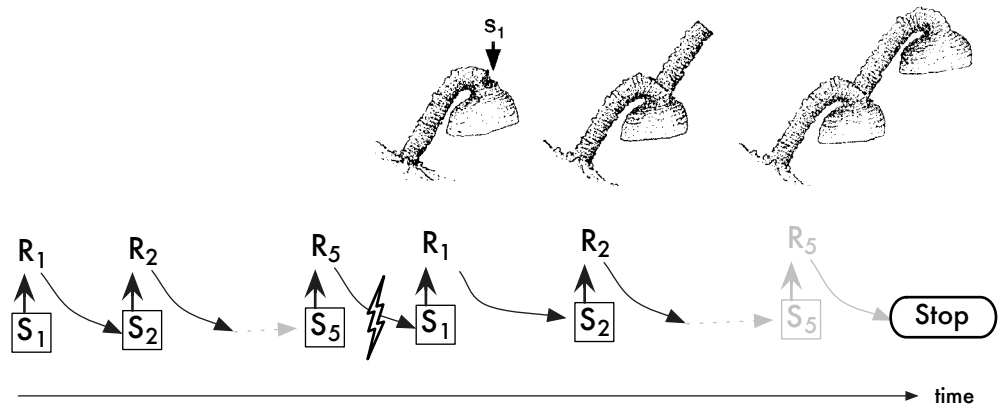


Figure 2.4: When a stimulating environmental configuration is created out of sequence, the pathological building behaviour of the stigmergic agent is revealed. Adapted from [157].

Stigmergic Construction Pathologies

However, the clearest example of the nature of stigmergic behaviour occurs when a stimulating configuration is created *out of sequence*, as illustrated in Figure 2.4. In this figure, once Stage 5 is achieved a hole is created, mimicing the stimulating configuration for Stage 1. Upon detection of this configuration, the building behaviour appropriate for State 1 is triggered within the wasp, and a new funnel is created and developed from this point.

While the examples above consider a solitary insect, if two or more insects do not (or cannot) distinguish between each other's work, one could easily continue the construction initiated by another. It is this type of indirect interaction implied by Grassé with the term *stigmergy*. For a more detailed discussion of the history of stigmergy, the reader is referred to Theraulaz & Bonabeau's excellent Special Issue of the Artificial Life journal[157].

2.3.2 Types of Stigmergy

While the principle of stigmergy – the guidance of agent behaviour by previous modification to the environment – is now clear, it is at best a general principle. It remains unclear exactly *how* changes in the environment might affect the behaviour of agents. The two commonly cited 'types' of stigmergy are **quantitative** and **qualitative** stigmergy. These differ in the type of stimuli which may trigger action in participating agents.

2.3.3 Quantitative Stigmergy

In quantitative stigmergy, stimuli are present in different *quantities*. The most obvious examples of such a stimulus are **pheromones** – chemical markers deposited by insects either onto material within the environment, or directly into the environment itself. Differing quantities of pheromone are then detected throughout the environment as the chemical *disperses*. Further away from the original deposit, the strength of the marker is weaker. Additionally, the strength of the chemical 'scent' decays over time (unless it is strengthened through further pheromone deposits).

In Grassé's original stigmergy research[76], termites marked pellets of soil with pheromone. These pellets are then randomly deposited in the local environment. Where these pellets are deposited close together, the cumulative strength of the pheromone encourages further deposits, and additional material is placed at such sites. In this manner, eventually certain sites (those where sufficient amounts of soil pellets were randomly placed) will have accumulated a sufficiently high concentration of chemical marker that most building activity will be focused there, while in other locations within the environment, the decay of the pheromone reduces the likelihood of further construction. This focussing of construction in certain region produces the individual pillars seen in termite nests.

Further examples of quantitative stigmergy can be found in the ant-inspired robotic sorting work of Deneubourg et al.[46, 45]. Mason's model of nest construction[115] depends on the geometry created by interacting pheromones. Quantitative stigmergy can also be considered as the mechanism underlying ant foraging behaviour and shortest-path determination[132, 12, 11]. The emergent multiagent system by Steels[149] (described previously) also demonstrates this pheromone-based coordination via the environment.

2.3.4 Qualitative Stigmergy

In contrast with quantitative stigmergy above, a system exhibits qualitative stigmergy when the agents respond to a discrete set of *types* of stimuli. The building behaviour shown in Figures 2.3 and 2.4 demonstrate such a system. At each stage, the arrangement of material in the environment presents the agent with a distinct *stimulating configuration* (S_1, S_2, and

so on). Whenever the agent encounters S_1, the agent performs the response action for that stimulus (response action R_1 in that example). In this manner, an agent's behaviour is regulated by a set of condition-response rules, triggered whenever a matching environment is encountered.

Excellent discussions of qualitative stigmergy, and the nest-building behaviour of wasps in particular, are available in work over the past ten yeras by Theraulaz, Bonabeau et al.[157, 156, 17] and Karsai[98].

Sematectonic Stigmergy

When communication takes place *only* through manipulation of material the environment, this form of behaviour is defined as **sematectonic** stigmergy[31, 129]. The best example of this type of behaviour is given in the nest-building lattice systems described by Theraulaz, Bonabeau et al.[158, 156, 29, 17], and similar systems by Karsai & Theraulaz[95, 94, 96, 99, 98], where the environmental changes sensed by agents are the developing nest structures themselves. Sematectonic stigmergy can also be seen in the robotic collecting behaviour demonstrated by Holland et al.[12, 86], as the environmental modifications are part of the goal-directed behaviour.

Sematectonic stigmergy and the work of Theraulaz, Bonabeau et al. form the basis of the original research presented within this thesis. Accordingly, their research with these systems will be discussed at length in Chapters 3 and 5, and throughout the remainder of this thesis.

2.3.5 Alternative Stigmergic Mechanisms

One aspect of stigmergic behaviour which remains unconsidered is highlighted by Holland in [86]. In order for stigmergy to be present, it may be sufficient that changes in the environment only *modify the outcome* of an agent's existing behaviour, rather than cause the agent to select a qualitatively different behaviour. In other words, the previous behaviour of other agents may have a qualitative or quantitative effect on some measurable parameter of the outcome of the current action.

In this context, Holland[86] also notes an interesting third potential stigmergic mechanism

– **passive stigmergy**. This type of environmental interaction lacks the notions of agency required for both quantitative and qualitative stigmergy:

> "Consider a car being driven along a muddy track. Although the driver might try to steer a particular course, the wheels may settle into deep ruts that take the car along another course. The actions taken by previous drivers have affected the outcome of the actions taken by the present driver."[86]

This notion of stigmergy approaches the purely physical behaviour demonstrated the effects of environmental forces within nature – the change in path taken by a river as a result of erosion caused by its own flow, for example . However, it is important to understand the distinction between *action* and *outcome*, as highlighted above.

2.3.6 Some Stigmergic Systems

Many of the emergent multiagent systems described above employ stigmergic interactions to achieve their intended collective behaviour. For instance, Steels' trail-laying agents[149] coordinate their actions by communicating through the environment, and each agent bases its current selection of activity only on local environmental information.

The clustering described by Deneubourg[46] and robotic implementations by Beckers et al.[12] and Holland[86] exhibit sematectonic stigmergic communication: as each agent modifies the environment, larger clusters are formed and the individuals begin to 'cooperate' in building fewer, larger clusters of items.

The models of nest excavation presented by Jérôme et al.[28] and Bonabeau[18] demonstrate the construction of 'galleries' (tunnels) using stigmergic environmental modification and trail-laying. Jérôme[28] also demonstrates collective adaptation – as factors (such as colony size) are modified while the system operates, the nest structure is automatically adapted to accommodate the necessary changes.

Finally, the abstract models of nest construction presented by Bonabeau and Theraulaz[158, 156] feature agents which interact only using sematectonic stigmergy: as the nest architecture is constructed by the placement of bricks, the stimulating configurations present in the environment change and affect the building behaviour of the agents.

An extensive bibliography of systems which either discuss or implement some form of stigmergic swarm intelligence has been collated by Shell[6], and the interested reader is directed there for further examples of stigmergy in both simulation and robotic implementation.

2.3.7 Stigmergy, ACO and Swarm Intelligence

It is worth remarking that stigmergy may be present in systems containing only a single agent (such as the *Nest* systems[158, 156]), and as such may not *strictly* be an example of 'swarm intelligence', given the obvious lack of 'swarm'. It can also be demonstrated that ant colony optimisation[17, 50, 69, 51] (ACO), the most popular and prominent example of swarm intelligence, may also be accomplished with a single agent[35], Given a pheromone which can be reinforced by a single agent faster than the marker decays, a single agent is capable of producing a shortest route solution.

However, such a harsh distinction discredits the significant cumulative speed-up which occurs when more than one agent is present in either of these systems. Nevertheless, ACO examples fulfilling such criteria will fall into the same category of emergent systems as the stigmergic systems which are the basis of the remainder of this thesis.

2.4 Designing Emergent Multiagent Systems

> "Swarm intelligence [is] "the interplay of computation and dynamics". The goal... is to explore that interplay and determine how to best, and most simply, design the computational component in order to take advantage of the dynamics" [117]

A constant underlying motivation behind any investigation into stigmergy, emergence and self-organisation is the application of these mechanisms towards solving novel problems. Biologically-inspired techniques[29, 17] have yielded excellent results in the fields of optimisation (e.g. Ant Colony Optimisation[51], above). However, the application of collective intelligence mechanisms to problems which are further removed, or less clearly analogous to their biological counterparts remains far more elusive. Very little work exists regarding how

[6]Dylan Shell's extensive annotated 'stigmergy' bibliography is available (as of January 2005) at http://robotics.usc.edu/~dshell/stigmergy.php

the emergent behaviours taken from biological and social systems can be applied to solve arbitrary problems[17, 150].

2.4.1 Controlling Emergence via *Symbolic Behaviours*

One of the first attempts to formulate a practical approach to building systems which exploit emergence is presented by Wavish[161], who suggests creating a set of *symbolic behaviours* correlated with the presence or absence of the corresponding emergent behaviours within an agent.

For instance, if a set of basic behaviours[7], such as `avoid_wall` and `move_toward_wall`, are known to produce an emergent behaviour, when these basic behaviours are active the symbolic behaviour `wall_following` is turned "on" within the agent. In a similar manner, other emergent behaviours may be "on" or "off" depending on which basic actions are currently being performed. Designers are then free to program higher-level activity based on the presence of these emergent behaviours. Wavish presents an implementation demonstrating a simulated dog herding together a flock of sheep. Emergent behaviours such as `drive_sheep` are composed of `round_up_sheep_left` and `round_up_sheep_right`.

Working with emergent behaviours is this manner, however, requires the designers to be able to accurately determine which basic behaviours might combine to produce an emergent effect. Furthermore, the agent must be able to *recognise* the appearance of these emergent behaviours and internally 'note' whenever some symbolic behaviour might be active. In effect, the designer must still manage the emergent behaviour by hand, and 'symbolic behaviours' simply act as a grouping of some pre-determined set or sequence of primitive actions upon the world.

2.4.2 From Simulation To Robotics

In [116, 117, 118], Mataric presents a troupe of robots, affectionately dubbed the "Nerd Herd", upon which a variety of emergent behaviours have been implemented. In a similar manner to Wavish, higher-level behaviours are composed from the simultaneous activation of

[7]Basic behaviours may be movements or other primitive and most likely physical actions.

'basis behaviours'[8] – `following`, `dispersion`, `aggregation`, `homing` and `safe-wandering`.

These behaviours can then be run in parallel (in a similar fashion to Brooks' Subsumption Architecture[26]) upon embedded processors within the robotic agents. Each basis behaviour suggests an output for the motors controlling the movement of the robot, and these outputs may be summed together, or subtracted from each other, to derive the final output to be sent to the actuators themselves. For example, 'flocking' can be achieved by summing the outputs of:

- `safe-wandering` – preventing the agent from moving such that it would collide with any objects
- `disperse` – checking the proximity of this agent to its nearest neighbours, and moving away from the calculated centre of local agent density (the 'centroid'[117])
- `aggregate` – moving towards the nearest agent, if that agent is beyond a certain threshold
- `homing` – moving the agent towards some identified location within the environment

By summing the suggested movements of all these basis behaviours, the net result is that the agents will move together as a pack, maintaining a steady distance from each other while avoiding any objects, and moving towards a common destination; in order words, moving like a flock of birds, or a shoal of fish, through the environment.

Despite the apparent success of the composition of emergent behaviours in a multiagent system, it remains unclear how these 'basis behaviours' might be composed to perform behaviours outside of the already well understood flocking[137], sorting[12, 86], foraging[149, 54] and other behaviours lifted directly from the biological context from which they were inspired. The important question must be raised – can multiagent systems be coerced into demonstrating emergent behaviours which are unlike those seen in social insects?

[8] Basis behaviours themselves are actually short algorithms build from primitive actions and sensor conditions.

2.4.3 Designing Emergence with Stigmergic Construction

The seminal lattice swarm experiments by Bonabeau, Theraulaz et al. attempted to explore the potential of (qualitative) stigmergic systems using random[156, 158, 17, 29] and then genetic algorithm search techniques [22, 19]:

> "We undertook a modelling approach by looking for the simplest automata that could generate complex architectures similar to those observed in nature... Then by working "backwards" from the shapes to be generated to the potentially corresponding algorithms, we eventually found it possible to produce complex shapes, some of them strikingly similar to those seen in nature"[22]

It is therefore clearly possible to produce 'interesting' architectures using their *Nest* model, and additionally possible to reverse-engineer stigmergic algorithms which could produce individual structures. However, for such a system to

> "...provide engineers with valuable insight when they need to design programs or machines exhibiting collective problem solving abilities..."[22]

it is insufficient to merely show that *some* structures can *possibly* be generated. Instead, an engineer must know when it is appropriate to apply stigmergic techniques, and when there is little benefit. Furthermore, no comment was made regarding the complexity of such reverse-engineering efforts. This may be likened to the computational power of cellular automata (CA); while it is possible to design and build universal computing devices within the substrate of a CA[166], it is far from productive to do so, in engineering terms.

Existing Stigmergic Design Work

The only existing work in the stigmergic arena which attempts to generate a methodology for the design and application of stigmergy to arbitrary tasks is presented by Mason[115]. This approach, as the author dubbed 'stigmergic programming', uses stigmergic behaviour triggered by the intensity of two pheromone gradients on a two-dimensional plane. This is illustrated in Figure 2.5. Two pheromones (A and B) are initially deposited in the environ-

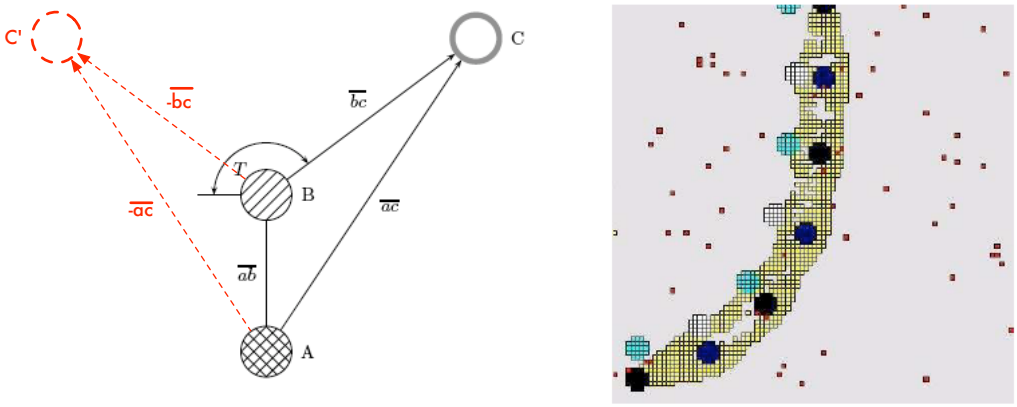


Figure 2.5: Mason's 'stigmergic programming'. The next location of building is defined by rules specifying the strengths of *both* pheromones A and B. On the right, the resulting curved wall of construction is shown. Figure adapted from [115]; the dashed red vectors are discussed in the text.

ment. The nest building location (C) is defined by specifying the relative strength at this location of *both* pheromones (as indicated by the magnitude of vectors $\overline{ac}$ and $\overline{bc}$):

> "A rule triggered by a band of intensity of a particular pheromone will, given a single deposit of that pheromone, be triggered in a ring centered on that deposit. Adding a second deposit breaks radial symmetry and creates an ellipse-shaped activation potential. By stringing together ellipses, directed chains of deposits can be formed."[115]

However, as shown in Figure 2.5, this does not fully specify the building behaviour of the swarm, since an identical 'ellipse-shaped activation potential' exists in the mirror symmetry through the original deposits. This is shown in the red dashed modifications in Figure 2.5. In Mason's original work, only point C is considered as the next site of activity. However, point C' exists at equal distances from A and B, so it would be impossible for an agent to determine whether or not its current location was C or C' based on the pheromone densities alone. While it is possible to counter this if the agent has access to some form of global information (i.e. an internal compass and the ability to detect the gradient itself, rather than just its intensity at given points), such measures remain unconsidered in Mason's original work.

2.5 Initial Conclusions

From this survey of existing, related work, it can be seen that 'emergence' is a subject which continues to inspire vigourous discussion. An steadily increasing wealth of research into systems which may claim to demonstrate such a quality is also being produced. In particular, there is a significant focus on systems which feature *stigmergy* as one of the interaction mechanisms through which the collective behaviour is achieved.

As the terminology which describes their behaviour is debated, insights from related fields such as the study of complexity[102, 55, 77, 100], self-organisation[5, 86, 29], biology[163, 64] and social science[38] will continue to fertilise this intriguing approach to understanding and manipulating complex systems.

For the purposes of the original work presented hereafter, the philosophical quagmire which surrounds emergence should not be taken too seriously. It is sufficient here to accept Bedau's "weak emergence"[13] (see Section 2.1.3) and focus not on the implications of this choice, but rather on the more practical aspects of controlling the global behaviour of a collection of autonomous but simple agents.

Bonabeau clearly describes the issue faced by designers of emergent systems:

> "Given a problem, what should be the specifications of individual agents and the pattern of their interactions so that the dynamical evolution of the collection of agents leads to a reasonably good solution to the problem?"[18]

Unfortunately, little progress has been made in the development of practical emergent design. The design of emergent multiagent systems currently relies on references to a catalogue of instances (both from recent research and the natural world), but this may have contributed to the limited number of real-world applications of emergence and stigmergy[51, 50]. It would seem foolish to *assume* that the scope of emergent behaviours is limited to applications which neatly fit into such natural metaphors.

The original work presented here attempts to go some way towards testing the behavioural envelope of stigmergic systems, and providing a new set of tools with which we can explore, and exploit, these capabilities.

Chapter 3

Nest-2.11.1 – Nest Building using Discrete Stigmergy

Overview

In this section the existing work by Bonabeau, Theraulaz and their co-workers[158, 156, 22, 19] with regards to abstract stigmergic systems is reviewed. Their original simulation system – *Nest-2.11.1* – is described, along with the methods and findings of their work with this software.

In [156, 158], Theraulaz and Bonabeau set out to "explore the space of possible architectures that can be generated with a stigmergic algorithm". They then set out a simple model – outlined below – which can be used to simulate the natural building process, and assert certain characteristics which are required for the generation of nest-like structures. The size of the space of possible stigmergic algorithms is also noted, and the constraints a space of this magnitude brings to a systematic exploration of the behaviours of stigmergic systems, even within their limited model.

The work presented in [22, 19, 17] describes the use of genetic algorithm techniques to search the space of algorithms. This reintroduces the problem of determining the fitness function for a structure, along with some fundamental problems regarding the evolution of algorithms in general. A further discussion of this work is found in Section 3.4.

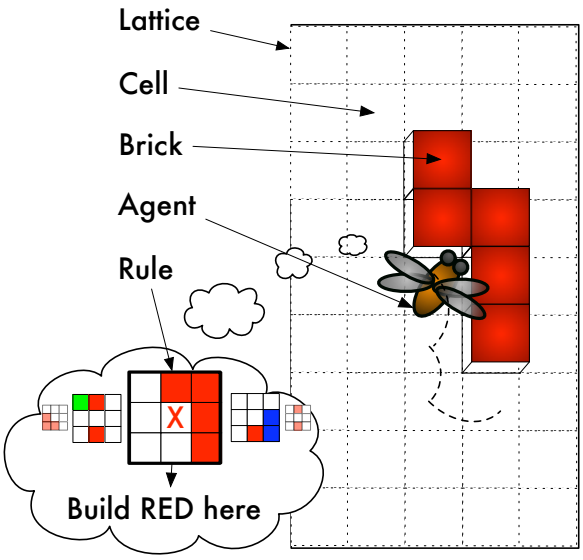


Figure 3.1: A simple representation of the main features of a lattice swarm system.

First however, a simple description of the model which forms the basis of the rest of this research is presented below.

3.1 Lattice Swarms

The model presented by Bonabeau et al.[158, 156, 22, 19], and implemented with their *Nest-2.11.1* software, is an example of an extremely simple system which is capable of demonstrating some of the construction behaviour seen in social insects. This model relies upon the segmentation of space into discrete, uniform regions hereafter referred to as *cells*. These cells are arranged in a regular manner so as to form a space-filling *lattice* of discrete locations within the entire space being modelled. Cells may be empty, or may contain a piece of matter, otherwise known as a *brick*. A brick completely fills the region of space represented by the cell, such that each location within the lattice can hold one and only one brick at any time. Two distinct types of brick are available.

Bricks are 'deposited' by an asynchronous automaton – the *agent* which represents the social insect. This agent is capable of moving around arbitrarily within the three-dimensional space described by the lattice of cells, moving from one randomly-selected empty cell to another. Once a brick has been placed within the lattice, it can neither be removed nor

modified in any way. Initially, the lattice contains a single brick.

At any point in time the agent may occupy a region of the lattice whose local arrangement of bricks and empty cells corresponds to a pre-determined **stimulating configuration**. The agent has access to a list of such configurations, otherwise known as *rules*. The list of rules is also called a **stigmergic algorithm** or **stigmergic script**. If an agent encounters a local configuration which matches one of the rules within the algorithm, a brick is placed in the agent's current location. The type of brick placed is determined within the rule.

This model describes the essence of the abstract system upon which the rest of this work is based.

3.1.1 Terminology

The description given above is a complete but very compact overview of the type of system considered in the rest of this thesis. While the systems later might relax or remove some of the constraints outlined above, the fundamental elements – cells, bricks, and rules – are constant

A Cell is the fundamental unit of space within the stigmergic system. Conceptually, a certain location space can be empty or filled, and an agent must occupy a certain region of space. A **Cell** defines the limits of this region, and the geometric relationships between different regions of space. Internally, a cell simply has a certain *state* or *value*, which determines whether or not it is filled or empty, and the type of brick present if it is filled.

A Brick is therefore simply a *filled* **Cell**. Bricks can be of various distinct 'colours', which are actually represented by the subset of cell values which does not include `EMPTY`. To clarify even further: discussion of *bricks* and *colours* is a nod towards the biological inspiration for this model; *cells*, *cell values* and *cell states* are indicative of the underlying model itself. Furthermore, the number of brick colours is generally one less than the number of valid cell values within a stigmergic system, since cells may also have an `EMPTY` value.

A Neighbourhood is simply a collection of cells (*neighbours*) which are immediately spatially adjacent to one cell in particular (the *central cell*); each cell has surrounding it a *neighbourhood* of cells.

A Rule is an arrangement of cells, identical in size and configuration to a **Neighbourhood**. Whereas a neighbourhood is a subset of the cells *in situ* within the global simulation space, rules do not *exist* within the spatial lattice; they are *examples* of neighbourhoods. Furthermore, the *central cell* in a rule defines the modification to the environment if the rule 'fires'.

A Stimulating Configuration is simply a **Neighbourhood** which is matched in a cell-to-cell comparison with a **Rule** in this stigmergic system. The comparison generally ignores the contents of the *central cell*, since it will normally be `EMPTY` in the neighbourhood, and filled with a brick in the rule.

3.2 The *Nest-2.11.1* Software

The *Nest-2.11.1* application is an implementation of the simple model described above. Stigmergic algorithms in two and three dimensions can be developed, within both cubic and hexagonal lattices. A significant number of auxiliary functionality has also been integrated in *Nest-2.11.1*, primarily mechanisms for generating stigmergic algorithms using genetic algorithm techniques. The extended techniques are discussed further in Section 3.4. This software has kindly been made available for the purposes of this project by Guy Theraulaz[1], Kjerstin Easton[2] and Joe Andrieu. Several screenshots of the software in use are presented below.

Figure 3.2 shows the main interface to simulations in *Nest-2.11.1*. A number of simulations can be simultaneously loaded at one time, and interfaces for modifying, simulating and evaluating architectures are provided. Figure 3.3 shows a stigmergic rule being displayed and modified. On the right a 3D version of the rule is shown, while the left shows each individual

[1] `theraula@cict.fr`
[2] `easton@caltech.edu`

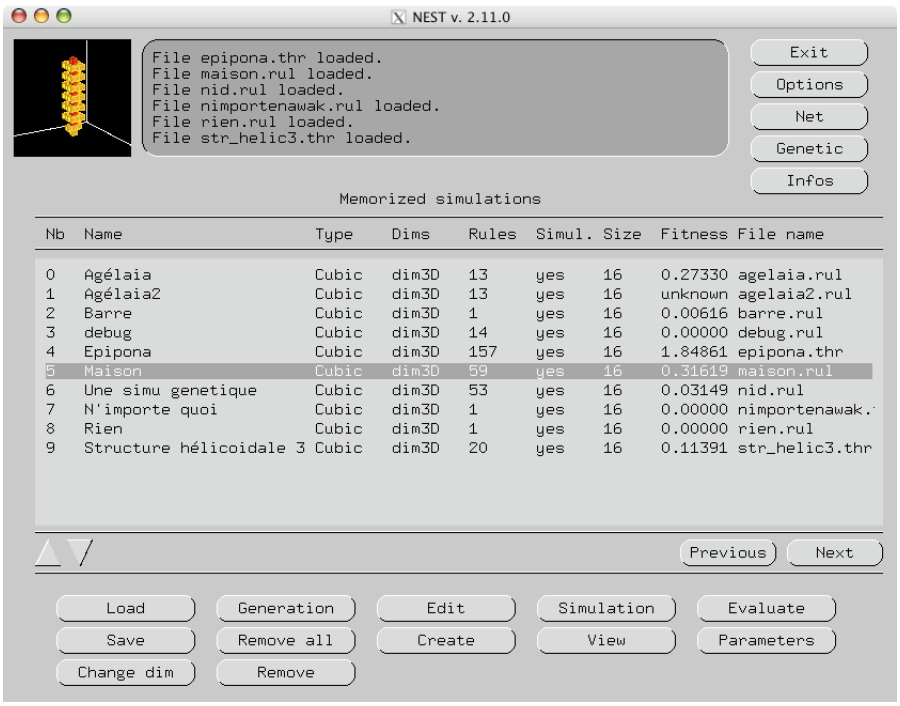


Figure 3.2: The main interface window for the *Nest-2.11.1* software.

layer. The centre cell within the middle layer shows the colour of brick which will be built if this rule fires. Finally, Figure 3.4 shows the inspection of a constructed architecture. In this figure, an invisible *cut* has been made, allowing the user to see 'inside' the structure without modifying the arrangement of bricks directly.

3.2.1 *Nest-2.11.1* Implementation

The software is written in the C programming language, requiring a Unix or Unix-variant operating system with the X11 display environment to compile and run. With some effort it can be made to run on both Mac OS X systems, and Microsoft Windows systems using either Cygwin[3] or MinGW[4] as a UNIX emulation layer between the *Nest-2.11.1* software and the underlying operating system. However, substantial experience compiling and debugging software on Unix systems is required to achieve this.

[3]Refer to www.cygwin.org for more information.
[4]More details about MinGW can be found at www.mingw.org.

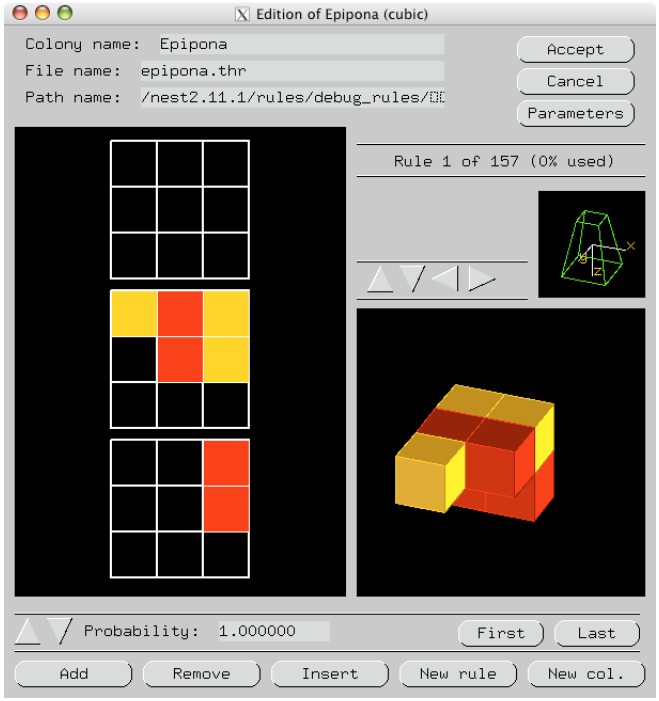


Figure 3.3: Editing rules in the *Nest-2.11.1* software.

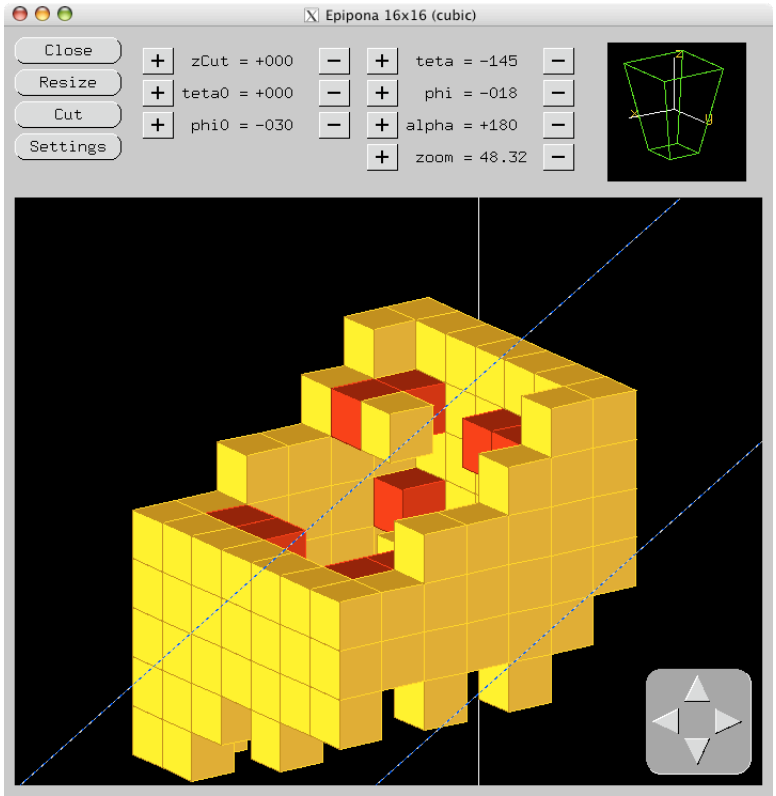


Figure 3.4: An architecture built by the *Nest-2.11.1* software.

3.3 Coordinated Algorithms and Coherent Structures

It was noted after an extensive random sampling of algorithms that the only structures which appeared to contain significant coherent structure were those where the algorithm had been designed by hand:

> "We have tested 10^6 randomly selected algorithms containing up to 40 stimulating configurations (+symmetries around the z-axis)" [156]

> "A random exploration of the space of algorithms... yields no interesting result but only random or space-filling shapes. However, it is possible to produce complex shapes, some of them strikingly similar to those observed in nature... by working backward from some shapes to be generated to their corresponding algorithms." [17], Section 6.3.2

> "Theraulaz and Bonabeau found *a posteriori* that structured shapes can only be built with special algorithms, coordinated algorithms, characterized by emergent coordination: stimulating configurations corresponding to different building states must not overlap, thereby avoiding the deorganization of the building activity." [17]

> "In order for the construction to proceed in a coherent way, there has to be a succession of a certain number of qualitatively distinct building stages... We call such a building algorithm ... a coordinated algorithm." [158]

A detailed discussion of the concept of 'coordinated algorithms' can be found in Section 5.1. For the meantime, we can see that this assertion demands the presence of distinct building stages, each entered in a controlled fashion and each corresponding to and responsible for some clearly-defined structural feature within the completed, 'coherent' architecture.

> "The difference between [coherent and incoherent] architectures is striking, though hard to formalize. One given coordinated algorithm always converges toward architectures that possess similar features. On the other hand, algorithms

> that produce unstructured patterns sometimes diverge: the same algorithm leads to different global architectures in different simulations."[17]

The informal meaning of 'coherent' here, with regards to an architecture and its construction, is as an antonym of *chaotic*. In other words, an observer would find it significantly easier to grasp regularities in the placement of bricks in a coherent architecture than in a non-coherent one.

A second property of 'coordinated algorithms' suggested above is the similarity of the architectures produced over multiple simulations; any number of architectures produced by the same coordinated algorithm will be structurally similar, whereas if the same set of simulations was performed using an uncoordinated algorithm, the resulting architectures will have little in common.

> "This tendency to diverge comes from the fact that stimulating configurations are not organized in time and space and many of them overlap, so that the architecture grows in space without any coherence."[156]

A factorial correspondence analysis[156, 15][5] is presented to show that the coordinated algorithms considered, or discovered, exist within a compact region of the space of all stigmergic algorithms. This suggests that not only are these 'coordinated algorithms' rare – so rare that not a single one was found during the random search – but there also exists a significant relationship between close algorithms and close architectures.

The notions of 'coordinated algorithm', 'coherent architecture' and the assertions made above are critically examined in Chapter 5, where we will attempt to further the original investigation into the behaviour of these abstract stigmergic systems along the directions initiated in [158, 156, 17].

3.4 Evolution of Stigmergic Algorithms

As will be described in detail in Section 5.5.1, it is clear that the size of the space of algorithms is far too large to explore systematically. To circumvent this issue, a genetic algorithm was

[5]For an English translation of [15], see http://www.micheloud.com/FXM/COR/E/

applied[22] and then revised[19] in an attempt to search the problem space automatically for new 'coordinated' algorithms.

In order to apply genetic algorithm techniques, a fitness function for evaluating the performance of individual solutions must be provided. The goal of this system is to produce coherent architectures, but as was recognised immediately by Bonabeau et al:

> "...it is in the present case hard to define fitness functions on *architectural phenotypes* (it is difficult to define formally the adaptive value of the biological plausibility of the shape of a nest: what is an "interesting" architecture?)"[22]

The subjective nature of evaluating some product as 'interesting' (or indeed 'coherent', as we will see later) is a significant problem when attempting to automatically generate such an item. Instead, the initial fitness function measures a "behavioural phenotype", derived from a simple observation of the existing coordinated algorithms:

> "In algorithms that generate coherent architectures, many micro-rules are used, whereas in algorithms that generate structureless shapes, only one rule or a few rules are actually used in the course of the simulation."[22]

Each algorithm in the population was simulated until a fixed proportion of the lattice was filled with bricks, or a limit on the number of simulation cycles to run was reached, at which point the algorithm was assigned a score by counting the number of rules which fired during simulation and dividing by 2. Additionally, a small number of biases were introduced: stimulating configurations containing large numbers of bricks are avoided, since they are unlikely to fire during any simulation[6]; each algorithm must contain a rule which matches against the single initial brick in the environment; and finally some rules that systematically induce rapid and random space-filling are to be avoided[7].

Using an initial population of 500 algorithms, a simple one-point crossover function and a mutation rate of 0.05%, 10^6 algorithms, each containing 40 rules, were evaluated. Only a

[6]Since the simulation begins in 'empty space', it is always likely that more empty cells than filled cells will surround an agent.

[7]The nature of these rules is never made clear in [22], but they may correspond to a set of rules termed *self-activating*, introduced in Section 5.5.4.

small number of 'coherent' architectures were generated, leading to the reiterated suggestion that "algorithms producing interesting shapes are very rare"[22].

3.4.1 An Improved Fitness Function

A revised fitness function was presented in [19]. While it includes the basic observation of the original function outlined above, it additionally specifies that coherent architectures are 'compact' (in that bricks tend to be face-adjacent rather than edge-adjacent with neighbours), and also that the structure features modular patterns which are large and repeat themselves.

The fitness function itself relies on the generation of a 'construction graph' which maps the activity of the rules as they place bricks during the simulation. An example of construction graph generation is shown in Figure 3.5. Each brick in the structure has a corresponding node within this graph. Each time a brick is placed, a new node is created, labelled with its order in the sequence of bricks placed and also with the *id* of the rule which placed it (represented in the figure as labels of the form `<order> / <rule id>`. Each node receives links from the nodes of other bricks which share its neighbourhood at the time it was placed.

The new fitness function is defined according to some properties of this graph, as follows:

$$F = k \times F_1^{1/3} \times F_2 \times F_3 \times F_4^{1/3} \tag{3.1}$$

where F_1 is the fraction of microrules used during the simulation, F_2 is the compacity of the architecture, measured using the ratio $\frac{\text{edges}}{\text{nodes}}$, F_3 is proportional to the mean size of the patterns detected in the graph, F_4 is an evaluation of the pattern matching scheme, and finally k is simply a constant. The derivation of each of these terms, and in particular the pattern-finding scheme, is described in detail in [19].

Crossover Methods

The crossover operation within a genetic algorithm is a naturally destructive process. However, in this situation it is desirable whereever possible to avoid breaking the dependencies between rules – the sequences of "handshakes and interlocks"[156] which are the hallmark

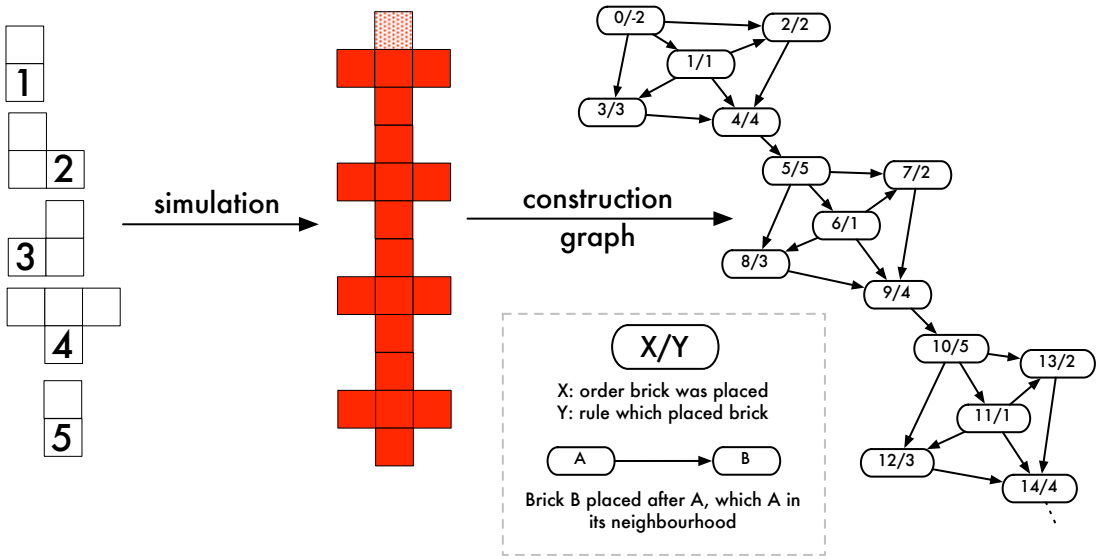


Figure 3.5: Generation of a Construction Graph. The rules on the left of the figure produce the central architecture when run in simulation. The construction graph on the right represents fully the construction process. Each node represents a brick, the internal labelling indicating the order of placement and the rule responsible. The initial brick, as a special case, is labelled with rule id -2, indicating that it was not placed by any rule within the algorithm.

of a coordinated algorithm. For this reason, a simple one-point crossover is not suitable for an effective search of the problem space. A specialised crossover method was devised, using a heuristic estimate of the dependencies between rules obtained during simulation. The fragility of algorithms is an issue that will return in Chapter 8, but space does not permit a further discussion of the specifics of this particular crossover method; for more details, again refer to [19].

3.4.2 Results

Several positive results are claimed in [19]. First, a positive correlation between the evaluation of the revised fitness function and manually-assigned observer ratings of structure were demonstrated. This is claimed as evidence that this new function more accurately captures some measure of coherent structure.

Secondly, reuse of 'sub-modules' – small collections of rules responsible for distinct features – was observed in the population. This is most likely a verification of the preservation of inter-rule dependencies by the crossover function (see above).

However, the increase in population fitness is slow, and seems to plateau quickly. The lowest fitness within the population remains close to zero, highlighting the continued fragility of these stigmergic algorithms despite the measures taken to preserve significant relationships between rules.

3.5 Critical Evaluation

Despite the apparent success of the application of a genetic algorithm in the search for interesting coordinated algorithms, there are several issues which must be examined carefully when evaluating the significance of these results.

It is difficult to separate much of this criticism from more fundamental issues regarding the original premise of 'coherent' architectures and 'coordination' in stigmergic algorithms. A more thorough critical discussion of these terms appears in Chapter 5.

3.5.1 Algorithm Length

Using a fixed length algorithm automatically in [22] inherently biases the search for 'coherent' architectures towards algorithms similar to those which have already been presented as 'coordinated' – each of the hand-constructed algorithms contains a significant number of rules, required by the complexity and modularity of the architectures these algorithms were derived by "working backward"[17] from.

By demanding that algorithms contain this many rules, and most importantly that all the rules within this algorithm are utilised, the possibility of finding smaller algorithms which might exhibit desirable properties is removed. Should such a smaller algorithm exist as a subset of these 40 rules, the other 'chaff' rules, which need never used during simulation to produce a coherent architecture, penalise the evaluated fitness. This point is actually made during the initial work, but its relevance not carried through to the latter series of genetic algorithm experiments:

> "[If] random behavioural rules are added to a coordinated algorithm, they may very well have a relatively small influence, or even no influence at all, on

> the corresponding coherent architecture obtained, because such an architecture is partially or fully constrained, so that random rules are unlikely to ever be applied."[156]

To correct this, a variable length genotype was used in the later work[19].

3.5.2 A Subjective Fitness Function

Despite the revision of the fitness function in [19], it remains a collection of heuristics which correlates with the opinions of a set of subjective observers. It was arrived at via "trial and error"[19], and while it may be an improvement over the original fitness function, it could still not claim to be a measure of the coherency of a structure.

What is exposed more explicitly by the derivation of this function is the correspondence between repeating patterns of bricks and the perceived desirability of an architecture. The factor F_2 is directly proportional to the mean size of the patterns detected within the structure. These patterns are found using a depth-limited search of the construction graph. The depth here is set at 4, implying that all interesting patterns can fit within a $5 \times 5 \times 5$ cube, and furthermore the term is then biased to "favor patterns with a lot of bricks"[19]. The size of patterns – the granularity of the modularity – which is considered desirable in this study is thus made explicitly clear. There appears to be no consideration that significant structure might be present in architectures built of very much smaller patterns (as we touch upon in Section 5.2.4), or indeed much larger ones. This issue will be the subject of further consideration in Chapter 9.

Initial Population Seeding

In [19] it is stated that "the initial population also contains a small number of algorithms that are known to produce structured patterns." By doing this, the progress of the genetic search is clearly skewed towards architectures similar to those already found and analysed 'a posteriori' by Bonabeau et al.[156]. This serves to further limit the exploration of possible algorithms to those within the small region of space where 'coordination' is known to happen, rather than fully explore the range of stigmergic algorithm behaviour.

3.5.3 Smoothness of the Problem Space

The fragility of the fitness of algorithms after even the smallest change would seem strong evidence that the mapping between close algorithms and close architectures may not be smooth, or certainly not as smooth as originally asserted in [156]. Bonabeau et al. also recognise this, despite their simultaneous yet seemly contradictory reiteration of the original assertion:

> "Two close structured architectures appear to be generated by two close coordinated algorithms ... That two close algorithms generate two close architectures could not be fully tested but is certainly not generally true: for example, removing a micro-rule from a coordinated algorithm's micro-rule table may result in a disorganization of the building process."[19]

What should be clear from the results of these further studies involving genetic algorithms is that this closeness actually stems from the 'sub-modules' encoded into 'coordinated' stigmergic algorithms: *two algorithms which contain identical or similar building stages will produce identical or similar architectures.*

For example, if an algorithm contains subsets of rules (building stages) which construct a column and a flat plane of bricks, it is very likely that, with some degree of variability, the structures produced will exhibit both of these features and thus appear close within the space of all architectures. The high fitness evaluations presented in [19] for architectures which share 'sub-modules' also support this revised assertion.

The fragility of the algorithms' fitness under the modification of individual isolated rules, rather than stages, strongly indicates that any mapping between 'close' algorithms (those which differ by only one or two rules) and the architectures they produce may be far from smooth. The modification of a *single* brick within an algorithm through crossover or mutation operations can entirely halt construction progress during simulation. This problem was clearly exhibited in [22], and somewhat reduced later [19], but an adequate solution in this particular mode of investigation remains elusive.

3.6 Related Systems

In this section, the relationship between the discrete, abstract model of stigmergy (which forms the basis of this thesis), and other simple models which have been used to investigate collective behaviour and 'emergence' will be considered.

Furthermore, it is interesting to consider how the simple model used here fits within a larger set of possible multiagent models; by understanding the choices and restrictions on this model, we can further clarify how these choices might manifest themselves as aspects of the system's behaviour.

3.6.1 Variations on Abstract Stigmergy

While the work presented in [158, 156] is the most often-cited investigation into simulated stigmergic building, other work does exist. In particular, initial research is presented in [115] which utilises *quantitative* stigmergy (i.e. the use of pheromones) for the construction of walls. Furthermore, this work attempts to define rules using multiple pheromones in order to spatially guide placement of building material.

An alternative, simple model demonstrating cell placement in comb structures is described in [95, 94, 96], along with an algorithm which can produce a wide range of the nest shapes seen in the *Polistes* wasp genus. This model uses rules which describe an agent's tendency to complete cell rows on the face of the structure.

3.6.2 Relationships to Other Abstract Emergent Systems

All simulations are necessarily abstract, since it is never possibly to completely model the reality in which emergent systems are observed. The purpose of modelling is in fact to simply capture the behaviour of a system sufficiently well using only that information which is believed to be relevant, and discarding any other variables within the system.

The model of stigmergy presented in [156], despite being presented within a strongly biological context, is similar to other discrete multiagent systems and multi-state networks[1] which generate 'structures'. Several such systems are discussed below, and illustrated in

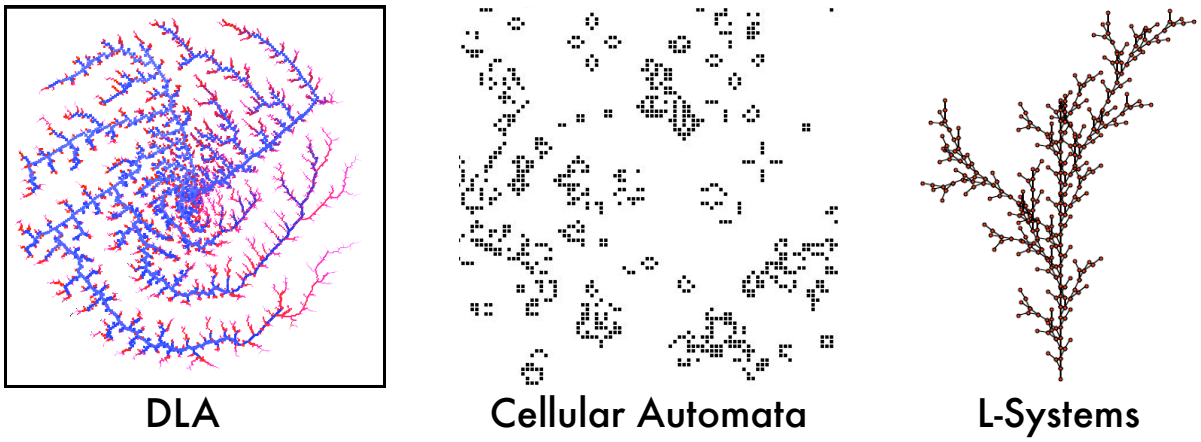


Figure 3.6: Examples of Diffuse-Limited Aggregation (DLA), Cellular Automata and L-System structures.

Figure 3.6.

Cellular Automata

Perhaps the most well-known abstract systems are Cellular Automata[**?**, 168, 109] (CAs), which despite their simplicity continue to receive significant attention[166]. CAs share the same spatial lattice as used in the *Nest* system, but differ primarily in the location of environmental change. In a CA, each cell's state is updated during every time segment of the simulation. This contrasts with the 'agents' in *Nest-2.11.1*, which limit activity to only a single location within the lattice at any time. However, if a *Nest* lattice were filled with agents, and each agent allowed to apply rules simultaneously, then the resulting system would correspond directly to a CA.

Additionally, it is also typical for a CA model to feature cells which are limited to binary states: *empty* or *filled*. While this constraint is not necessary (for instance, see [166]), the largest body of work concerning CA behaviour only considers this binary variety.

Asynchronous Cellular Automata

In contrast to 'classic' Cellular Automata, as described above, Asynchronous Cellular Automata (ACAs) *do not* update each cell at the same time, using some global clock cycle. Instead, in each time segment a single cell is selected at random and updated according to

the state of its neighbours, without any other cells having their states updated[142].

While it was never noted by Bonabeau and Theraulaz[158, 156], the *Nest* model described here can be considered a constrained version of this system, since they share many characteristics:

- A regular lattice arrangement of cells, each of which may take a number of states, frequently corresponding to 'filled' and 'unfilled', or 'alive' and 'dead';
- A single 'active' cell, chosen randomly at each time step;
- The state of the updating cell is modified based only on the states of its neighbouring cells.

However, the *Nest* model features a further set of constraints:

- The cell selected to be 'active' must be an *empty* cell, and
- The update rules for the ACA must only describe cell transitions from *empty* to *filled*

These constraints reflect the notion that an agent can only exist in empty space, not within the bricks forming the architecture, and also that once a brick has been placed, it cannot be removed.

Despite the apparent similarity between the *Nest* model and ACAs, there does not appear to be any direct transferable knowledge which might assist in the understanding of how a *Nest* stigmergic algorithm behaves. Most work surrounding ACAs is focussed either on modelling very simple patterns from a mathematical perspective[152], exploring the computational possibilities[166], or reproducing the self-replicating structures examined in studies of artificial life[123].

An excellent, recent survey covering the many varieties of cellular automata has been produced by Ganguly et al.[70], and the interested reader is directed there for further details.

L-Systems

L-systems are typically used to model patterns found in nature, such as the striking regularities seen in plants[133] and the patterns formed on shells[119]. L-system rules describe a

grammar of development, representing the transformation or growth of structural elements from a single initial 'axiom'. It is possible to produce structures and patterns which are strikingly similar to those seen in nature, such as the fern pattern shown in Figure 3.6.

The rule used to produce this structure is `F[+F+F]F[-FF]F`. Each iteration, every instance of `F` within the structure 'string' is replaced with the string defined in this rule. When the string is displayed graphically, substrings in square brackets are rendered as branches, with `+` indicating the branch should be drawn to the left of the trunk, and `-` branches drawn to the right.

L-systems do not generally operate within a lattice representation of space. Instead, simple grammars are often augmented with angle parameters which indicate the relative positioning of branches to be added. In the L-system shown in Figure 3.6, the angle used is $27°$.

Since L-systems use a grammar which involves symbol replacement, the behaviour of an L-system cannot be directly modelled by a *Nest* system. Within a *Nest* simulation, material can only be placed, and not removed, so no replacement is possible.

Diffuse-Limited Aggregation

It is noted in [22, 17] that the *Nest* model is a generalisation of Diffuse-Limited Aggregation[154] (DLA), along with other growth models (such as the L-Systems described above). In DLA, structures are grown within a volume of space, where randomly-moving bricks attach to any structure they encounter, in a crystalline-growth manner.

It is also noted in [17] that a large number of rules (63×2^{20} and 240 rules for 3D and 2D cubic geometries respectively) are required to produce a DLA simulation within the more-general *Nest* model. It is also noted that significantly smaller rule sets are possible if the model is extended to allow rules to contain 'don't-care' cells – that is, cells which match within the rule regardless of the corresponding bricks in the neighbourhood in which this rule is firing. With this addition, only 6 rules are required for 3D DLA simulation, and only 4 in two dimensions. This suggests that other models might be able to more-compactly express the growth and assembly of certain structures than the *Nest* model; the stigmergic

construction model presented here does not *elegantly* encapsulate the behaviour of all simple systems.

3.7 *Nest-2.11.1* – Summary

The original purpose of the work presented in [158, 156] was simply to produce a model which was capable of reproducing some of the building behaviours seen in social insects, and certainly in this respect it was been a definite success. Despite the simplicity of the model, unarguably life-like structures can be produced, and the dynamics of the building process has been investigated.

However, beyond this it seems little can be said concretely. The space of possible behaviours for even this limited, discrete system is vast, and it appears that only a tiny proportion of the resulting constructions are recognisable as complex or meaningful. Faced with this, algorithms to satisfy the original goals were constructed by hand, and the strong assertions surrounding the terms *coherent architecture* and *coordinated algorithm* were based on the properties of these manually-generated systems, as compared to those explored by a limited random sampling.

By more systematically exploring the space of algorithms through the use of genetic algorithm techniques[22, 19, 17], the assertions made in [158, 156] have been more thoroughly tested, and to some extent retracted. In this section it has been shown that the use of a genetic algorithm exposes important properties of simple stigmergic algorithms. They are fragile and even the smallest change can significantly affect the behaviour of the simulation and the structure produced.

It is no longer so clear that the mapping between algorithms and architectures is so smooth, and the fundamental assertion that 'coordination' is required for 'coherence' can no longer be based so strongly on the original analysis. The concepts of 'coordination' and structural 'coherence' must themselves be subjected to further scrutiny, and this effort is continued in Chapter 5.

In the interim, while the software used for this investigations was available for use for the remainder of the research presented here, it had reached a point in its own evolution

where the restrictions implicit in its implementation placed unacceptable limitations on the direction of any future research into this particular family of systems.

To set future investigation free from any artificial implementational restrictions, a new software system was developed – *Nest-3.0* – capable of supporting the original model yet allowing freedom for expansion and modification with relative ease. This software is described in Chapter 4.

Chapter 4

The *Nest-3.0* System

Overview

In this section a new system for stigmergic simulation is presented, which improves upon the *Nest-2.11.1* system used in the original work discussed in the previous chapter. This new software, entitled *Nest-3.0*, improves upon the previous version by including: a full multiagent implementation; more flexible rule construction; and comprehensive support for rule rotations. *Nest-3.0* introduces the ability to remove bricks from the structure in addition to increasing the number of available discrete brick types, resulting in a general model for stigmergic architectural modification.

The implementation choices made during the development of *Nest-3.0* are discussed, and an extensible system framework is presented. Each distinct software module is then detailed along with the objects used to model the simulation state. The details of two lattice geometries are described and their implementations and differences discussed. The Simulation core and enhancements to the agent model are then detailed. Finally, performance considerations and optimisations are outlined and areas for future extension and investigation highlighted.

4.1 *Nest-3.0* vs. *Nest-2.11.1*

The original nest-building simulation system (*Nest-2.11.1*) and its source code was kindly made available for this project, but while the concepts which underlie lattice swarms are reas-

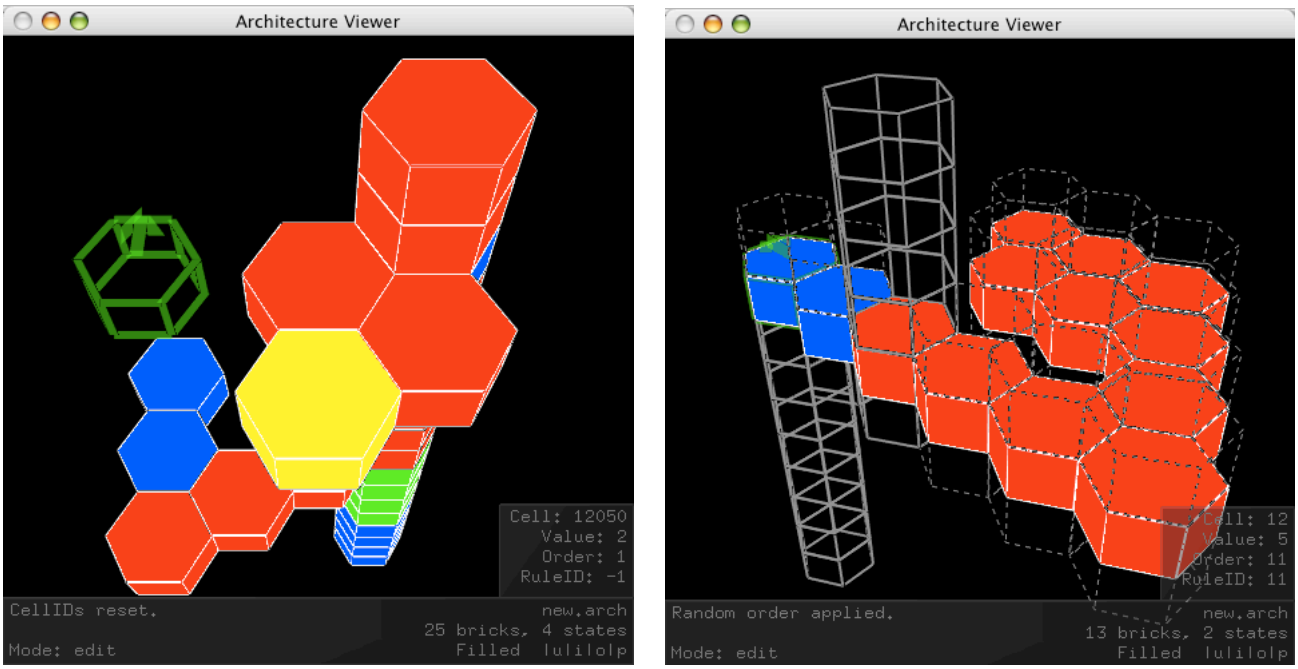


Figure 4.1: An example of an interface to the *Nest-3.0* system.

onably abstract, the given implementation was very restrictive and would tightly constrain an further investigation into the properties of such systems.

A new lattice swarm system – *Nest-3.0* – was implemented which gives broader scope of experimentation with greater control over the parameters and constraints involved. The features of the new system are outlined in Table 4.1 below, with reference to the original system where appropriate.

4.2 Abstract Stigmergic Improvements

Whilst algorithms have been developed by Bonabeau & Theraulaz using the *Nest-2.11.1* software which are very successful in producing 'biological' structures, there are aspects of the simulation system which could be significantly improved.

4.2.1 Brick Geometry

The structures in *Nest-3.0* exist within a discrete version of space, split into distinct cells. However, the design of the software makes no assumptions about the nature of this space, allowing new geometric arrangements of cells to be used during experimentation. Currently

Feature	***Nest-2.11.1***	***Nest-3.0***
Implementation Languages	C++, X11	Ruby, C++, OpenGL
Platform	Unix	Unix, Windows, Mac OS X
Extensibility	Requires recompilation of source code & detailed knowledge of the internal system structure	Based on very-high-level scripting language (Ruby), simple to extend
Architecture Geometry	Cubic, hexagonal; hard-coded into the system	Capable of supporting many architecture types
Agent Behaviour	Only one agent; movement is random	Many agents; different (i.e. non-random) movement behaviours possible; agents may also carry internal state if desired
Brick types	Limited to 4 colours	Virtually unlimited number of colours
Rules	Fixed size, with rotations around the z-axis only. Rules must match local environment exactly	Unlimited size, with rotations around x-, y-, and z-axes possible; can also match against dont care cells, or cells which must be from a specific range of colours
Building behaviour	Agents place bricks in empty cells in current position only	Agents can place bricks in any empty space around them; removal of bricks also possible (excavation).

Table 4.1: Comparison between *Nest-2.11.1* and *Nest-3.0*

Feature	**Cubic**	**Hexagonal**
2D Neighbourhood Size	9 cells	7 cells
3D Neighbourhood Size	27 cells	21 cells
Rotations	24	12

Table 4.2: Geometric differences for cubic and hexagonal architectures. The determination of rotation is explained fully in Sections 4.5.2 and 4.6.1.

two different cell geometries are available – cubic and hexagonal. Each geometry has naturally different characteristics, notably the number of cells in a local neighbourhood and differing axes of symmetry. Details of the differences between cubic and hexagonal geometries are given in Table 4.2.

It was noted when describing the original simulations [156, 158, 19] that structures built from hexagonal cells were aesthetically 'more biological' than those consisting of cubic bricks:

> "We see that [hexagonal] geometry allows round shapes to be easily created, which is not the case for cubic bricks." [156]

Given that such simulations were directed towards producing nest-like structures which naturally consist of hexagonal cells this seems hardly surprising. However it is important to note from this the significance of the fundamental geometries upon which a structure is based, and the impact of these geometries upon not only what structures are possible, but which structures are algorithmically *simple*.

For example, a relatively smooth cylinder can be produced with far fewer rules when using a hexagonal geometry than when working within a cubic geometry. Similarly, it is much simpler to generate structures with perpendicular features within a cubic simulation, as a direct result of the angles at which face-adjacent bricks can be placed. As can be seen in Table 4.2, both the neighbourhood sizes and number of possible rule rotations is greater within a cubic geometry than a hexagonal geometry. This in turn increases the number of possible unique local neighbourhoods and this difference in the size of the state space can have a significant impact on systematic considerations of stigermgic systems within this framework. This is explored further in Chapter 5.

The original *Nest-2.11.1* system allowed the user to choose between either cubic or hexagonal cell geometries. *Nest-3.0* separates the geometry of the underlying system from the simulation code, making it simple to provide other types of brick shape for experimentation, or even a closer approximation to cell placement within a continuous spatial environment.

4.2.2 Perceiving the Local Environment

The perceptual range of an agent within a *Nest-2.11.1* simulation is limited within the simulation to only those cells which are face- or edge-adjacent to the agent's current location. For cubic simulations this equates to a $3 \times 3 \times 3$ cube around the agent's location. For hexagonal simulations it is represented by the 6 surrounding cells, and the 7 cells directly above and below it. While this seems to follow a literal interpretation of how a stigmergic system should behave, with only immediately local simulii influencing the agent, it may be *too* simple, disallowing many interesting sense-configurations.

In *Nest-2.11.1*, agents are represented as 0-dimensional points with no associated in-

formation: effectively they are not embodied within the space they 'exist' within. This is convenient because it allows the system designer to give this non-entity abilities which are easier to write in software than produce in the real world. An example of this is the ability to sense perfectly in all directions around itself, including above and below.

In reality, agents are only able to sense within a limited field depending on their current orientation. *Nest-3.0* allows designers to produce this effect by allowing them to control the size and region of bricks which the agent can sense in relation to its current position. This can be used to produce agents which are capable of a wide variety of sensory abilities, such as only being able to sense directly in front and above their current location for any given distance.

4.2.3 Rotation

When matching a rule against an agent's local environment, often it is useful to allow the rule to be matched in varying rotations around some central axis. Allowing rotations of rules often produces a simpler rule set, when used appropriately. This can be seen very clearly in Figure 4.2. *Nest-2.11.1* allowed rules to be rotated around the central vertical axis, giving 4 rotations for each cubic rule, and 6 for each hexagonal rule. When the rules are compared to the local environment, if any rotated version of the matches then the (rotated) rule is applied in the position that it matched.

Allowing rotated rules to match during a simulation effectively removes an agent's ability to discriminate between different global directions. *Nest-2.11.1* allowed rotations around the Z-axis only. *Nest-3.0* extends this to also allow rotations through the X- and Y-axes. Furthermore, specific sets of rotations can be used, giving the user complete control over which types of global directional fields the agents can sense. The most obvious uses of this feature are enabling the following:

Internal Compass By disabling all rotations around the Z-axis, an agent has the ability to discriminate its rotation with respect to the 'up-down' axis, effectively providing it with an internal compass[88, 73, 162].

Gravity By disabling rotations around the X- and Y-axes, the agent is given the ability to

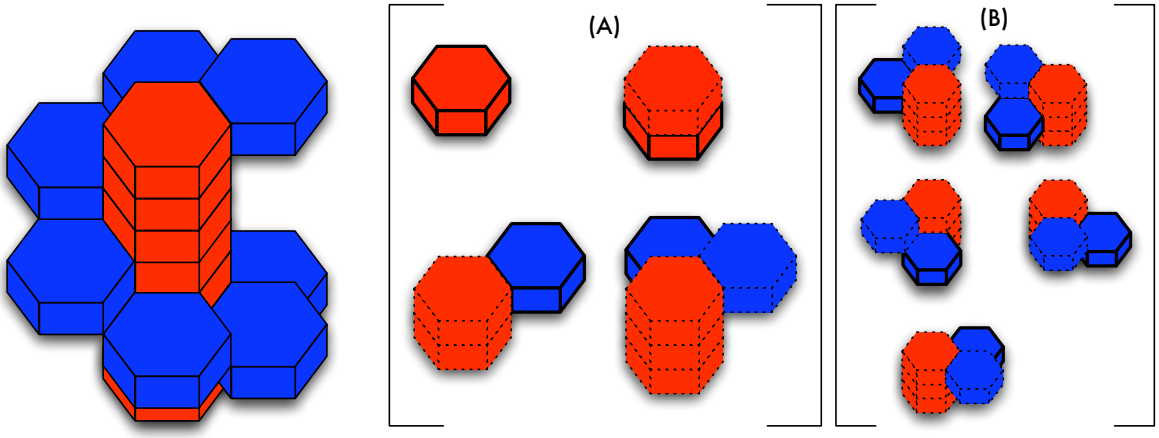


Figure 4.2: To create the structure above, all rules above must be present within the agent's rule set. However, if rotated versions of rules are allowed to match local environment configurations, only the rules in A are required, at the very beginning of the simulation. Each rule in B is a rotated version of the final rule in A. Using rotated rules allows a helix around a central column to be built using effectively only 4 rules, rather than 10.

sense its rotation with respect to the 'ground'[1], or in other words a sense of what is 'up' and what is 'down': gravity.

4.2.4 Agent Behaviour

An important property of any agent-based system is the relationship between the autonomous entity and the environment in which it is placed, and more specifically how the agent *moves*, how the agent senses and most importantly how the agent *modifies* that environment.

Nest-2.11.1 operates by selecting a single empty location, at random, and matching rules against that neighbourhood during each simulation cycle. The agent in this simulation has been reduced to a single zero-dimensional point of activity within the lattice. If actually considered as an agent, its movement behaviour is best described as 'warping' instantaneously from one part of space to another, without consideration of the constraints of actual physical travel. This single agent warping is adequate for investigating the behaviour of very simple stigmergic systems, and in fact sufficient for modelling a strictly sematectonic (see Section 2.3.4) stigmergic system. However, this implementation of agents does not allow designers

[1]There is no actual surface within the simulation which could be considered either literally or analogous to the ground in reality. However, in specific simulations a 'ground' could be emulated with suitable choices for valid rotations along with the presence of a pre-generated surface of bricks.

to effectively explore other important aspects of stigmergic systems within a more realistic *multiagent* setting.

In the new system the user has the option of using any number of agents, each with a unique, individual position within the building lattice during the simulation, more accurately implementing the notion of a 'swarm'[17]. This location information is the simplest example of individual agent state, and experimenters are free to extend the system as they see fit. One possible extension would be storing an agent's orientation, which could be implemented by dynamically determining which rule rotations to consider based on this internal state. In general, the agents now have the opportunity to modify their behaviour based on individual internal and/or external persistent state.

A secondary benefit of maintaining distinct agent instances is the possibility of allowing agents to detect not only bricks but also other agents within their local environment, and modify their behaviour accordingly[76]. This could be as simple as a preference to move towards areas of a particular agent density. A more sophisticated enhancement of the agents might allow individuals to detect aspects of other agents' internal states. Such enhancements are beyond the scope of this investigation, but they are nevertheless relatively trivial extensions to the current agent model.

Agent Movement

Furthermore, the movement of the agents can be constrained in a variety of ways. For example, agents can be restricted in their movement to unoccupied cells adjacent to the current cell, rather than mysteriously reappearing at the other side of the architecture. Another interesting movement strategy would constrain agents to positions where they are adjacent to a previously-placed brick – on the 'surface' of the architecture.

Controlling the areas around the architecture which are available for construction to take place is a potentially very important aspect of simulations where external factors, such as gravity or the position of the sun, strongly influence the construction behaviour, or where construction is not entirely deterministic. In these situations, movement constraints can be used along with internal states to influence the building behaviour and produce a more

realistic simulation.

Any positioning constraint may be realised within *Nest-3.0* simulations. It is hoped that the ability to control this, along with the possibility of internal state within agents, will enable the investigation of much richer swarm systems without increasing the complication of the individual agents or the fundamental mechanisms under which the system operates.

Pheromones

Another method of behaviour control seen frequently in natural insect colonies and related artificial swarm systems is the use of deposited *pheromones* to guide agent movement[7, 17, 29]. Ant Colony Optimisation (ACO)[50] exploits positive- and negative-feedback using deposits of virtual pheromones to search for solutions to problems which can be expressed as path optimisation problems. In general, pheromones guide the movement of agents, attracting them towards stronger sources: positive feedback is created by allowing agents to reinforce the strength of the pheromone along that path, while the slow decay of pheromones into the environment produces negative feedback and reduces the likely that poor solution paths will be selected.

Within the context of the *Nest* system, pheromones could be placed in the environment in conjunction with the placement of bricks of certain colours. Pheromones on recently-placed bricks could be used to encourage further building at that site, mostly likely resulting in structures with elongated features. Alternatively, the presence of pheromones could discourage building behaviour at those sites, thus prompting agents to build uniformly around the architecture and produce a more even, symmetrical structure.

While *Nest-3.0* does not directly provide pheromone-based agent control, the flexible architecture can easily accommodate this as a new feature. Internally, pheromones could be easily implemented as a simple, floating-point value within each cell of the lattice for each type of pheromone possible within the system. This contrasts with the use of discrete values for cell values (`EMPTY` and the various brick *colours*). The simulation system itself should then propagate the values to surrounding cells.

4.2.5 Architecture Modification - Building and Excavation

While the original system allowed agents to only place bricks in empty locations, *Nest-3.0* features the ability to remove bricks as well as place them in the structure. This allows the investigation of excavated structures as well as those built in free space. Furthermore, it allows *Nest-3.0* to implement a generalised structural modification model, in which rules simply define arbitrary modifications of any type – placing bricks, removing bricks, or even modifying bricks in place – to local environments they match against. When a rule matches the environment around an agent, that location within the lattice is simply changed according to the state of the cell designated within the rule as the *build cell*.

The ability to remove bricks as well as place them dramatically increases the space of possible rule sets which can be realised within the lattice swarm system, and also enables the implementation of behaviours which require modification to existing structures, such as the design of error correction mechanisms. This could also be used in the construction and dismantling of temporary structures which guide the further construction of the architecture. It may even be possible to develop non-deterministic algorithms which used constructive and destructive rules balanced in an equilibrium which ensured a certain overall form is present whilst the construction of the structure on a finer level is constantly in flux.

Just as the sensing envelope can be modified, the cell in which the local environment is modified after a rule is selected to fire can also be relocated. *Nest-2.11.1* required that the agent place the brick in the same location as the agent, but when excavation is allowed, this seems counter-intuitive: for an agent to remove a brick, it must be *within* that brick, and if an agent can be within a brick it would seem to be able to move about within solid matter. This is solved by allowing the agent to modify a cell within the local neighbourhood other than the one it currently occupies. By allowing this flexibility, a 'sensible' model of excavation can be produced within a lattice swarm system.

These features extend the constrained version of environmental sensing and modification present in the original system into a more comprehensive stigmergic environmental modification model, where there is no qualitative difference between 'building and 'excavating, and algorithms can be defined which feature both behaviours.

4.2.6 Rule Matching

Fundamentally both the *Nest-2.11.1* and *Nest-3.0* systems are simple rule-matching engines. In developing *Nest-3.0*, the format of the rules and the way in which they can be matched has been greatly extended.

Just as the region of the environment which the agent can sense is now free to be any shape, rules can also be of dimensions other than the immediate local environment around the agent's position.

Brick States

It is now possible to ignore the state of a cell, or to constrain the rule to match only when the cell in that region is a member of a specified set of states (colours). This gives an agent the ability to ignore parts of the local environment when necessary, and can help produce simpler and more flexible rule sets. The number of different states a cell can be in has also been increased from *Nest-2.11.1*'s 4 to a virtually unlimited number (2^{32} states)

Probabilistic Matching

A final rule-matching feature which increases flexibility is the probabilistic matching of rules to local configurations. Each cell within the rule (be it a brick or empty space) is given a real number between 0 and 1, and when the cell is compared to the corresponding region in the local environment, a random number between 0 and 1 is generated and compared to this probability factor.

If the random number generated is below the probability factor, then the cell in the local environment must match the constraints for that cell if this rule is to be allowed to fire. However, if the generated value is above the probability factor, the cell is ignored and treated as if it was successfully matched. In this way, higher probability factors can be assigned to cells where it is more important that an accurate cell match is made during the rule matching process,

4.3 Implementation Overview

In this section the specific details of the implementation of *Nest-3.0* are discussed, including the languages and tools used and an overview of the major components of the system.

4.3.1 Programming Languages and Libraries

Nest-3.0 has been developed using the high-level object-oriented scripting language Ruby[2], along with some C/C++ and OpenGL[3]. Each of these tools brings important features to the system as a whole.

All the software libraries and components are multi-platform and open-sourced, allowing the simulation to run under any operating system with a C++ compiler and an available OpenGL library.

Datastructure Manipulation in C++

The underlying datastructures have been implemented as C++ objects. C++ compilers are available for almost every hardware platform in existence, so this represents a solid basis for the core of a multi-platform application. A second but equally important benefit is the speed at which compiled C code runs. All of the functions which compare cell contents and local neighbourhoods are written at this level.

C++ represents the most effective choice for producing compiled and therefore fast code, whilst still maintaining platform independence. If speed is critical, the next step would be to implement functions using Assembly Language code, but at such an extreme low level instructions become processor-specific and platform independence is easily lost.

This C++ code can be compiled using most common compiler systems, such as the GNU[4] `gcc` compiler and `make` utility. It contains no references to platform-specific header files, and since no filesystem code is present there are no platform dependencies in the entire low-level module. The SWIG[5] (Simplified Wrapper and Interface Generator) automatically creates

[2]http://www.ruby-lang.org
[3]http://www.opengl.org
[4]http://www.gnu.org
[5]http://www.swig.org

an additional C++ file `Nest_wrap.cpp`, providing wrapper functions allowing the C++ data and methods to be called from the Ruby scripting language. The `makefile` required to compile the codebase is generated using a Ruby tool named `extconf`, which automatically ensures that paths to the local Ruby libraries are inserted correctly within the compilation process.

High-Level Application Logic in Ruby

While C++ produces optimised and fast code, the tasks a programmer must engage in to develop complex applications are often laborious and lead to the introduction of numerous bugs. For example, manual management of memory using pointers is a powerful tool, but as such it is very easy to "shoot yourself in the foot"[6] with it.

Instead, higher-level languages have recently become very popular by providing easy access to commonly used data structures such as lists and hashes, and powerful programming constructs such as introspection and dynamic binding. These languages have been around since the birth of the modern Unix system in the form of shell programming languages such as `sh`, `bash` and `csh`, and these were traditionally used to produce *scripts* to ease system administration by automating the execution of sequences of common commands. These days popular scripting languages include `perl`[7], `Python`[8] and recently, Ruby.

These languages are typically simple to read and can be used to produce solutions to problems in less time than developing using lower-level languages like C++. They are also typically *interpreted* rather than compiled. Because of this, they traditionally run at speeds which are orders of magnitude less than the equivalent compiled code. However, as computer hardware grows faster, this difference in speed is increasingly less significant, and the difference in development time due to the sophisticated programmatical constructs which are available outweighs any performance loss. The removal of the compilation step also makes it trivial to add new functionality at any point during the program's lifespan.

[6] See http://burks.brighton.ac.uk/burks/language/shoot.htm for some anecdotal-yet-accurate comparisons of common programming languages and how they compare in terms of "shooting yourself in the foot" – simple errors with disastrous consequences.

[7] http://www.perl.org

[8] http://www.python.org

Ruby was selected for this project over the other scripting languages available because of its consistent object model (compared with Python and Perl), dynamic typing and powerful built-in datastructures. These are important features which form the basis of the flexibility to plug in different geometry implementations without requiring changes to the application logic, simulation system and other high-level system components. Ruby can load and use extensions written in C++ and existing C++ classes can be modified and extended within the scripting language. Ruby also has a very shallow learning curve, and most programmers can become fluent within a few hours of first coming into contact with it.

The C++ objects have been augmented with Ruby code to perform high-level operations such as loading and saving files, generate brick orderings and extracting rule structures from the architecture itself. The algorithm extraction implementation and other architecture manipulation code is written entirely in Ruby. The user interface logic is also written in Ruby, making calls to the underlying OpenGL extension bindings available on each platform.

Data Visualisation in OpenGL

Because the structures being manipulated within the *Nest-3.0* system are three-dimensional, it is appropriate that they be displayed in such a manner. OpenGL is an industry-standard library for displaying 3D scenes, and implementations exist for all common software platforms. The actual OpenGL library is implemented in C but similarly to the custom datastructure extension wrapper produced by SWIG above, wrappers exist which allow OpenGL commands to be executed from within a Ruby script.

The GLUT (OpenGL Utility Toolkit), which assists in implementing most common 3D rendering tasks, along with providing support for application menus and user interaction, is also used in the same manner.

The Ruby user interface implementation is separated into two loosely-coupled modules: the `GLArchitectureDisplayer` and the `GLUTViewer` modules. The `GLArchitectureDisplayer` consists of raw OpenGL function calls required to draw an architecture, cell information and basic editing controls such as the cursor (see Figure 4.3), but strictly contains only the essential code to produce an OpenGL display as a *Nest-3.0* structure. The `GLUTViewer`

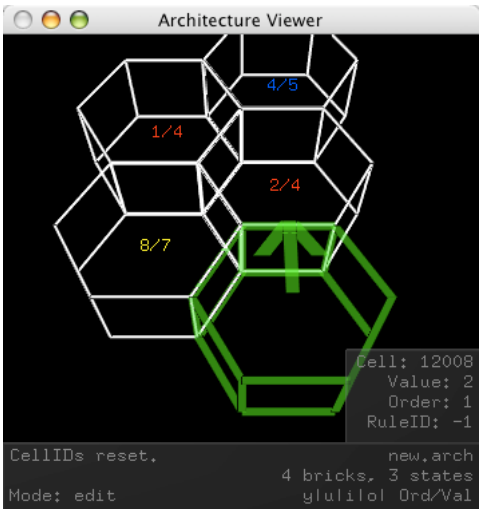


Figure 4.3: An alternative view of a structure, using the `GLUTViewer` display system. The cursor is shown as a highlighted green wireframe cell, with an arrow indicating which direction is 'NORTH'. The panel in the lower-right of the screen shows the information for the selected `Cell`. In this screenshot, `Cell`s are displayed using the ordering and encoded values, rather than a solid colour.

module implements the interface 'sugar', such as overlaid windows to display information about selected cells, and mechanisms to allow the user to interact with the structure using the keyboard. This strict separation means that the `GLArchictureDisplayer` can quickly be inserted into any other user interface toolkit capable of rendering an OpenGL scene (such as Fox[9]), or even translated back into C code.

4.3.2 System Architecture

Internally the *Nest-3.0* simulation system is split into 5 software modules, as shown in Figure 4.4, and notionally into 3 layers[10]. The Model layer is responsible for maintaining the underlying datastructure. It contains the hexagonal and cubic lattice implementations, primarily written in C++.

The Controller layer holds the logic and algorithms required to manipulate the data from a user interface, and also provides the interface with information regarding the data. For maximal flexibility – especially during algorithm prototyping – the modules within this layer

[9] http://www.fox-toolkit.org

[10] The *M–V–C* pattern of design employed here is commonly advocated as a powerful and important programming methodology, emphasising flexibility, extensibility and future maintainability of software code. For more information about object-oriented software design, refer to http://ootips.org/mvc-pattern.html

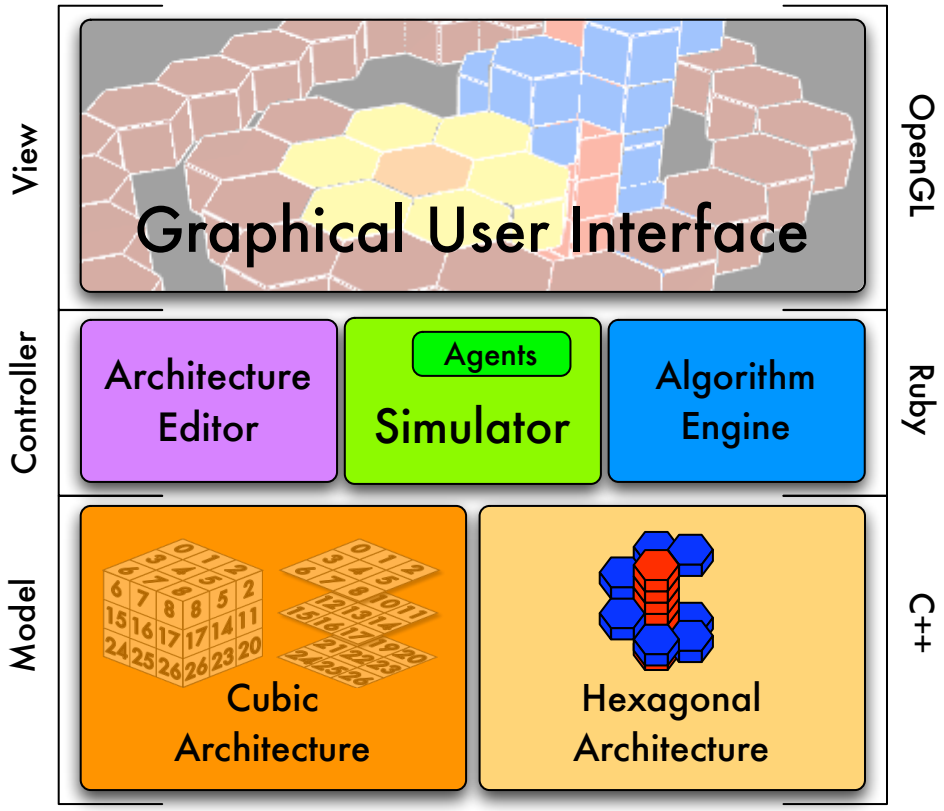


Figure 4.4: An overview of the composition of software modules within the *Nest-3.0* system.

are written entirely in Ruby.

Finally the View layer provides the user with an intuitive representation of the data along with feedback as the data is modified by either the user or internal processes. Ruby bindings to system-level OpenGL libraries are used to implement this module.

Architectures

The **Architecture** modules contain the code specific to maintaining architectural data, including maintaining cell relationships, extracting local neighbourhoods from the architecture and matching local environments to rules, saving and loading architectural and rule data and many other functions which are required to model the architecture.

One of the major problems within the original system was trying to support cubic and hexagonal architectures in both 2 and 3 dimensions by one single software core. This is where use of a better object-oriented implementation, along with some of the other dynamic

code capabilities of scripting languages, allow greater flexibility than previously available in *Nest-2.11.1*.

Within these modules, specific implementations for cubic and hexagonal architectures are present, each providing a common interface to the rest of the system such that new architecture types can be implemented and used in experimentation without requiring modification of the rest of the software.

Simulation

The **Simulation** module contains general, high-level code for running simulations using the various different types of architecture available. The implementation for running and maintaining simulations, agents and agent states is found here. The implementation of this module is described in Section 4.7.

Algorithm Engine

This software module contains code for computationally manipulating architectures and rules, the results of which can be displayed graphically to the user. The functions provided by this module are described in detail in Chapter 5, and will not be discussed further here.

Architecture Editor

The module acts as a controller system between the Graphical User Interface and the underlying data, and enables the user to directly modify structures using a three-dimensional cursor and commands to manipulate the view upon the structure. This component ensures that only valid data (brick values, orderings and so on) are used, and provides feedback about the architecture, rules and cells.

Graphical User Interface

The **GUI** provides the user with a friendly means of interacting with both the architectural data and the simulation parameters of the system. Architectures which are being both edited and produced via simulation are displayed in 3 dimensions using the standard OpenGL

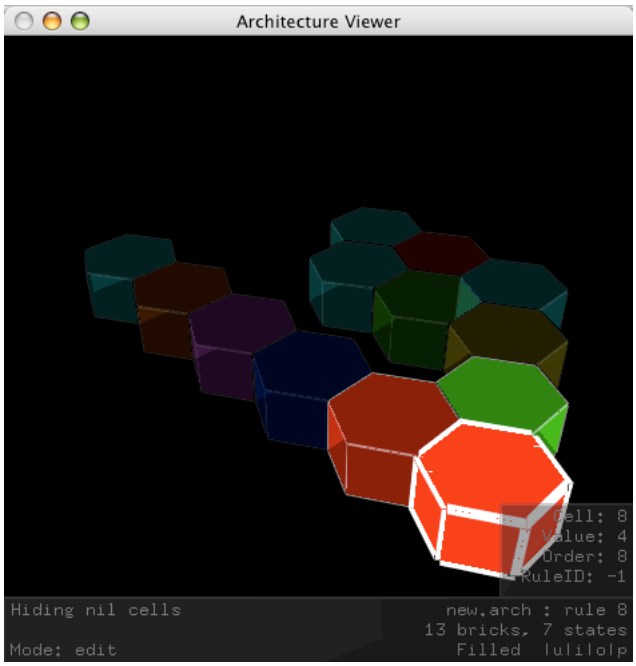


Figure 4.5: A simple architecture, first seen in Figure 4.1, after processing by the Algorithm Engine. The viewer has been switched to rule display mode, in which the build cell is highlighted, neighbourhood cells within that rule are shown dimmed, and unrelated cells are transparent.

library. As demonstrated in Figures 4.1 and 4.5, both architectures and rules can be displayed, rotated and modified using this interface, and the results of simulations displayed either in realtime or once the simulation has completed either a given number of cycles or the architecture contains some number of bricks.

4.4 Implementation Details

4.4.1 System Objects

The simulation system is described within the implementation code as a structure of related objects, whose functionality can be extended and modified as required. Wherever possible functionality is reused to avoid duplication and ease modification of the system. The fundamental objects currently in use are:

`Architectures` store details of an arrangement of `Cells`, with methods to rotate architectures, extract local neighbourhoods from around `Positions` and match against `Rules`.

`Rules` A subclass of `Architecture`, basically just a small arrangement of cells which is matched against a neighbourhood within the structure being built. It also contains probability information and statistics. The inherited match function may be extended to include per-cell probabilistic matches.

`Cells` stores all information about a particular location within the lattice, such as the type and colour of any brick present (or allowable in the case of `Rules`).

`Agents` represent each individual agent with the system, storing its position and any other individual state information (flags, individual rules, etc).

`Positions` describe a location within an architecture. One of the largest problems with the original simulation was trying to develop some universal way of referencing cell positions between both cubic and hexagonal lattices. This class is sub-classed by all `Architecture` implementations in order to provide a single interface to positioning.

4.4.2 Cells, Bricks and States

The abstract model of space used here divides the environment into discrete cells, modelled internally using `Cell` objects. Each `Cell` has associated with it a state. Most often this state is used to determine if a brick has been placed in this cell or not. It should be noted that hereafter a *brick* is simply a `Cell` object which is not empty, or in other words, a `Cell` object whose state is not `EMPTY`, but instead `RED`, `BLUE` or in general any other 'colour'.

Cell Matching

The state of each single `Cell` is encoded into a fixed-length bit representation, with a single bit representing the presence of one particular state. As shown in Table 4.3, the bit sequence for a single cell can encode a number of distinct states, and this sequence of bits can be easily transformed into a simple integer number. Because a single bit within the array encodes to a single state, we can use multiple set bits to encode a cell within a rule which can be in any number of states (see Table 4.1). This is achieved by matching the value of this `Cell` against the target `Cell` using a binary `OR` function rather than a straight comparison.

State	EMPTY	RED	BLUE	YELLOW	GREEN	ORANGE		Value
Bit	*0*	*1*	*2*	*4*	*8*	*16*		
Value	0	1	0	0	1	0		= 9

Table 4.3: Multi-state matching using Bit Arrays

State	EMPTY	RED	BLUE	YELLOW	GREEN	ORANGE	Value
Bit	*0*	*1*	*2*	*4*	*8*	*16*	
Rule Cell	0	1	1	0	1	0	11
Target Cell	0	0	0	0	1	0	8
Result of OR	0	0	0	0	1	0	8

Table 4.4: An example of bit-setting to enable bricks to match against multiple values.

A match is determined using the following boolean function:

$$match(cell_{rule}, cell_{target}) = (cell_{rule} \mid cell_{target}) = cell_{target} \tag{4.1}$$

For example, a rule cell configured to match against `RED`, `BLUE` or `GREEN` target cells is shown in Figure 4.4.

4.5 Cubic Geometry

The cubic geometry implementation featured in *Nest-3.0* is implemented using a simple array structure, with cells referenced absolutely using a simple (x, y, z) coordinate system. The index of the cell (x, y, z) within the array is given by the function

$$index(x, y, z) = x + (y \times XSIZE) + (z \times XSIZE \times YSIZE) \tag{4.2}$$

where $XSIZE$ is the total size of the architecture along the x-axis, and $YSIZE$ is the length along the y-axis. Indexing of a $3 \times 3 \times 3$ cube is shown in Figure 4.6.

4.5.1 Matching Neighbourhoods using Bit Arrays

In order to match rules against neighbourhoods within the architecture quickly and efficiently, the cubic implementation maintains two representations of the structure within an

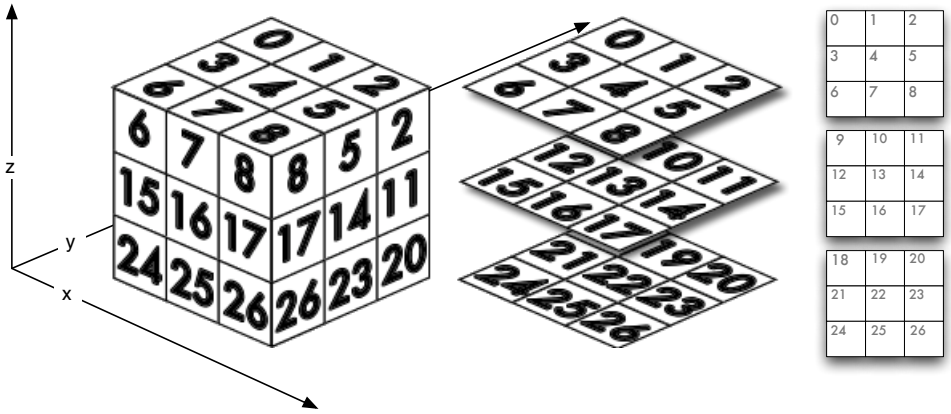


Figure 4.6: Indexing of a three-dimensional cubic structure. The flat, planar layout on the right is used in following sections to clearly display 3D architectures in two dimensions.

`Architecture` (and therefore also in a `Rule`, since it is a subclass of `Architecture`). As described above, an element in the array is used to store the Object data for each individual `Cell`. However, matching rules against local neighbourhoods using this indexed array requires the comparison of each cell in a separate operation.

By maintaining a binary representation of the architecture, we have the means to compare architectures using a format – the integer value produced by the bit array – which can be manipulated very efficiently by the underlying processor. While keeping the two representations synchronised incurs some overhead, rules are matched against neighbourhoods far more frequently than the central architecture is modified by a rule being fired during any simulation run.

As an example of the reduction in computation, when comparing a neighbourhood of cubic cells, the index lookup function (Equation 4.2) must be performed 54 times (27 times for each of the neighbourhoods being compared), and the integer comparison of cell values must be performed 27 times, resulting in $27 + 54 = 81$ operations at the highest level of implementation. The index lookup function itself consists of three multiplications and two additions, giving $27 + (54 * 5) = 297$ integer operations which must be performed for each neighbour match. In contrast, if the neighbourhood is reduced to a single integer, the processor only has one operation to perform during each match. This comparison operation is performed $a \times r \times c$ times during a simulation run, where a is the number of agents in the

Cubic Rotations		
0: no rotation	8 : $xxyyy$	16 : z
1 : x	9 : xxx	17 : zx
2 : xy	10 : $xxxy$	18 : zxx
3 : xyy	11 : $xxxyy$	19 : $zxxx$
4 : $xyyy$	12 : $xxxyyy$	20 : zzz
5 : xx	13 : y	21 : $zzzx$
6 : xxy	14 : yy	22 : $zzzxx$
7 : $xxyy$	15 : yyy	23 : $zzzxxx$

Table 4.5: A listing of the 24 valid cubic rotations, by the number of rotations around each axis. For example, xxy indicates the cube is rotated around the x-axis, then once again, and then around the y-axis.

simulation, r is the number of rules in the rule set, and c is the number of cycles, and so a reduction of computation of this magnitude is clearly of significant benefit.

The number of bits required to store a region of cells is given by $X \times Y \times Z \times S$, where X, Y and Z are the sizes in each respective dimension, and S is the number of possible states each cell could hold. We must therefore fix the number of states available in the simulation when the build lattice is first created. It is important to note that this number can be very big; for a $3 \times 3 \times 3$ rule with 20 possible states, the number of bits required is 540. Even on the most modern processors, the largest number than can be stored as a single integer is only 64 bit long. For this reason, we have used a specialized `BitVector` extension[11] which is capable of manipulating bit arrays of lengths up to 2^{32} bits, which is for our purposes effectively unlimited.

4.5.2 Rotation of Cubic Structures

As seen in Table 4.2, there are three axes of symmetry for any cube. A cube can be rotated to any of four positions around each of the x, y and z axes, giving a total of 24 unique states. The number of unique rotations is derived from the fact that a cube has 6 square faces, and each of those faces when facing upwards on the cube can then be rotated around the centre of the face 4 times, and thus $6 \times 4 = 24$. These unique rotations are shown in Table 4.5.

Intuitively, some rotations might seem to be missing from this list, but in fact they are

[11] BitVector is written by Steffen Beyer and available at http://www.engelschall.com/u/sb/download/

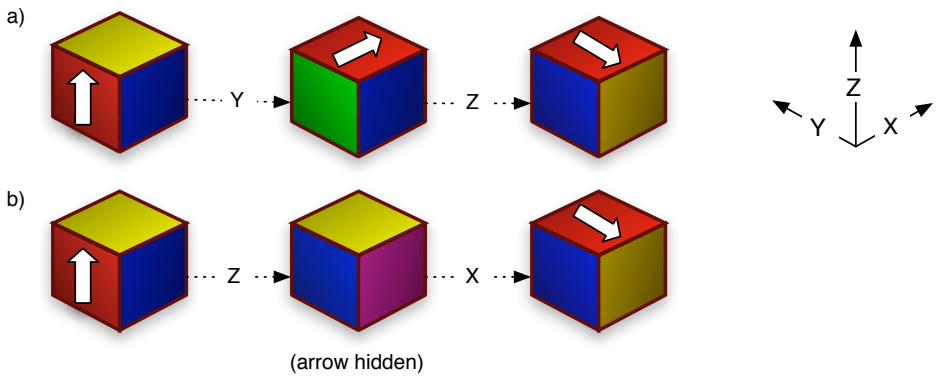


Figure 4.7: Rotational Equivalence. Cubes *a)* and *b)* are initially identical. Cube *a)* is rotated once through the y-axis, and then once through the z-axis. Cube *b)* is rotated once through the z-axis, and then once through the x-axis. Both result in identical orientations. It is important to note that the axes *do not rotate* with the cube.

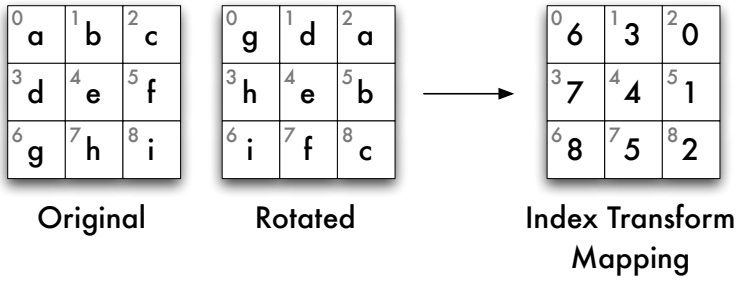
simply equivalent to already present rotations. For example, the rotation yz is identical to the rotation zx, yyz results in the same overall rotation as zzx, and so forth. This can be clearly seen in Figure 4.7.

4.5.3 Rotating Architectures using Index Mapping

Rather than actually transform the architecture in place to match a rotated set of cells, it is simpler and faster to *redirect* coordinate references for one cell to the cell which *would* be in that position if the architecture was actually rotated. This can be achieved by developing a set of mappings between *rotated* indexes and *base* indexes. A simple example of this is shown in Figure 4.8.

The `Cell` which was at index 0 has moved to index 2, `Cell` 6 has moved to index 0, and so forth. The value of a cell (x, y, z) after architecture rotation is now determined by first calculating the temporary index from the coordinates (see Equation 4.2), and using the value at that temporary index within a mapping array (far right in Figure 4.8) as the index of the actual cell value within the *original* array.

The functions which provide these *index transforms* for cubic architectures of size s are given in Figures 4.9, 4.10 and 4.11. It is also important to note that all division in these equations is integer based, and no fractional values are present *at any point* within the



$$a = \begin{cases} s & \text{if } a = 0, \\ (i+1) \bmod s & \text{otherwise.} \end{cases}$$
$$\text{RotateZ2D}(i, s) = (s \times a) - \frac{i \bmod s^2}{s} - 1 \tag{4.3}$$

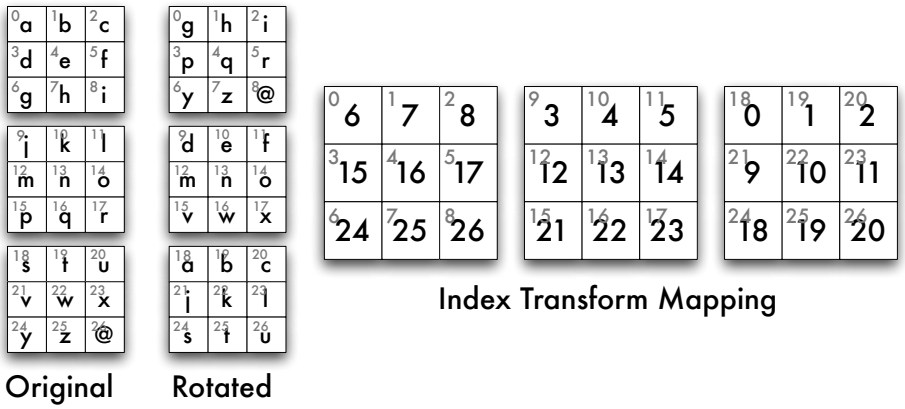
Figure 4.8: An illustration of cell index rotation through the Z axis on a $3 \times 3 \times 1$ ('2D') matrix. Using the first as an example, each element of the transform array can be read as "when rotated through Z once, the cell at index 0 has the value of the cell originally at 6."

expression. For instance, $\frac{1}{2} = 0$, remainder 1. This is important, and is made clear by the absence of otherwise-obvious factor reductions, such as the presence of $(s^2) \times \frac{i}{s^2}$ in $RotateZ$. In this case, $(s^2) \times \frac{i}{s^2}$ cannot be reduced to $\left(\frac{(s^2) \times i}{s^2} = i\right)$.

These equations are derived by carefully considering the movement of cell positions in terms of offsets based on the architecture size. For instance, considering the 2D example in Figure 4.8, a temporary index a is created which represents the 'column' of i (its position on the X axis, in other words). Multiplying this value by s, the size of the architecture side, transforms this 'column' value into a corresponding 'row' (i.e. Y axis) value. For example, if $i = 6$, $a = 0$ and $a \times s = 0$; this tells us that whatever was in position 6 originally will appear somewhere in the top row.

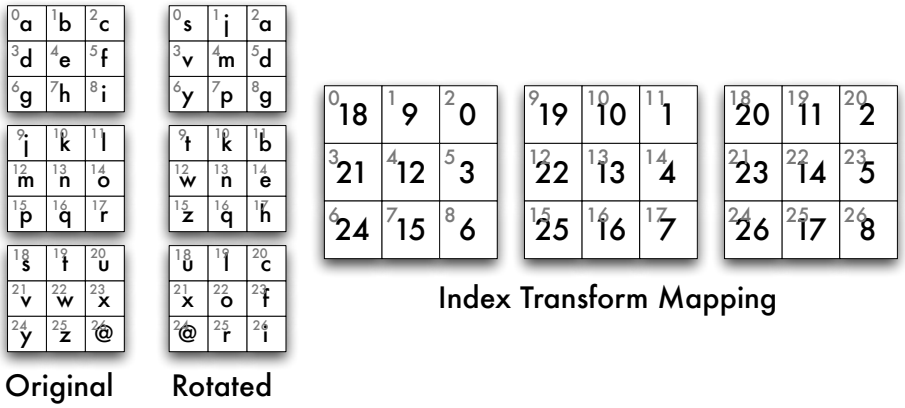
The rest of the terms in the following equations perform similar functions. For instance, in Figure 4.11, the a component steps our rotated index along the Y plane of the architecture by jumping in increments of s. The $(s^2) \times \frac{i}{s^2}$ component serves to increment through the Z planes by jumping by s^2 (This effect can be inferred by comparing it to the equation in Figure 4.8). The -1 term takes into account our indexes begin at 0 rather than 1.

Since the size of the rules is known and does not change while the simulation is running, these *index redirection* matrices can be calculated in advance, avoiding the time-consuming



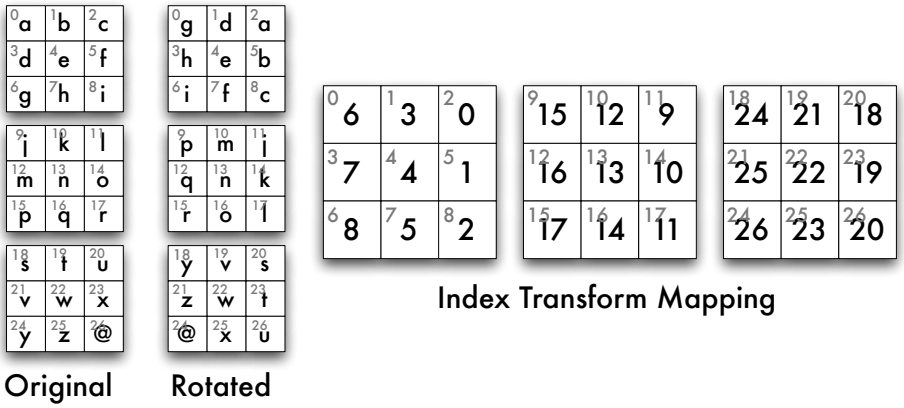
$$\text{RotateX}(i, s) = \left(\left(\frac{i}{s} \bmod s\right) + 1\right) \times s^2 - \left(\left(\left(\frac{i}{s^2}\right) + 1\right) \times s\right) + (i \bmod s) \tag{4.4}$$

Figure 4.9: The mapping and function to produce indexes rotated through the X-axis for cubic architectures of size s.



$$\text{RotateY}(i, s) = (i \bmod s) \times \left(s^2\right) + \left(\left(\left(\frac{i}{s} \bmod s\right) + 1\right) \times s\right) - \left(\frac{i}{s^2} + 1\right) \tag{4.5}$$

Figure 4.10: The mapping and function to produce indexes rotated through the Y-axis for cubic architectures of size s.



$$a = \begin{cases} s & \text{if } a = 0, \\ (i+1) \bmod s & \text{otherwise.} \end{cases}$$
$$\text{RotateZ}(i, s) = (s \times a) - \frac{i \bmod s^2}{s} + (s^2) \times \frac{i}{s^2} - 1 \tag{4.6}$$

Figure 4.11: The mapping and function to produce indexes rotated through the Z-axis for cubic architectures of size s.

repetition of the above calculations. Following each of the rotations listed in Table 4.5, transformation matrices are constructed incrementally by reapplying transform functions to existing rotation matrices. The xxx rotation index matrix is created by first generating the x rotation matrix using the *RotateX* function (Equation 4.4), then creating the xx matrix by applying *RotateX* to the x matrix, and finally a third iteration of *RotateX* to produce the rotated index matrix for three rotations around the X axis. Other rotation combinations are produced in a similar manner.

Just as a bit array can be used to represent the contents of an architecture for fast comparision, bit arrays representing all valid rotations of each rule (up to 24, depending on simulation parameters) are generated and cached within each `Rule` object, indexed in an array as in Table 4.5. In this manner, rotated versions of each rule can be compared simply by look-up, and no calculations need be performed on the `Rule` during the rule-matching cycle of the simulation.

```
def Architecture.setFromInt(int)

  if int == 0 # special case: BitVector doesn't like '0'
    fillWith(Cell::Nil)
  else
    numBricks = @xsize*@ysize*@zsize
    bitVectorSize = numBricks*Cell::NumTypes
    tempBitArray = BitVector.new_from_int(int, bitVectorSize)

    tempBitArray.resize(bitVectorSize)

    numBricks.times { |x|
      @cells[x] = Cell.new(tempBitArray.chunk_read(Cell::NumTypes,
                                                   x*Cell::NumTypes))
    }


    if @bitArraysEnabled
      @bitarrays[0] = tempBitArray
      updateBitArrays() #recalculate all of the rotated bitarrays
    end
  end

  refreshCellCoords()
  initBoundingBox()
  updateBoundingBox()

end
```

Figure 4.12: Ruby code from the cubic `Architecture` implementation, which converts a large integer value into a series of `Cell`s within a cubic lattice.

4.5.4 File Format

The cubic architecture implementation takes advantage of the ability to convert a neighbourhood of arbitrary size into a bit array and subsequently into a single integer number, when saving and loading structure data. If we are given the original dimensions of the architecture and we have this (very large) integer value, we can reconstruct the original cell lattice.

A fragment of code from the cubic `Architecture` implementation is shown in Figure 4.12. This function is responsibly for reloading the large integer value into a `BitVector` object, and then extracting sub-sequences of bits to be interpreted as the individual `Cell` values. Finally, the bounding box (see Section 4.7.3) is created and the cell coordinates (used for display purposes only) are created.

4.5.5 Summary

Within cubic `Architecture` objects, `Cell` data is stored in an array, and methods translating 3D coordinates to array indexes are provided. Furthermore, rotated index matrices are calculated and cached, allowing fast rotation of the cell data. Finally, bit-array versions of the structure (and all valid rotations) are generated and cached to enable very fast comparison of `Rule`s as the simulation runs.

4.6 Hexagonal Geometry

Hexagonal architectures are based upon stacked planes of hexagonal cells. Like squares and triangles, hexagons tile without gaps, making them ideal as the basis for an abstract discrete representation of space. Hexagons are present throughout nature, featuring all scales from microscopic crystal arrangements to macroscopic cells in insect nests.

Unlike cubic architectures, hexagonal structures do not lend themselves to representation in n-dimensional arrays; the relationship between array elements cannot accurately represent the relationship between face-adjacent hexagonal cells without a large amount of additional processing.

It is for this reason that the hexagonal `Architecture` implementation in *Nest-3.0* stores `Cell` objects in a graph-like structure. Each hexagonal `Cell` maintains references to its 6 surrounding neighbours and the `Cell`s above and below, as illustrated in Figure 4.13. The references are stored in an 8-element array, indexed using direction constants `N`, `NE`, `SE`, `S`, `SW`, `NW`, `UP`, and `DOWN` (internally mapped to integer values from 0 to 7).

Generating a Hexagonal Lattice

Because the hexagonal geometry implementation is a graph, rather than an indexed array, no coordinate system exists for mapping between discrete locations and the cell data. Instead, the hexagonal implementation relies directly on the links between cells as shown in Figure 4.13. In other words, rather than incrementing a Z coordinate and using Equation 4.2 to access the cell data, the links between cells themselves are used to traverse the structure as

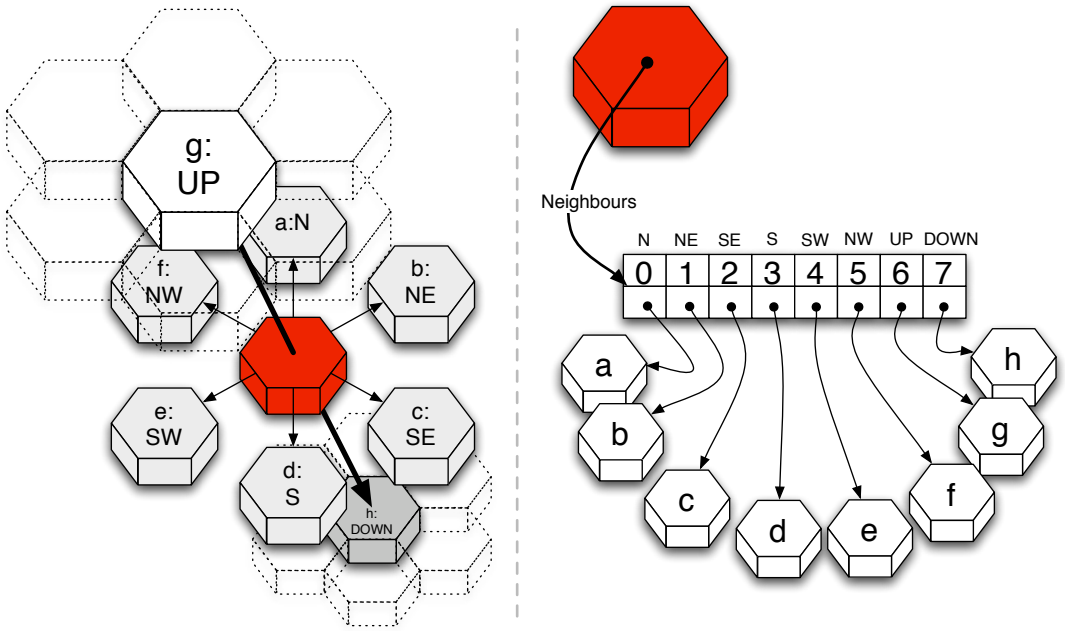


Figure 4.13: Inter-`Cell` linkage in the hexagonal lattice. Each `Cell` is linked to the 8 face-adjacent cells – six within the same plane of cells, one above and one below.

it is stored within memory.

It is therefore crucial that the integrity of these links is maintained; if `Cell` A links to `Cell` B in the `UP` direction, then `Cell` B **must** link to `Cell` A in the `DOWN` direction. As the architecture grows and new `Cells` are created, care must be taken to ensure that links are correctly created, since the lattice represents the fundamental spatial universe in which the architecture and agents will exist and interact.

It is relatively simple to ensure that the `Cell` linking described above can be performed automatically and consistently, if the placement of new cells is restricted to locations where the new `Cell` will be face-adjacent to existing cells. However this, does not avoid all potential linking problems.

Consider the situation shown in Figure 4.14, at which point a new `Cell` is about to be created in position **X**. Since all cells are created relative to another, cell **X** will be created as a new `Cell` in the `NW` direction of `Cell` 5. It is clear from the visual representation that `Cell` **X** must also be linked to `Cell` 1, but the only way to algorithmically determine this is by analysing the shape of the path between `Cell` **X** and `Cell` 1 (via `Cells` 5, 4, and so on),

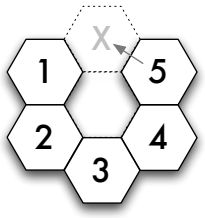


Figure 4.14: Closing the loop – a cell placed `NW` of Cell 5 must also be linked in the `SW` direction to Cell 1.

and then determining whether or not that path is reducible to a direct link.

In general the situation is much worse. Upon creation of `Cell` **X**, *all possible paths* through the graph of `Cell`s must be considered to determine which other `Cell` objects might be adjacent to the new `Cell` and require linking. Clearly this is computationally unacceptable. Some means of providing the 'big picture' must be provided so that linking of new `Cell` objects is accurate and consistent.

HyperCells

The cubic spatial representation differs from the hexagonal representation in that by using a co-ordinate system, the cubic implementation has a global means of addressing individual cells. Within a graph, such as that used to implement the hexagonal cell structure, there is no such global addressing system; `Cell` objects must be found by exploring their neighbours.

It is exactly this which produces the problems illustrated in their simplest case in Figure 4.14. In order to overcome this, we must provide some global addressing system for the hexagonal graph.

Consider a single plane of hexagonal cells. Each group of 7 'real' cells within the plane are all linked as 'children' to a single hexagonal cell on a plane which exists *above* this plane, dimensionally speaking. What we mean by this is that, rather than being linked as an `UP` neighbour (see Figure 4.13), it is linked `ABOVE`. This 'hypercell' is the *parent* of each of these 7 cells, as illustrated in Figure 4.15. The parent hypercells for adjacent groups of 'real' cells are adjacent cells in this 'hyperplane'. In this way we have a lattice which exists *above* our original arrangement of cells.

We can then use this 'hyperlattice' to provide a wider view of the architecture to the

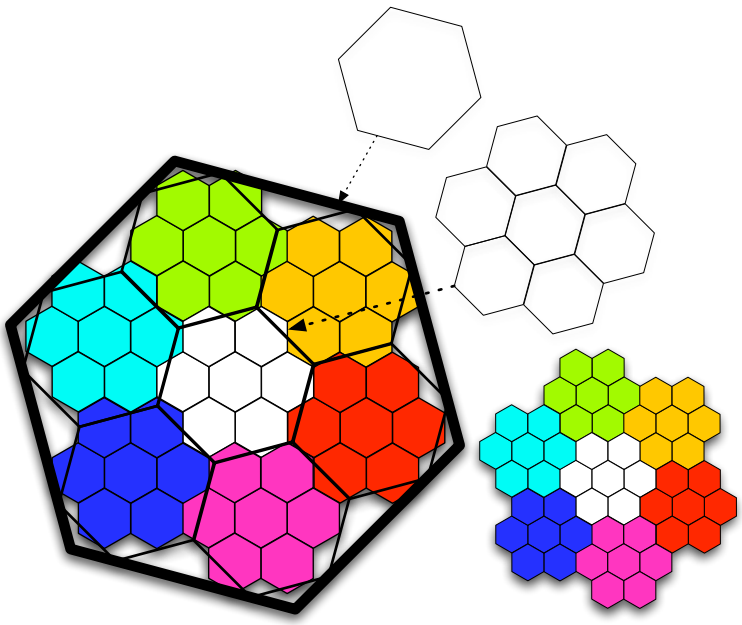


Figure 4.15: Hierarchies of hypercells are used to define sub-regions within the lattice.

methods that must perform the linking. This is illustrated in Figure 4.16. Whenever a new cell is created within the lattice, it must be created *from* a cell with already exists in the structure. For instance, `Cell1.set(N, CELL_RED)` would create a new cell in the `N` direction from `Cell1`, and set the value of this new cell to be red. What we must ensure is that the new `Cell` object is also correctly linked to any *other* adjacent cells.

Each `Cell` object maintains a *dirToParent* property, indicating the real direction in which the `Cell` lies directly 'under' the *parent*. Using this direction in combination with the direction in which we are creating the new Cell, we can determine the hypercell under which the new `Cell` should be linked. Since each hypercell tracks all the cells beneath it, we can then use that information in the hypercell to determine which cells are adjacent and link them to the new `Cell` object appropriately.

With some consideration it can be seen that this mechanism relies on the 'hyperlattice' being correctly linked; we have simply deferred the original problem. However, the hyperlattice has $\frac{1}{7}$ of the cells of the original lattice, and thus the task of ensuring valid inter-cell links has become simpler. Rather than accept this significant reduction in complexity, we can apply some of the previous logic and produce a much more elegant solution. Since the

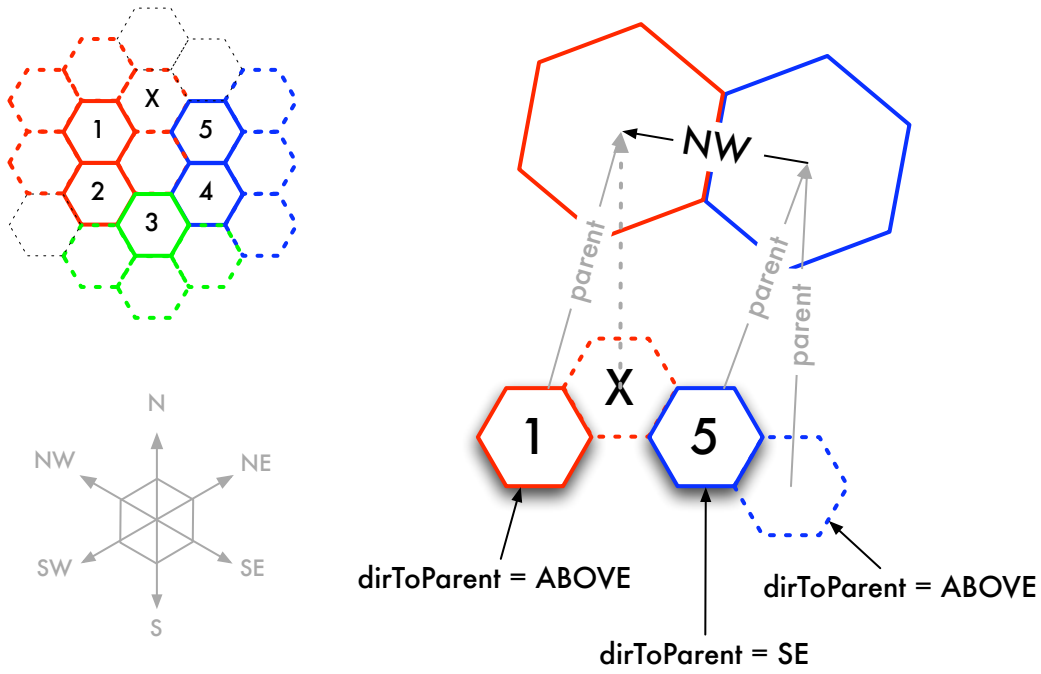


Figure 4.16: Linking hexagonal cells using a 'Hyper Lattice'.

hyperlattice was created to enable accurate linking within a 'real' lattice, we can create *hyper-hyperlattice* and exploit the same mechanisms described above to link the hyperlattice itself, and so on. This hierarchy of hypercell objects extends from the level of the lattice in which agents and bricks are placed, up until there is only a single 'hypercell' at the highest level, and is illustrated in Figure 4.17.

4.6.1 Rotation of Hexagonal Structures

Hexagonal neighbourhoods have only one axis of symmetry – the z-axis passing through the central cell. Through this axis, a hexagonal rule can be rotated 6 times. Determining the `Cell` data under any given rotation can be found simply by applying a rotation offset to the original direction constant, and taking the remainder from a division by six (the number of directions within the rotated plane), as shown in Equation 4.7. Because the directions valued 0 to 5 are those which lie within the plane (`N`, `NE`, `SE`, `S`, `SW` and `NW`), the rotated index within this remainder will always lie within the plane.

$$\text{cellDataIndex}(dir, rotatation) = (dir + rotation) \bmod 6 \tag{4.7}$$

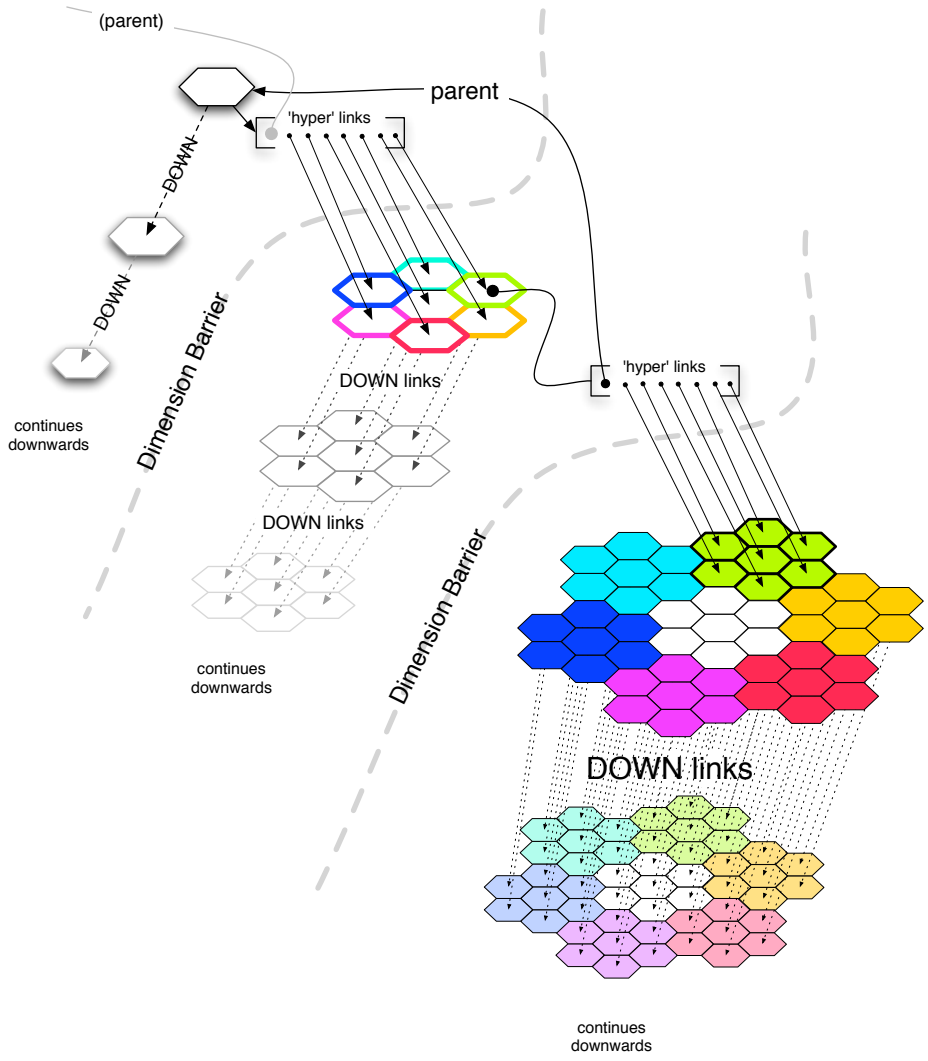


Figure 4.17: The HyperStructure used to construct the interlinked `Cell` network. The lattice space is *expanded* through dimensions, each dimension using neighbourhood information from the dimension above in order to ensure that inter-cell links are valid in the final 'real space' dimension.

```
30
1
1=4,1,0 : -1 2 -1 6 -1 -1 12002 12001
2=4,6,0 : -1 -1 3 -1 1 -1 12005 12004
6=5,2,0 : 1 -1 5 -1 -1 -1 12017 12016
3=4,5,0 : -1 -1 -1 4 -1 2 12008 12007
4=7,4,0 : 3 -1 -1 -1 5 -1 12011 12010
5=6,3,0 : -1 4 -1 -1 -1 6 12014 12013
```

Figure 4.18: The initial fragment of an architecture file representing a simple ring of cells. The header consists of the first two lines, and each subsequent line details a single `Cell` object within the structure.

An addition pseudo-symmetry can be created by placing the hexagonal rule 'on its head', effectively inverting the `UP/DOWN` component of the structure and reversing the planar directional component. A simple function, shown as Equation 4.8, is used to reverse any direction within the hexagonal lattice.

$$\text{ReverseDirection}(d) = \begin{cases} DOWN & \text{if } d = UP, \\ UP & \text{if } d = DOWN, \\ ((d+3) \bmod 6) & \text{otherwise.} \end{cases} \tag{4.8}$$

4.6.2 File Format

Hexagonal `Architecture` objects are stored in a simple plain-text file format, which consists of a simple header and then a series of lines, each describing a `Cell` within the lattice. An example of the file which produces the completed ring architecture shown in Figure 4.14 is shown as Figure 4.18.

The header consists of two lines, detailing first the maximum vertical depth of the lattice in which the architecture is constructed (30 in this case, which is the default for new hexagonal architectures) and secondly the unique `id` of the central cell of the architecture. This cell is used as a point of reference as each of the following lines is parsed and new `Cell` objects are created and inserted into the `Architecture`.

The rest of the file is a listing of value and neighbour information for each `Cell` within the architecture. Each line can be separated into three logical segments, delineated as

`id = <values> : <neighbours>` where `id` is an integer value, `<values>` is a comma-separated list of integers and `<neighbours>` is a final list of integers. Each of these segments is outlined below:

`id` is a unique integer used to identify this `Cell` object within `<neighbours>` contents in the rest of the rule.

`<values> = <val>, <order>, <rule>` where: `val` is the integer state information or this `Cell`; `order` is the ordering assigned to this `Cell` within the whole architecture; and `rule` is the `id` of the rule which placed this `Cell`object. Any of these values can be -1, which indicates an `UNDEFINED` value.

`<neighbours> = <N> <NE> <SE> <S> <SW> <NW> <UP> <DOWN>` where each of those values is the integer `id` of the cell which is linked in that direction from this `Cell`. If the value is -1, then no cell is present in that direction.

4.6.3 Summary

Since the geometry and cell relationships within a hexagonal lattice do not naturally fit with a coordinate scheme of cell addressing, a graph of linked objects is created to represent this lattice geometry.

This serves to simplify operations such as rotation of neighbourhoods, but complicates the creation of the lattice due to the possibility of mis-linked `Cell` objects within the lattice.

4.7 Simulation Implementation

In this section the details of the simulation aspect of the *Nest-3.0* system are described. This component is responsible for transforming sets of rules back into completed architectures.

The simulation system implemented in *Nest-3.0* is actually fairly simple; much of the required functionality is built into the data structures being manipulated. Only two objects are present in this part of the system: `Simulation` and `Agent`. They are both described in detail in the sections below.

4.7.1 The `Simulation`

A `Simulation` object is a simple container for the information relevant to a single simulation. It gathers together the rule set, agents and several limiting parameters and wraps them with a very simple layer of code to enable simple simulation runs to be performed.

Initialisation

The minimum set of parameters required by the `Simulation` object are:

`rule-file` the name of a file which contains the rule set to be used by agents in this simulation.

`cycle-limit` the maximum number of simulation cycles to be performed before the simulation halts. The default value of this parameter is 1000, meaning that unless specified otherwise the simulation will halt regardless of progress once 1000 cycles have been completed.

`brick-limit` the maximum number of bricks to be built before the simulation halts. The default value of this parameter is 50, meaning that unless otherwise specified the simulation will halt once 50 bricks have been placed within the architecture.

`num-agents` the number of agents to be created within this simulation. This parameter must be greater than 0, i.e. every simulation requires a single agent.

Given values for these parameters, a `Simulation` object can be created. As a part of this process, the build architecture lattice is created (an empty lattice within which agents are placed and bricks will be deposited during the simulation) and the limit parameters are set. The file containing the rule structures is loaded and converted to a simple array of architectures ready for comparison. The required number of `Agent` entities are created and given access to the rule data for this simulation. The agents are finally assigned random locations within the build lattice. The simulation is at this point in a position to start running

```
def runCycle()
  # call the run method for each agent
  @agents.each { |agent| agent.run() }

  # increment the number of cycles executed
  @cycles += 1

  # return true if either of the stopping conditions have
  # been met
  return ((@architecture.numBricks < @brickLimit) and
          (@cycles < @cycleLimit))
end



def runSimulation()
  @cycles = 0
  while runCycle() do
    # this loop will finish when runCycle() returns false
  end

  puts "Simulation complete: #{@cycles} cycles, " +
       "built #{@architecture.numBricks} bricks"

end
```

Figure 4.19: The *Nest-3.0* simulation run cycle code, implemented in Ruby.

Simulation Run Cycle

The run cycle of the simulation is very simple, since most of the logic is actually present within the `Agent`s themselves. The code which makes up the simulation run cycle is shown in Figure 4.19. On a current typical desktop computer[12] the simulation runs very quickly, and the building behaviour of the agents can he difficult to observe in any great detail. The seemingly-artificial split into two methods shown in Figure 4.19 facilitates interactive simulation from the user interface - by calling the `runCycle` method rather than the `runSimulation` method, a single iteration of agent activity can be processed and the results displayed immediately.

[12] At the time of writing an 'entry level' PC sports a 2.5 gigahertz processor and 512 megabytes of RAM; future computers will certainly be more powerful.

Simulation Results

The built architecture can be saved using the same mechanisms present for any `Architecture` object. Within this object, each `Cell` is tagged with the `id` value of the `Rule` which caused the `Cell` to change value, and also with its sequence number within the overall architecture construction. This information can be used to replay the simulation and extrapolate the causal links between the rules at a later date by examining the `Rule id` information for each neighbourhood of rules. In this way, all simulation results can be stored within the same format as is used to hold the architecture information itself, resulting in a single file format to be maintained.

4.7.2 Agent Implementation

As can be seen in Figure 4.19, the implementation of a simulation run cycle is very generic. This is intentional, as it allows the behaviour of agents within the simulation to be implemented exclusively within the `Agent` class. This in turn means that it is trivial to use new agent implementations without modifying the surrounding simulation implementation. The only requirement is that the `Agent` object responds to the `run` method.

The simple agent behaviour provided within the default `Agent` implementation is shown in Figure 4.20. This agent does not maintain any internal state other than position, and attempts to move into an adjacent empty location. If no empty locations can be found adjacent to this location, the agent is 'magically' transported to another empty cell within the build lattice. This prevents the agent being left in a location into which it has just built a brick.

The `build` method is equally simple: all rules are checked in sequence, and if a rule matches the current location – the `buildMatches(<Cell>)` function compares all `Cell`s except the build `Cell`, which holds the desired change of value and will necessarily not match the value in the build lattice – then the `Rule` is selected for firing. The location value is modifed according to the value in the build `Cell`, and this modified `Cell` within the built architecture is tagged accordingly.

As mentioned above, each agent is represented by a separate instance of the `Agent` class,

```
def run
  build()
  moveRandomly()
end

def build
  @rules.each { |rule|
    if (rule.cc.buildMatches(@location))
      @location.setValue(rule.buildValue())
      @location.setOrder(@simulation.architecture.numBricks)
      @location.setRuleID(rule.cc.order())
      @location.expandSpaceAround()
      break # we only want to match a single rule
    end
  }
end

def moveRandomly()
  # create a copy of the array of available directions
  directions = @allowedDirections.dup

  # find a cell in a random valid direction with is empty
  until directions.empty?
    dir = directions.delete_at(rand(directions.length))
    newcell = @location.get(dir)
    if newcell != nil
       # stop if this cell is EMPTY
      break if (newcell.value() == CELL_EMPTY)
    end
  end

  if directions.length == 0
    # this means we couldn't find anywhere to move, so warp somewhere
    @location = @simulation.findEmptyCell()
  else
    # set this agent's new location
    @location = newcell
  end

end
```

Figure 4.20: The Ruby code describing the behaviour of a simple *Nest-3.0* agent. This agent first matches its current location against the rule set, and builds a brick if a match is found. Secondly it attempts to move into an adjacent empty `Cell` object. If no empty cell can be found, the agent is moved to a random empty location within the structure ('warping').

and as such they maintain their own unique location information. This implementation of agent behaviour does not feature any of the more sophisticated extensions that were outlined at the beginning of this section. However it is relatively trivial to implement examples of this behaviour. For instance, Figure 4.21 shows a modification to the movement function which ensures that the agent only moves to an empty cell where it is in contact with the surface of a non-empty cell, or in other words *moves along the surface of the construction*.

The key difference in this modified movement function is the line flagged with `***`, which includes the additional neighbour checking. Within this line a copy of the cell neighbour array (see Figure 4.13) is created, and `Cell`s are removed from this duplicate if they are empty. If any `Cell`s remain, there must be a brick adjacent to this `Cell` and therefore it is suitable to move to. Similarly the 'warping' fall-back has been modified to include this constraint. It can be seen from the very small change in the code that Ruby enables extension and modification of agent behaviour in an extremely painless manner.

Additional extensions might modify the `build` or `move` methods to check other properties of the agent's location, such as the number of agents currently occupying that cell, and modify its behaviour appropriately.

4.7.3 Simulation Optimisations

Several programming optimisations have been used to improve the performance of the simulation, by streamlining the computation used in common situations, and reducing the likelihood of wasted simulation cycles. These optimisations are outlined below.

Bit Array Matching

As described above, within the cubic implementation, neighbourhood configurations are reduced to a single integer value for the heavily-utilised matching function. This provides a dramatic speedup since each individual cell does not need to be located and compared.

```
def moveAlongSurface()

  # create a copy of the array of available directions
  directions = @allowedDirections.dup

  # find a cell in a random valid direction with is empty
  until directions.empty?
    newcell = @location.get(dir)
    if newcell != nil
      if (newcell.value() == CELL_EMPTY) &&
         # *** note this additional check ***
         (newcell.neighbours.dup.delete_if{|n| n.isEmpty()}.length > 0)
        break
      end
    end
  end

  if directions.length == 0
    # this means we couldn't find anywhere to move, so warp somewhere
    @location = @simulation.findEmptyCell()
    # keep searching until we find a suitable empty cell
    while (@location.neighbours.dup.delete_if{|n|
                n.isEmpty()
           }.length == 0) do
      @location = @simulation.findEmptyCell()
    end
  else
    # set this agent's new location
    @location = newcell
  end

end
```

Figure 4.21: A modified agent movement function which ensures that the agent is always in a cell which is face-adjacent (i.e. a direct neighbour) of a cell which is *not* empty.

Agent Bounding Box

Another optimisation present within the cubic implementation is a bounding box, which restricts agent movement to a confined region of the total build lattice. The region defined by this bounding box is determined dynamically as construction of the architecture progresses. As each brick is built, its coordinates are checked against the current bounding box values. In each direction, if the brick coordinate is equal to either the minimum or maximum coordinate of the bounding box in that axis, the bounding box limit is expanded to ensure that the minimum and maximum values are at least one cell less than or greater than the extremities of construction respectively. If an agent attempts to move to a region outwith this bounding box, it is denied and an alternative new location within the bounding box must be found.

By maintaining a bounding box, agents within simulations which do not employ surface-constrained movement behaviours are naturally restricted to a small region of space around the construction, increasing the likelihood that an agent will encounter a stimulating configuration of `Cell`s. This in turn speeds up building behaviour of the simulation as a whole, reducing the amount of computation required to build architectures of a given size.

There is no bounding box present within the hexagonal implementation, because `Cell` objects are only created around the extremities of the construction as it is being built. In this manner, the architecture as a whole is constantly surrounded by a single layer of empty `Cell`s by default. The line `@location.expandSpaceAround()` within the `build` method in Figure 4.20 ensures that when a `Cell` value is modified, enough new empty `Cell` objects are created around the current location to guarantee this natural bounding box.

4.8 Future Extensions

Outlined here are some possible extensions to the *Nest-3.0* system which could be undertaken at a later point to improve the flexibility and usefulness of the tool to those who wish to explore the behaviour of stigmergic systems.

4.8.1 More Geometries

The cubic and hexagonal geometries supplied lend themselves perfectly to a lattice simulation, because they tile perfectly in two and three dimensions. However, it would be very interesting to see which structures can be simply expressed using a stigmergic algorithm in alternative geometries. The next obvious regular shape which provides a space-filling tiling across the plane is the triangle (or a triangular prism in three dimensions), but this could be considered a finer-grain version of the existing hexagonal geometry.

Beyond lattices, an implementation of stigmergic systems using a more realistic representation of space (and therefore an alternative representation of stigmergic rules) would not only form the basis of a more biologically-plausible model of stigmergy in social insects, but may give far clearer insights into the effects artificial geometries impose on stigmergically-constructed architectures. The design of the system architecture (shown in Figure 4.4) is such that the simulation logic can operate on any new spatial representation which conforms to a very unrestrictive programming interface.

4.8.2 Agent State

While the capabilities for persistent individual agent state are already present in the *Nest-3.0* system, there is no interface to manipulate this functionality or modify the building or movement behaviour according to this internal state. The development of such an interface would therefore be a natural next step in the life of the software.

4.8.3 Pheromones

As with internal states above, while it is trivial to extend the `Cell` model to accommodate new floating-point state information, and thus support the propagation of pheromones throughout the environment, there is no interface for specifying and manipulating this information.

Modifying agent behaviour according to this extended local state would necessitate a more sophisticated simulation creation interface, in which these advanced agent behaviours could be specified. This was not developed further as both internal state and advanced

environmental interactions diverge too far from the central topic of this thesis, but remain intriguing extensions which can doubtless be used to produce very interesting swarm behaviour in the future.

4.8.4 Architecture Evaluation

As noted in Chapter 3, some initial attempts at automatic architecture evaluation were provided in the *Nest-2.11.1* system, along with further work on automatic evaluation by a genetic algorithm in [19, 22]. Since the focus of the *Nest-3.0* system is the evaluation of random stigmergic algorithms, these features are not as relevant to our goals, but the inclusion of such evaluation mechansims would certainly be interesting from the perspective of the user.

4.9 *Nest-3.0* Summary

The *Nest-3.0* stigmergic simulation software has been presented. It supersedes the original *Nest-2.11.1* software developed by Bonabeau and Theraulaz in almost every respect by providing a multi-platform system for stigmergic investigation. Many of the artificial limits present in the original software have been effectively removed, and the scope for experimentation dramatically broadened.

The software core upon which *Nest-3.0* has been constructed provides a more suitable basis for further investigation of abstract stigmergic systems than could ever be possible using the original software, and this was the primary motivation for its inception. It has also been shown to be easily extensible by end users to provide agent behaviours which vastly surpass both the original system behaviour and the needs of the remainder of this investigation.

Chapter 5

Automatic Generation of Coordinated Stigmergic Algorithms

Overview

At this point, it is useful to review what we have considered thus far, and so refresh the motivations for this thesis. In Chapter 3 we were introduced to the simple model of collective stigmergic building behaviour developed by Bonabeau et al.[158, 156] and implemented in the *Nest-2.11.1* software. We also considered the investigations performed using this software, and the use of both extensive random sampling and genetic algorithms to explore the space of behaviours this system could exhibit. Finally, the restrictions of this simple model were highlighted.

Chapter 4 introduced a revised model, designed to overcome the limitations of the original. The *Nest-3.0* model included a vastly improved agent model, a generalised rule-matching system and dramatic increases in the flexibility and extensibility of the system. An implementation of this model has been presented in the *Nest-3.0* software. By considering which constraints present in the original model can be relaxed, and how this might be achieved, a greater understanding of how such constraints may influence the range of system behaviours is achieved.

The purpose of both these systems is for the greater part identical, and is outlined

perfectly in [156] – **"our goal is to explore the space of possible architectures that can be generated with a stigmergic algorithm"**. These simulation systems simply serve as tools to enable this exploration.

In this section, the assertions originally made by Bonabeau and Theraulaz[156] regarding stigmergic algorithms will be described, explored and tested. The notions of *coordinated algorithms* and *coherent architectures*, while mentioned earlier, will be considered in detail in an effort to further understand these assertions and the behaviour of abstract stigmergic systems. Most importantly, the shortcomings of these concepts as a means of achieving the above-stated goal will be explored.

At this point, a new set of concepts – *pre-* and *postrules* – are introduced as a more concrete and powerful means for exploring the potential of abstract stigmergic systems. The applications and again most importantly the shortcomings of these concepts when used as tools for exploring stigmergic behaviour are discussed, along with their relationship to the concepts developed in the original *Nest-2.11.1* work. Finally, a roadmap for the further investigation of stigmergic system behaviour is presented as the basis for the remainder of this thesis.

5.1 Coordinated Algorithms

In this section we will examine the assertions originally made by Bonabeau and Theraulaz regarding the conditions necessary for a stigmergic algorithm to produce a 'desirable' structure.

5.1.1 Building Stages

In their original work, Bonabeau and Theraulaz[158, 156] discovered that while abstracted stigmergic systems such as those modelled using *Nest-2.11.1* were capable of producing interesting and highly-structured architectures, the algorithms which produce such structures are very rare:

> "[We] found *a posteriori* that structured shapes can be built only with special

algorithms, coordinated algorithms, characterized by emergent coordination."[156]

Here lies their main assertion regarding the behaviour of stigmergic systems of this type – only algorithms which exhibit this particular *coordinated* property will produce architectures that are in some way *structured*. The question of what exactly defines a 'structured' architecture will be considered later, but what characteristics are given to define a *coordinated* algorithm? The following definition is given in [156], and will be used as the basis for discussion of 'coordinated algorithms'.

> "[At] any given time, there must exist an active set of local configurations that all trigger the same qualitative type of brick deposit. Let $(C_1, C_2, \ldots, C_n)$ be the set of all local stimulating configurations, that is the configurations which trigger the building behaviour (put down a brick) if they are encountered. In order for the construction to proceed in a coherent way, there must be a succession of a certain number of qualitatively distinct building states. Let $(S_1, S_2, \ldots, S_n)$ be the set of these building states. Each of these states is characterized by a subset of C, $C(S_p)$, with $\cup_p C(S_p) = C$ and $\forall p_1 \neq p_2, C(S_{p1}) \cap C(S_{p2}) = \oslash$. Considering each building state S_p, each time a brick is put down, the result gives rise to one or more configurations $\in C(S_p)$, and the construction process may go on either in a sequential manner (a single configuration allows the agents to put down a brick – not to be confused with a sequential algorithm), or in a parallel manner (in this case, several configurations allowing the deposit of a brick are simultaneously present).
>
> The completion of the building state S_p then corresponds to the appearance of new configurations $\in C(C_{p+1})$."[156]

This compact description is somewhat hard to follow, and so for our benefit it is considered carefully below from a critical standpoint. At this point we are already familiar with the components of *Nest* systems, and so where possible we can use this knowledge to re-frame some of the more confusing terminology used above and obtain a stronger grasp on what might be inferred.

Coordinated Algorithms, Step By Step

1. *"At any given time, there must exist an active set of local configurations..."*

The set of local configurations can be defined simply as the set of neighbourhoods containing one or more bricks around an empty cell. The active set must be those locations in which a brick could be built next, i.e. those locations where a rule matches the local arrangement of bricks.

2. *"...that all trigger the same qualitative type of brick deposit."*

Since the qualitative nature of the construction behaviour is simply the colour of brick to be placed, it is unclear why at time t only the building of a single type of brick should be allowed.

3. *"Let $(C_1, C_2, \ldots, C_n)$ be the set of all local stimulating configurations, that is the configurations which trigger the building behaviour (put down a brick) if they are encountered."*

Set C is the set of rules in this simulation.

4. *"In order for the construction to proceed in a coherent way, there must be a succession of a certain number of qualitatively distinct building states. Let $(S_1, S_2, \ldots, S_n)$ be the set of these building states. Each of these states is characterized by a subset of C, $C(S_p)$, with $\cup_p C(S_p) = C$ and $\forall p_1 \neq p_2, C(S_{p1}) \cap C(S_{p2}) = \oslash$."*

The collection of rules is divided into non-overlapping subsets, which are termed *building stages* or *building states*. A 'stimulating configuration', or rule, may be present in one and only one *stage*. This non-overlapping quality must be the 'qualitatively distinct' property mentioned here; it is difficult to see what else it could refer to.

5. *"Considering each building state S_p, each time a brick is put down, the result gives rise to one or more configurations $\in C(S_p)$..."*

Each rule within a specific *building stage* modifies the environment, creating another set of local environments, one or more of which are also within that same *stage*. This is the essence of stigmergy – actions within the environment modify that environment, leading to situations where new actions become available. What is said here, however, is that those new actions must belong to the same *stage* as the action which occurred previously.

6. *"... and the construction process may go on either in a sequential manner..."*

Only a single new matching local configuration of bricks is created by this environmental modification, and as such there is only one rule which can fire in this situation.

7. *"... or in a parallel manner (in this case, several configurations allowing the deposit of a brick are simultaneously present)."*

In this case, the set of local configurations created by this modification matches more than one rule, so building may continue at more than one location around where the previous rule fired.

8. *"The completion of the building state S_p then corresponds to the appearance of new configurations $\in C(C_{p+1})$."*

So if any of the new configurations is not present in S_p, a new state has been entered, and this state is complete.

9. *"Such states are at the root of the modular structures that appear in the architecture. One may have the two following cases (where R_i denotes the set of responses generated in state S_i):*

a strictly linear chain of building states:

$$S_1 \xrightarrow{R_1\downarrow} S_2 \xrightarrow{R_2\downarrow} S_3 \xrightarrow{R_3\downarrow} S_4$$

a chain of building states that can have recurrent states:

$$S_1 \underset{R_4\uparrow}{\overset{R_1\downarrow}{\rightleftarrows} S_2 \overset{R_2\downarrow}{\rightleftarrows} S_3 \overset{R_3\downarrow}{\rightleftarrows}} S_4$$

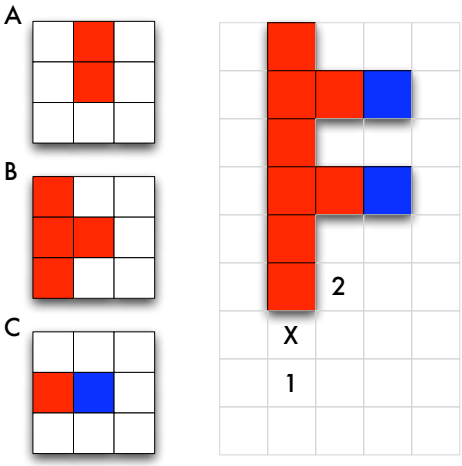


Figure 5.1: A simple set of rules building a coherent architecture. Each rule in this example can be considered a distinct *building stage*. If Rule **A** fires at location **X**, then simulating configurations for both rule/stage **A** and rule/stage **B** are present within the system (at locations **1** and **2** respectively.

It is possible and often desirable that an environmental modification produces configurations which match more than a single building state, as seen with the simple system illustrated in Figure 5.1, so the strict linearity/cyclicality implied here seem over-constrained.

The informal summary of this definition is provided as follows:

> "...if one wants a swarm of agents to build a given architecture, one has to decompose it into a finite number of building steps, with the necessary condition that the local configurations that are created by a given state and which trigger building actions, differ from those created by a previous or a forthcoming building step so as to avoid the disorganization of the building activity. We call such a building algorithm ... a coordinated algorithm, because all individuals cooperate in the current building state at any time."[156]

This serves to only marginally clarify what still appears to be a fairly vague assertion – *in order to ensure that the construction of the structure does not career out of control, the causal links between stimulating configurations must be managed in some way.*

Furthermore, the notion of *building states* as described above seems to belie the type of building behaviour we should expect or even rely upon in a swarm system. If multiple agents are simultaneously active and modifying the environment around the constructed structure,

they will often be constructing elements of the architecture which are not part of a single building 'stage'. As is seen in Figure 5.1, agents can be involved in the construction of distinct 'stages', or intuitively distinct elements of the structure, without any compromise of the resulting architecture. The concept of explicit *building states* is not entirely without use, but simply may **not be necessary** to ensure the production of interesting or desirable architectures.

Building Stages, Emergent Systems and External Observers

Finally, we must consider how useful the concept of building stages might be as we attempt to achieve our goal of "explor[ing] the space of possible architectures that can be generated with a stigmergic algorithm". At their strongest, Bonabeau et al.[156, 22, 19] assert that the existence of stages is an *indicator* that a 'coherent' architecture will be built if such a rule set is used in simulation. However, no means for concretely identifying the presence or absence of these stages is provided or even suggested:

> "Although coordination is an intrinsic property of the algorithm, it is not obvious to see it in the rule table, and it is often necessary to run the algorithm to discover the coordination it produces."[156]

In other words, the behaviour of the system may only be understood once the system has been observed running. A somewhat more concrete means of identifying such stages is possible through the generation of a 'construction graph'[19] (see Section 3.4.1), but despite this apparent formalisation the concept of building states or stages remains ambiguous, and their identification only possible *during or after* simulation.

This is strikingly similar to the position we must adopt with other emergent systems, and as such brings us no closer to being able to exploit the advantages of a distributed system of this nature. Clearly we must consider other means of exploring and exploiting the behaviour of these systems.

5.2 Coherent Structure: Measuring the Value of Architectures

The initial work by Theraulaz and Bonabeau on lattice swarm systems[158, 156] was presented without a satisfactory definition of *coherent structure* – the particular type of structure that only a *coordinated* algorithm can produce. Instead, architectures were evaluated subjectively, and some characteristics of such *coherent* architectures were outlined:

> "The notion of coherent architecture is obviously difficult to formalize... a given coordinated rule table always converges toward architectures that possess similar global features... On the contrary, architectures resulting from successive simulations using the same noncoordinated algorithm are very dissimilar."[158]

> "This tendency [for noncoordinated algorithms] to diverge comes from the fact that stimulating configurations are not organized in time and space and many of them overlap, so that the architecture grows in space without any coherence."[156]

It is asserted that some algorithms tend to converge upon one particular architecture, with individual simulation runs producing variations in the exact structure, but remain architecturally very similar. For instance, a particular algorithm may always produce a vertical column of bricks, with planes 'grown' perpendicular at intervals down its length, such as architecture **A** in figure 5.2. While the exact spacing between planes may vary between simulations, the structural features are nevertheless striking and always present.

This is a result of "the implicit hand-shakes and interlocks that are setup at every [building] stage"[156]. On the other hand, *uncoordinated* algorithms tend to produce structures which are seemingly random, space-filling and which do not exhibit structural similarity across simulation runs; "algorithms with overlapping configurations yield structureless shapes, never found in nature"[156].

Thus we must determine the 'coherency' of a structure according to the the repetition of 'features' across several simulation runs, and by how 'natural' the structure appears to

the observer. The latter criterion is almost certainly a purely subjective judgement by the observer, but the former – the presence of 'features' – might show promise for formalisation. This is considered more fully in Chapter 9 as part of the final aim of this research.

5.2.1 Existing *Nest-2.11.1* Measures of the 'Coherency' of Structures

In order to measure the fitness of structures and apply genetic algorithms techniques to the generation of random coordinated algorithms, a function attempting to provide a measure of coherence was devised in [22] and further explored in [19]. The initial fitness function was based purely on the behaviour of the system during simulation, rather than the architecture produced. The fitness function was "based on a simple observation: in algorithm that generate coherent architectures, many micro-rules are used, whereas in algorithms that generate structureless shapes, one one rule or a few rules are actually used in the course of the simulation."[22]

This function is refined to correlate further with observer's ratings of coherence in [19] using measures of compacity of the architecture (the number of face-adjacent bricks in the architecture divided by the total number of bricks) and the complexity of various 'patterns' identified in the structure. Because these measures are shown to correspond with the ratings given by observers to a sample set of architecture, the function is asserted as a satisfactory measure of meaningful structure within an architecture.

These experiments have been considered earlier (see Section 3.4), and will not be discussed in more detail here. We have seen in the previous section that the notion of *building stages* may not be as robust as one might hope; here we will see that the framework for considering *coherent architectures* is similarly subjective, and may not be useful as a tool for understanding the true behaviour of stigmergic systems.

5.2.2 Architectural 'Features'

If we consider the two architectures shown in Figure 5.2, we can see they are clearly distinct from each other. Intuitively, architecture A exhibits a great deal of structure, whereas

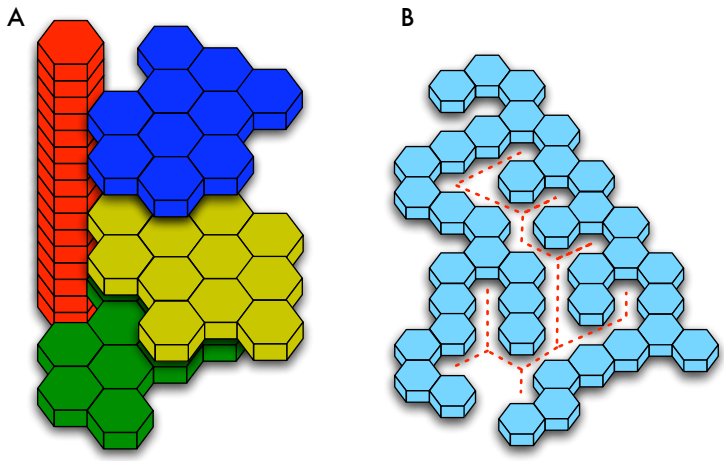


Figure 5.2: Architecture **A** – a set of planes from a vertical column – seems obviously structured. Architecture **B** initially appears random and space-filling, until we consider the regions of empty space defined by the structure.

architecture B appears to feature a large degree of randomness. As observers, we find it easy to describe A as *flat planes projecting from a single column of bricks*, whereas B might be described as *a randomly branching tree-like structure*.

Based on these descriptions, it follows that we might want any automatic measure of structure to value architecture A highly, and the same measure provide a significantly lower value for architecture B. However, what we must question is what really constitutes an architectural 'feature'. As indicated in Figure 5.2, if we consider the empty space with architecture B, rather than the bricks themselves, we can see that a clear feature exists – corridors of empty space, each a single cell width across at any point. This corridor feature could be considered equally as 'natural'[114][1] as the more-obvious structure in A.

It is clear given this insight that our original assessment of the lack of structure in architecture B was mistaken. While the multi-planar arrangement of bricks in architecture A might be simpler to describe, particularly in a hierarchical manner, this form of description cannot satisfactorily capture what it is that makes a structure 'coherent', 'interesting' and ultimately 'desirable'.

[1]Such corridor structures might even be considered closer to those seen in nature than the nest-like structures cited in [156], since the insects must live in the free space *within* the construction, whereas the *Nest* model implies that bricks, unlike the cells within real nests, are solid objects.

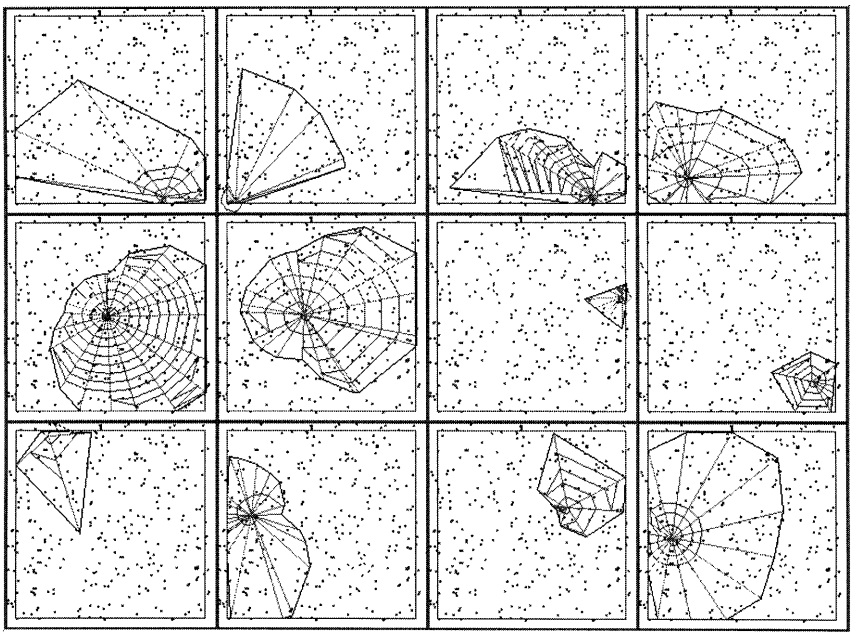


Figure 5.3: Spider webs undergoing functional evaluation using the *NetSpinner* software. *Illustration taken from [103].*

Functional Evaluation of Structures

The *desirability* of an architecture in this context can be measured either through correspondence with some specific output we expect, or by the expression of structural features which are functionally successful. Obtaining an objective measure of either is certainly non-trivial.

The work presented here is not concerned with functional properties or structures, although they are certainly important. However, as an illustration of how other simulated structural products have been evaluated, two such efforts are presented.

Perhaps the most striking example of an evolved structure which was successfully functionally evaluated are the spider webs produced by Krink et al. in [103]. A genetic algorithm was used to modify the values of several parameters governing the construction of the web. The resulting web 'solutions' are then produced and their functional effectiveness and efficiency evaluated by randomly "peppering the webs with artificial prey" and deriving a final fitness based on the position and value of prey caught (prey which struck over a line of 'silk') and the amount of material used in the web. This is illustrated in Figure 5.3.

A second prominent example of structural evolution and functional evaluation is given in the evolution of Lego^{TM}structures by Funes and Pollack [66, 67]. In this study, real brick

structures were designed using a simulated model of the possible connections between Lego bricks, which could then be constructed in reality and tested. Various different structures were evolved, including bridges (which must support weight over some distance), scaffolds (structures reaching a certain height), crane arms (structures capable of supporting a given weight with only a single base point) and tables (structures capable of supporting a given weight over some specified surface area).

Both the Lego constructions and spider webs above are structures with clearly-defined purposes, and it is this aspect of the formulation of these studies which allows solutions to be evaluated objectively. In contrast, the nest structures presented in [158, 156, 22, 19, 17] have only the goal of being 'interesting', 'structured', 'coherent' or 'biological-like'. Only one of these criteria – 'biological-like' – suggests a function or purpose which might be testable. For instance, we might consider a generated structure a *good* nest if a large internal volume of space is completely enclosed (save for some entrance) by a minimal number of bricks. Alternatively, we might define the quality of a nest structure based upon the presence of certain features - planes, columns, corridors and so on.

However, deriving precise definitions of such structural features is certainly non-trivial, as will be seen in Chapter 9. In reality the functions of structures built by social insects are multitudinous and complex, including temperature regulation, food production, and assisting in the production of new generations – all behavioural aspects of social insect life which are certainly far more complex to model than the intersection of randomly placed prey with a spider web, or even the well-understood mechanical physics of a Lego construction.

What should be now evident is that the quality of any structure is can only be measured given the domain within which that structure is to be considered, and a clear definition of the purpose(s) within that domain which that structure must fulfil. In the case of the original nest work, while the domain was loosely biological, no requirements are specified other than it should 'appear' biological. This is clearly too vague a criterion from which a strong measure of structural quality might be derived.

Experimental Bias

The assertions made originally in [158, 156] spring from two key aspects of their investigation: a random sampling of algorithms did not produce any significant structures, while it was possible to produce stigmergic algorithms by hand which *could* produce life-like architectures.

> "Although we present almost exclusively structured architectures. . . structureless architectures are much more likely to be generated if an arbitrary rule table is used."[158]

> "[The factorial correspondence analyis indicates] the existence of features common to all coherent architectures at the algorithmic level."[158]

The existence of common features can only be claimed for the group of *hand-composed* algorithms originally considered. Furthermore, since these algorithms were all created by hand, following the same mental processes and directed specifically towards producing nest-like structures, it is unsurprising that they occupy a close region of the algorithmic space, as was later noted:

> "The apparent smoothness of the mapping from algorithms to architectures . . . [may[be an artefact of the proceedure used to design algorithms"[22]

For example, many of the structures depicted in [158, 156, 17, 22, 19] feature planes extending from around a column structure. The rules required to build a space-filling plane will be common to each of those algorithms, and those which build columns will again be similar. This closeness may in fact be an artefact of the limited series of architectures created by the investigators, and the biological correspondence which the initial explorations into the capabilities of the *Nest-2.11.1* model sought to achieve.

5.2.3 Structural Coherence through Behavioural Consistency

The deterministic quality of coordinated-algorithms would also appear to be more tenuous than originally declared:

> "[A] coordinated rule table always converges toward architectures that possess similar global features; sometimes, the result is even deterministic... On the contrary, architectures resulting from successive simulations using the same noncoordinated algorithm are very dissimilar."[158], 1995

> "It is not true, however, that every unstructured architecture is generated with a nonconvergent algorithm: some of them can be produced consistently in all simulations. moreover, even in shapes built with coordinated algorithms, there maybe some degree of variation, which is higher in cases where the number of different choices within one given building state is large."[17], 2000

We see that some coordinated algorithms are observed to be deterministic, and others nondeterministic, with varying degrees of freedom dependent on how tight the 'interlocks' between rules are, but also that noncoordinated algorithms can be both deterministic and nondeterminstic too. Therefore it cannot be possible to establish the coordinated nature of an algorithm using only the similarity of architectures produced by that algorithm over a number of simulations.

5.2.4 *Structure* without *Stages*

Figures 5.1 and 5.4 serve as examples of very simple stigmergic systems which produce a clear structure without the definition of any stages whatsoever. The latter in particular could be considered the antithesis of Bonabeau and Theraulaz's assertions – the resulting architectures exhibit interesting properties at a closer level of inspection than the observation of repeating units within the global structure:

Corridors and Chambers Both structure **A** and **B** display the same global features - corridors of a single brick width, and chambers with a maximum size of 2×2 bricks.

Bricks vs Space Ratio When simulated within a 12×12 space, giving 144 cells in which bricks *could* be placed, structure **A** contains 75 bricks, and structure **B** contains 74. The ratio of filled-to-empty space is therefore 1.09 and 1.06.

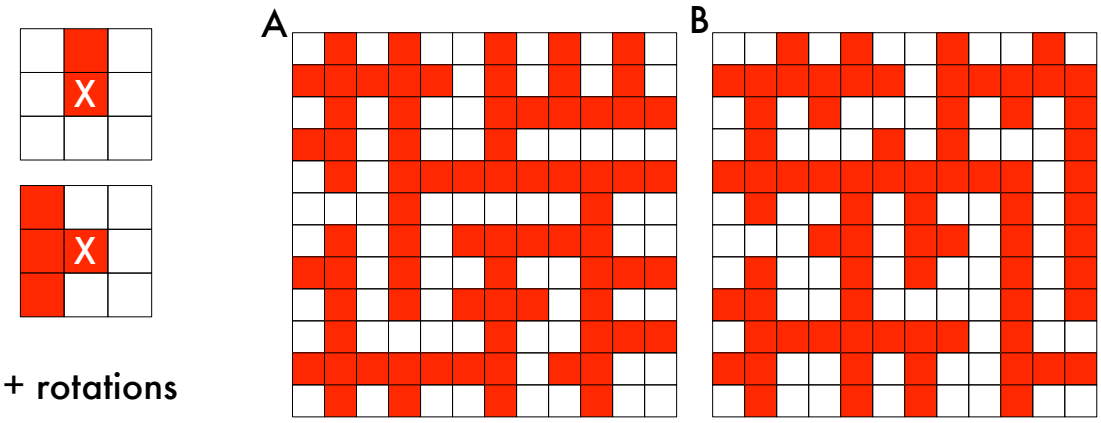


Figure 5.4: With rotation enabled, seemingly-complex architectures with clearly defined *features* (in this case corridors and chambers of definite sizes) can be produced with extremely simple rule sets. The results of two different simulations are shown in the right, both using the same two rules on the left, plus all rotations through the Z-axis (effectively 8 rules if fully specified).

These two features are common to every architecture produced through simulation of this simple rule set. The production of regular-sized corridors is a result of the fact that in both rules, a brick is only placed if there are no bricks to either side of that location. This implicitly guarantees that every single brick will be framed by empty cells to either side, but since the size of the neighbourhood only stretches a single cell around the build location, it is possible (and likely in this case) that the location 2 cells from this one will be filled.

The chamber size of at most 2×2 is also a direct result of the empty-cell padding feature of the rules in combination with the limited neighbourhood size. Since the first rule, if simulated on its own, would only produce a straight line of bricks, the second rule in the system is responsible for creating all corners within the architecture. This rule may place cornering bricks at any point along a line where it is not directly adjacent to any other brick. Typically we would expect the pattern shown in architecture **A** in Figure 5.5, but there it is equally likely that the situation in architecture **B** will be produced, with a gap of 2 cells between bricks placed by this 'cornering' rule.

However, if a gap of 3 cells is present between two bricks, the neighbourhood centred around the central empty cell can be matched against the second rule, and thus at some point during the simulation – if left to run for a sufficient number of cycles – a brick *will* be placed there, producing the same local structure as architecture **A**. In this manner, the

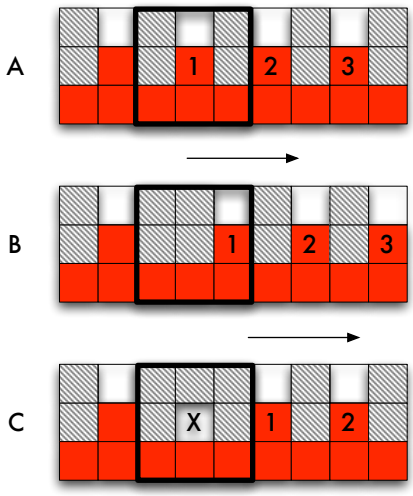


Figure 5.5: Illustrating the maximum cavity width of 2 bricks. Cells which are greyed out represent locations where no rule will match at any point. In architecture **C**, it is not possible to prevent the 'cornering' rule from firing in location **X** and thus producing an identical architecture to **A**.

maximum line length of bricks which is not interrupted by a 'corner' brick is 2, and thus the maximum size of cavity which can be defined by these rules is 2×2.

This example serves to illustrate that interesting architectures with significant (or potentially useful) properties can be produced from very simple rules, with requiring any notion of building stages, or the type of strict high-level coordination that they imply. It was suggested in [19] that "no 'interesting' pattern can be generated using a single type of brick"; these examples might therefore through some doubt on the definition of 'interesting' used as a basis for the above assertions.

Finally, given the definition of abstract stigmergic systems in Section 3.1, a rule-set is a flat sampling of the space of possible simulating configurations with no internal sub-divisions into 'stages'. In order to fully appreciate the behaviour of a system of rules, the relationships between all rules within it must be explored and understood.

5.2.5 The Perception of Structure

The identification of 'coherent' structure must be dependent on an observer's ability to recognise and characterise features – their ability to recognise patterns[42, 8, 102] (see Section2.1.1). In figures 5.2 and 5.4 it was shown that architectures which might be considered

random and unstructured may consistently exhibit significant and interesting properties, despite the apparently simplicity of the algorithms used in their construction. Recognising this alternative form of 'structure' requires a shift in perception and criteria by those who are measuring such aspects of the architecture; even as simple a change as considering the nature of the *empty space* created within an architecture rather than the specific arrangement of bricks.

The relative quality of 'structure' within any particular architecture therefore depends on the choice of which specific criteria are selected as important contributors to the measure of 'structure'. It would seem that in the original work by Bonabeau, Theraulaz et al.[156, 158, 19] such criteria might be the presence of large regular columns, repeating modules whose size is around that of the neighbourhood an agent can sense, and above all a regularity in brick placement. Based on the above figures, it has been shown that other, equally valid criteria might be used to determine the desirability or otherwise of a stigmergically-produced structure.

The importance of selecting appropriate structural measures exactly mirrors that of selecting specific behavioural aspects of multiagent systems in order to describe *emergent behaviours*, as was discussed in Section 2.1.4.

5.3 Beyond Stages: Post-Rules

We have seen that the concept of stages, as outlined above, only provides an *a posteriori* indication of the behaviour of one particular set of rules; a recognition of the origins of various high-level features in the resulting structure, as opposed to an understanding of how those features are produced, and most importantly how their production influenced the behaviour of the system. How could such an understanding of the system dynamics be gained?

5.3.1 The Simplest Stigmergic System

Let us consider the simplest possible stigmergic system, shown in Figure 5.6. It consists of a single rule matching against only one brick and using only a single brick colour in the

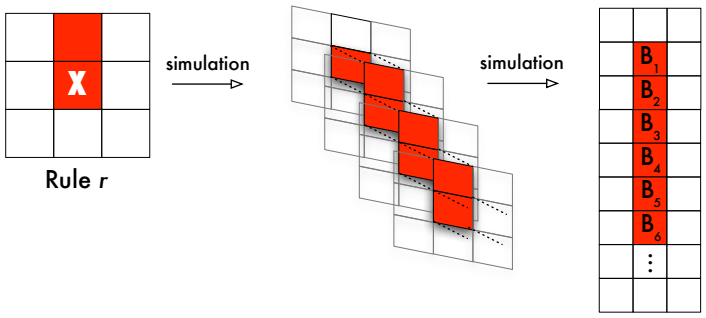


Figure 5.6: The simplest stigmergic system, consisting of a single rule which fires to produce a straight line of cells. The system will continue building until manually stopped.

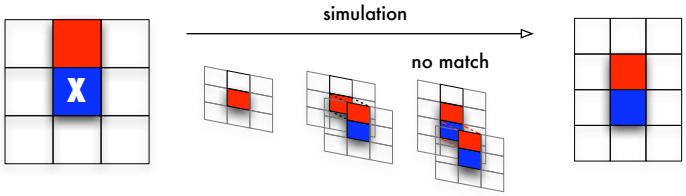


Figure 5.7: The single-rule stigmergic system, this time using a different colour for the build brick. After matching once, the rule cannot fire again.

whole system. As in the the original *Nest-2.11.1* system, the environment initially consists of a single `RED` brick – B_1 in the figure. The behaviour of this system is fairly self-evident: if simulated it produces a straight line of bricks, without limit. The process which results in this straight-line structure is also fairly simple to infer. Rule r matches against any `RED` brick which is surrounded on the 'east', 'south-east', 'south', 'south-west' and 'west' sides by `EMPTY` cells. When rule r matches in the cell 'south' of this single brick, it places brick B_2 in that cell. This brick, like the initial brick, is surrounded in those same directions by `EMPTY` cells. The only place which the system can match a rule in this new environment is 'south' of B_2, resulting in the placement of B_3. This behaviour will continue indefinitely.

As a consequence of rule r firing and placing a brick, a configuration of bricks now exists against which rule r can match again. It can be said that when rule r fires, it *enables* rule r – itself – to fire again. Since this abstract system is entirely deterministic (there are no other factors which may affect how rules fire), we can suggest that this rule *causes* itself to fire again.

In contrast, if we change the colour of the brick built (as in Figure 5.7), then the rule can

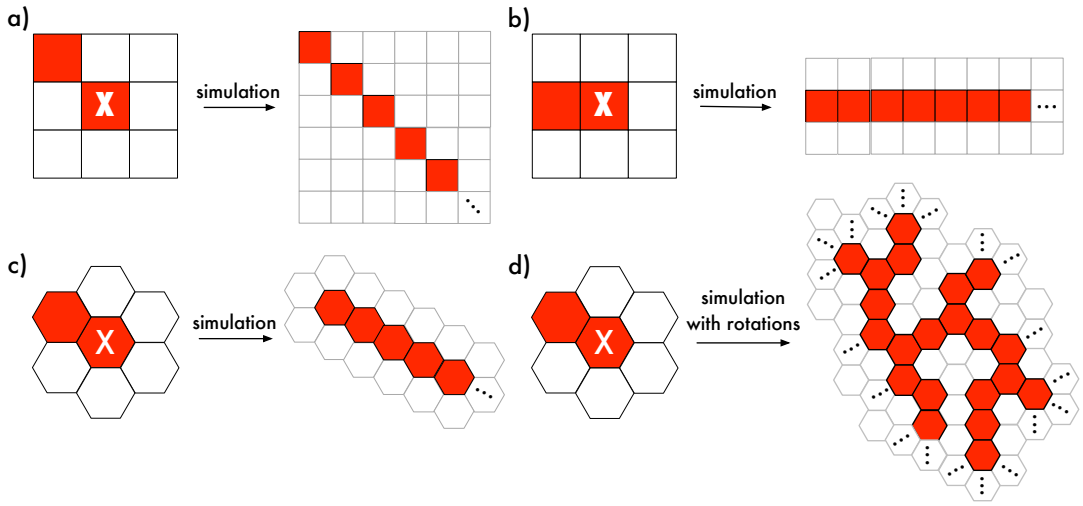


Figure 5.8: Further examples of single-rule stigmergic systems. In each case, construction continues until externally halted. All systems except for **d)** produce a straight-line structure. The structure produced by d) has structure, but because of the allowed rotations the line produced is not straight.

only fire a single time; it does not produce any configurations which allow the rule to fire again. This is a clear demonstration of the use of differing brick colours to control the firing relationships between other rules (including this rule itself). This is, in fact, the sole purpose of multiple brick colours within a stigmergic algorithm, and goes some way to explaining the assertion in [19] that two or more colours are required to produce an 'interesting' architecture. This nature of method of control is explored further in Chapter 7.

Figure 5.8 displays more examples of stigmergic construction with single-rule algorithms. It is trivial to determine the behaviour of these systems with only a small amount of effort expended by the observer, and only system **d)** requires slightly more consideration before its behaviour can be summarised. Most collections of rules do not, unfortunately, give up their secrets so readily. While such simple systems might be easily comprehensible by us, we will require assistance of some form if we are to grasp more complex stigmergic systems. The purpose of this thesis is to describe some efforts towards developing techniques and algorithms towards this end.

5.3.2 The Simplest Rule and The Two-Colour Assertion

It was noted in discussions of Bonabeau and Theraulaz's original work[19, 29] that the authors believed no interesting architecture could be generated using an algorithm with only a single brick colour. We are now in a position to more fully understand the algorithmic basis behind such an assertion.

Since every architecture begins with a single brick placed in the environment, in order for an algorithm to produce *anything*, a rule must exist within that algorithm which matches against a single brick. The 'simplest rules' presented in Figure 5.8 are rules of exactly this type. If we consider only rules **a)** and **b)** in this figure, along with versions rotated 90°, 180° and 270° around the z-axis (the line formed by looking 'down' at the centre '**X**' brick from above), all possible 2D cubic rules which might match against a single brick have been produced. It is also now easy to note that each one of these rules will produce an environment in which it can fire again immediately and indefinitely[2], creating a single straight or diagonal line.

Similar rules exist for 2D hexagonal and both 3D systems. The only means available to prevent this continuous building, if it is undesirable, is by modifying the colour of the brick to be built, as shown in Figure 5.7. Since *all algorithms* must include a rule of this general structure in order to match against the single initial brick, all algorithms must therefore include at least two brick colours to build architectures in the controlled and predictable manner Bonabeau & Theraulaz describe as 'coherent'.

5.4 A Note Regarding Geometries and Dimensionality

It should be noted here that while most of the figures in this section are presented as two-dimensional cubic systems for consistency, the techniques and algorithms described are *not* specific to this cell arrangement, and can be applied to *any* abstract spatial lattice representation. The only concepts which we rely upon are the existence of regular arrangements of linked cells, each of which can maintain some discrete value.

[2]This type of rule is termed *self-activating*, and will be discussed directly in Section 5.5.4.

For the sake of clarity and conciseness, the examples presented within the majority of this thesis contain only two dimensions. To illustrate the nature of the constraints between rules and the behaviour of the *Nest* system in three dimensions would not only require far more cognitive effort by the reader but also a great deal more space. This is simply as a result the increase in neighbourhood size within 3D lattices, allowing a greater number of possible (yet qualitatively equal) rules to be constructed within the chosen spatial abstraction.

Despite this, the underlying methods and conclusions apply equally to 3D structures and stigmergic algorithms as they do to the 2D examples presented hereafter. At the level of abstraction which we are considering, the spatial lattice arrangement of neighbouring cells is only a convenient method of presenting structure, but this arrangement only exists within the visualisation of the *Nest* system.

Underlying this representation is an abstract collection of nodes which are connected in a uniformly regular manner to neighbouring nodes; in other words, a constrained graph with no spatial context[1]. It is trivially possible (although not useful) to represent a 3D lattice within two dimensions, simply by rearranging the bricks and cells as they are displayed, without destroying or creating any new relationships between them.

5.4.1 Post-Rules

When considering those systems above, the sequence of rule firing exhibited during simulation is clear in our minds: rule r fires, which leads to another rule firing, which then subsequently leads to yet another rule firing. The chain of events then becomes obvious. It also happens that in those cases depicted in Figures 5.6 and 5.8, each event within that sequence involves the same, single rule.

Most interesting systems, however, contain more than one rule, and often many more[3]. When presented with a set of rules, the most relevant question to be answered is simply *which rule(s) will fire next?* Consequently, we may ask *which rule(s) can fire after that?* These questions can be restated more formally:

> Without prior examination of the behaviour of the simulation, if Rule X has

[3]The algorithm presented as an appendix in [156] contains 66 rules, for example.

fired at time t, what are the **post-rules** of Rule X, i.e. which rule(s) can fire at time $t + 1$?

By carefully considering the mechanisms we have defined for the underlying operation of our abstract stigmergic systems, we can devise a means of automatically determining those rules which may be eligible to fire at time $t + 1$.

Neighbourhoods, Rules and Stimulating Configurations

Before embarking on a detailed discussion of rule matching behaviour in abstract stigmergic systems, it is worthwhile repeating the clarification of some terms from Section 3.1.1. These terms will be used frequently in the following chapters:

A Neighbourhood is simply a collection of cells (*neighbours*) which are spatially adjacent to one cell in particular (the *central cell*); each cell has surrounding it a *neighbourhood* of cells.

A Rule is an arrangement of cells, identical in size and configuration to a **Neighbourhood**. Whereas a neighbourhood is a subset of the cells *in situ* within the global simulation space, rules do not *exist* within the spatial lattice; they are *examples* of neighbourhoods. Furthermore, the *central cell* in a rule defines the modification to the environment if the rule 'fires'.

A Stimulating Configuration is simply a **Neighbourhood** which is matches in a cell-to-cell comparison with a **Rule** in this stigmergic system. The comparison generally ignores the contents of the *central cell*, since it will normally be `EMPTY` in the neighbourhood, and filled with a brick in the rule[4].

Unless stated otherwise, the dimensions of neighbourhoods, rules and stimulating configurations are identical within the context of any particular stigmergic system.

[4]When considering excavation (see Section 4.2.5) this situation might be reversed. Here we consider only the permanent placement of bricks, as in the original simulations outlined in Section 3.1. Excavation and post-rules will be discussed in Section 5.4.4

Post-Rule Generation

Consider the single rule r which comprises the simple stigmergic system in Figure 5.6. At time t, the set of stimulating configurations within the environment is C, where $r \in C$. At some point in time during a simulation's run, there will come a point at which r is able to fire[5]. If r fires at time t, then at $(t+1)$ the set of stimulating configurations in the environment is now $C \cup C_r$, where C_r is the set of stimulating configurations created *as a direct result* of the brick being placed by r. Since no other rules have fired between t and $(t+1)$, the only neighbourhoods which have been modified are those which include the location in which r built – the neighbourhoods centred around each *empty* cell surrounding the location in which r placed a brick. These neighbourhoods are the new stimulation configurations, C_r.

Consider the simple rule labelled *Rule r* in Figure 5.9. If the central brick (marked 'X') is placed, the neighbourhood of each of the surrounding cells (labelled 1 to 7) is modified. By firing, rule r has given rise to these new stimulating configurations, shown around the central rule in Figure 5.9. Rules which match these stimulating configurations are those which can fire as a *direct* result of r firing – we call these rules the **post-rules** of rule r. For example, given the new neighbourhood around cell 4, we now know that a rule which builds at time $t+1$ in cell 4 must match against the `EMPTY` cells 2, 3, 5 and 6.

However, because some of the cells in each of these new neighbourhoods lie outside the scope of the neighbourhood of r, we cannot immediately tell what cell values each post-rule might have. For this reason, each cell lying outside the scope of the original neighbourhood of r is given the value `UNDEFINED`.

We have already seen that neighbourhoods and rules are effectively different interpretations of the same set of cells. Only two steps are required to transform a copy of the neighbourhood around cell 4 in Figure 5.9 into a rule:

1. **Replacing `UNDEFINED` Cells** – Each of these `UNDEFINED` cells can hold any value valid within the constraints of this particular stigmergic system.

2. **Setting the Build Cell** – the build cell should hold a brick, or in other words the

[5]It is quite possible, and in fact very common if the stigmergic algorithm is generated randomly, that such a situation where r could fire will never arise. However, for the purposes of this argument, we assume that we have selected r to be a rule which *does* fire at some point during the simulation.

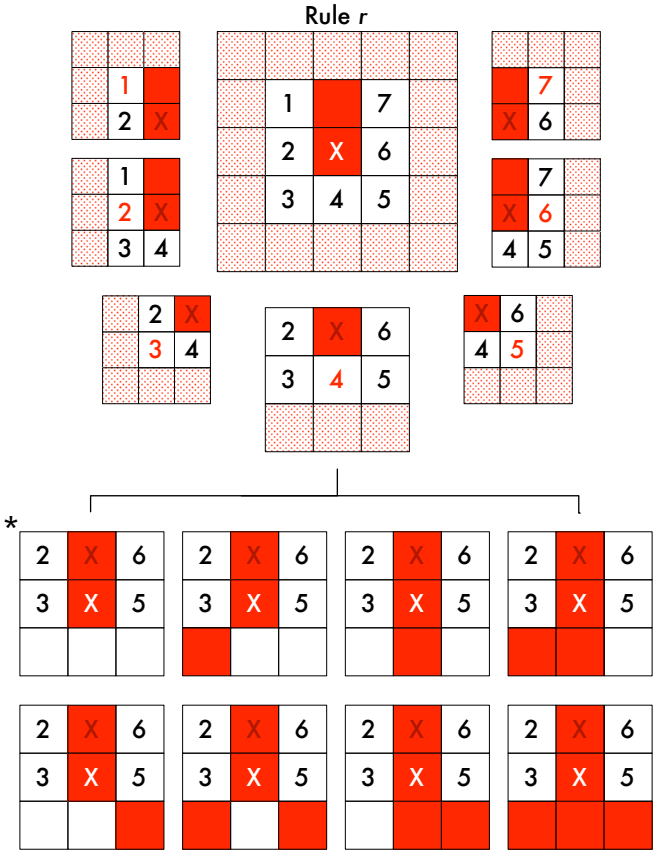


Figure 5.9: Determining the post-rules of the single rule from the simplest stigmergic system. *Meta-rules* for each of the 7 empty cells are generated. Dotted cells in *meta-rules* may be either filled or empty. The expansion shown contains the original rule itself, showing that it will cause itself to fire repeatedly. `UNDEFINED` cells are shown with a dotted fill .

Given a stigmergic system with possible cell values S, and a rule within that system r:

1. The set of post-rules for r, $P_r = \emptyset$.
2. For each `EMPTY` cell e in r:
   (a) Create a **meta-rule** m by extracting the local neighbourhood around e. Cells which are not present in r are set to `UNDEFINED`.
   (b) For each value $u \in (S - \{\texttt{EMPTY}\})$:
      i. Set the value of the build-cell (the *central cell*) of m to be u.
      ii. Let U_r be the set of all `UNDEFINED` cells in r, and A the set of all permutations of $|U_r|$ cell values from S.
      iii. For each combination of values $a \in A$, assign a to U_r and add this rule to P_r.

Figure 5.10: The post-rule generation algorithm.

cell should have any value, **except** `EMPTY`.

For instance, if we have imposed the constraint that only a single colour of brick may be used, each of the `UNDEFINED` cells can be either `EMPTY` or `RED`. The build cell cannot be empty, so it must be set to `RED`.

Rules which contain any `UNDEFINED` cells are called **meta-rules**. A meta-rule can be 'expanded' into a set of normal fully-defined rules (rules which contain no `UNDEFINED` cells). This is achieved by generating all possible permutations of valid cell values over the subset of cells within the rule which are `UNDEFINED`. In other words, we have u `UNDEFINED` bricks, and $|S|$ valid cell values, and we must generate every unique way that these bricks can be assigned these values; the number of unique assignments is given by $|S|^u$.

The expansion of a single meta-rule taken from rule r is shown in the lower portion of Figure 5.9. Here we can see that the row of three `UNDEFINED` cells is assigned every possible combination of `EMPTY` and `RED` values to produce $2^3 = \underline{8}$ unique rules. Since there are only two values in this particular system, the sequence of 'filled' and 'empty' cells in this example mirrors a binary counter progressing from decimal 0 (binary 000) to decimal 7 (binary 111).

The algorithm for generating all post-rules from a given rule r is shown in Figure 5.10.

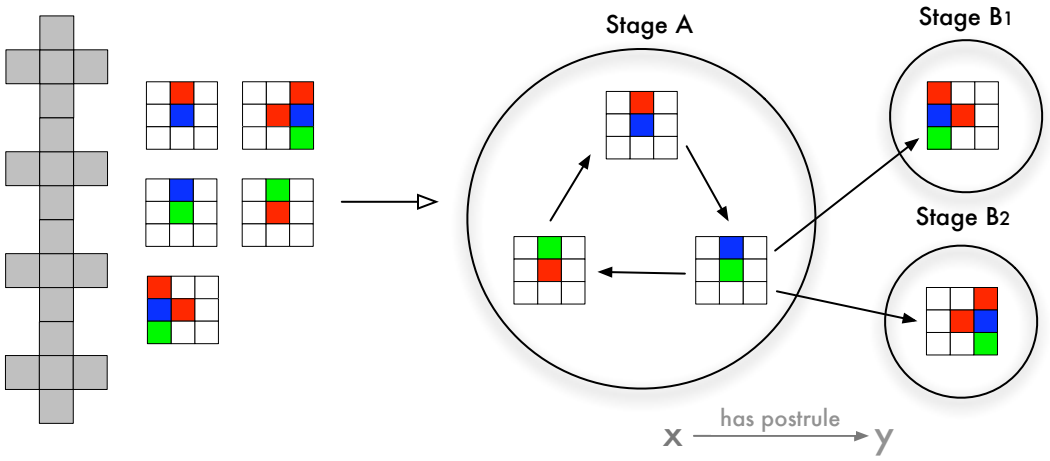


Figure 5.11: Building stages shown as relationships between rules using post-rules. Each stage consists of sequences or loops of post-rules. New stages are entered by following post-rule links outside the current loop.

Post-Rules and Building Stages

With the ability to generate post-rules, we can now begin to examine the behaviour of a rule set *without* having to use those rules in simulation and then attempting to interpret the construction behaviour *a posteriori*. We can also frame the original assertion based on 'building stages' within the more concrete framework of post-rule links.

If we are given a set of rules R, we can select any rule $r \in R$ and generate all of r's post-rules P_r. The set intersection $R \cap P_r$ gives us those rules in R that may fire directly after r during simulation. We can therefore establish post-rule *links* between r and the members of $R \cap P_r$.

This is illustrated in Figure 5.11. The overall structure is produced by 5 rules, which can be separated into three 'stages'. The first stage A constructs the central column of the structure, and will produce the line of bricks continually, independent of any other construction behaviour – this is implicitly indicated by the presence of the loop in this stage. Meanwhile, stages B_1 and B_2 are independently triggered at certain points during the construction of this column (by stage A), resulting in the spurs at either side. The rules in stages B_1 and B_2 are post-rules resulting from the expansion of meta-rules centred around cells 1 and 7 respectively in Figure 5.9.

Given a stigmergic system with possible values S, and a rule within that system r:

1. The set of pre-rules for r, $Q_r = \emptyset$.
2. For each **non-**`EMPTY` cell b in r:
   (a) Create a *meta-rule* m by extracting the local neighbourhood around b. Cells which are not present in r are set to `UNDEFINED`.
   (b) Let U_r be the set of all `UNDEFINED` cells in r, and A the set of all permutations of $|U_r|$ cell values from S.
   (c) For each combination of values $a \in A$, assign a to U_r and add this rule to Q_r.

Figure 5.12: The pre-rule generation algorithm, for generating those rules which if fired may create the environmental configuration the given rule matches.

5.4.2 Pre-Rules

In addition to producing the set of rules which could fire *after* any given rule within a stigmergic system of this type, we can also generate those rules which may have fired *immediately previous* to the rule under consideration, or more accurately, those rules which might have created the simulating configuration which matched against this rule. These **pre-rules** are simply the reverse of post-rules, working backwards in the series of building events rather than forwards. The algorithm for producing pre-rules is slightly simpler but almost identical to the algorithm for post-rules, and is shown in Figure 5.12.

Those rules which contributed to the local environment matching the current rule r must have placed a brick in this environment. Furthermore, the brick which this rule places does not exist at time $(t - 1)$. Therefore, if we remove the central brick from r, each pre-rule must have placed a brick in one of the non-`EMPTY` cells within r.

5.4.3 Meta-Rules

When determining post-rule links within an algorithm, it is not necessary to fully expand the meta-rules. Instead, the same rule matching mechanisms developed for the *Nest-3.0* system (specifically, cell matching from a set of values as described in Section 4.2.6) can be used

to match UNDEFINED cells in meta-rules against fully defined rules from the algorithm. This can represent a significant computational saving.

Meta-Pre-Rules – The number of meta-rules created is equal to the number of filled cells in the original rule, excluding the central cell, which represents the brick that will be built should this rule fire. The number of fully defined pre-rules represented by a meta-rule with c UNDEFINED cells is given by s^c, where s is the number of valid cell states (including EMPTY).

Meta-Post-Rules – The number of meta-rules created is equal to the number of empty cells remaining in the original rule during post-rule generation, The number of fully defined post-rules is given by $(s-1) \times s^c$, since we must also assign the state of the brick which is built by the rule, and this must be non-EMPTY.

For instance, if rule r in Figure 5.9 exists in a stigmergic algorithm where only a single brick type is allowed (or in other words, cells can have only two distinct values, RED and EMPTY, so $s = 2$), then the total number of post-rules of rule r, expanded from the 7 meta-rules (for each of the 7 empty cells), is given by:

$$4 \times ((2-1) \times 2^5) + 3 \times ((2-1) \times 2^3) = \underline{152}$$

Clearly, if the number of possible brick types is increased, the number of post-rules increases exponentially.

Depending on the nature of the geometry under consideration, and the position of the meta-rule in relation to the original rule, the number of UNDEFINED cells and as a consequence the number of fully defined pre- or post-rules differs significantly. This is illustrated in Figure 5.13 and Table 5.1.

5.4.4 Post-rules, Pre-rules and Excavation

While the *Nest-3.0* system described in Chapter 4 features excavation behaviour in addition to building in the original stigmergic model, until now we have based our algorithms on the

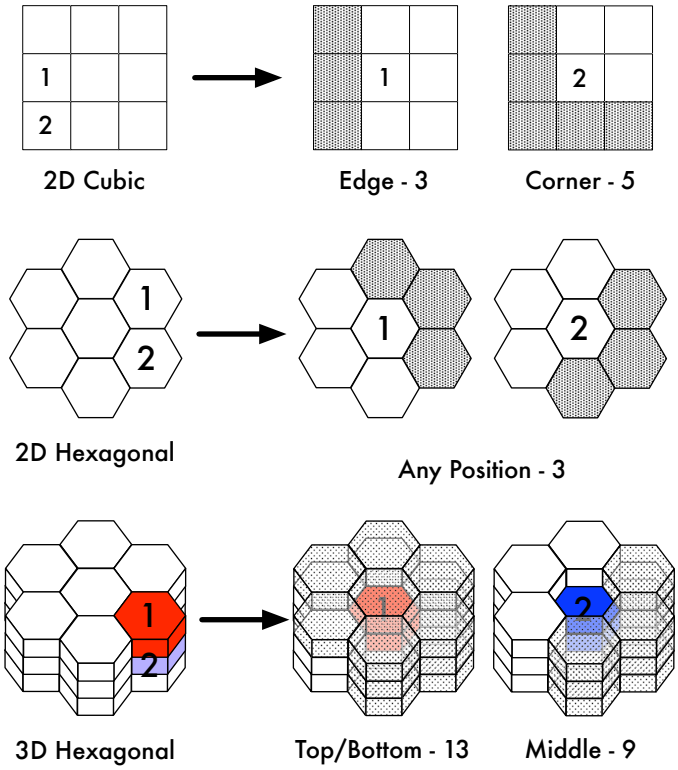


Figure 5.13: The number of `UNDEFINED` cells depends on the geometry of the rules, and the position of the meta-rule's central cell in the original rule.

Geometry	Position	Values	Post-Rules
2D Cubic	Edge	2	8
		4	192
		8	3584
	Corner	2	32
		4	3072
		8	229376
2D Hexagonal	Any	2	8
		4	192
		8	3584
3D Hexagonal	Middle	2	512
		4	786432
		8	939524096
	Top, Bottom	2	8192
		4	20136592
		8	3848290697216

Table 5.1: The number of post-rules generated from meta-rules in different locations with various geometries.

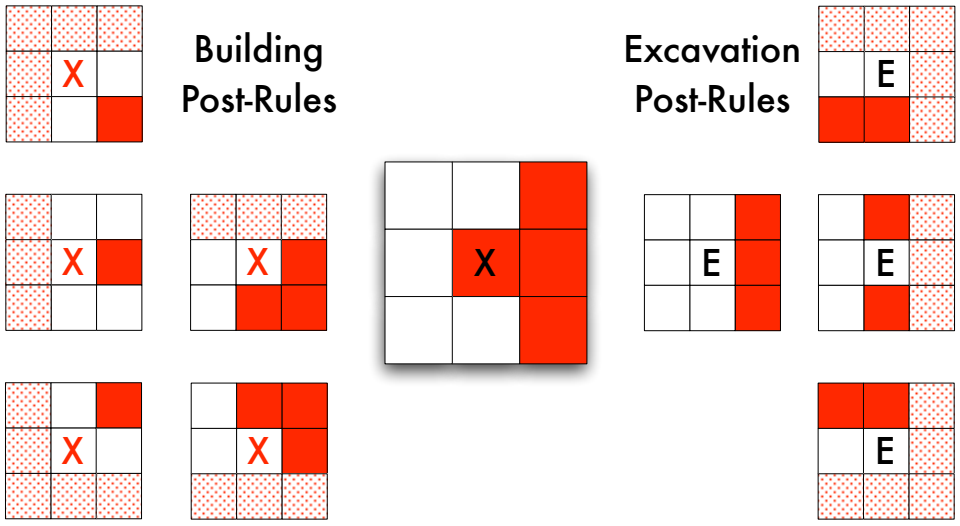


Figure 5.14: Meta-rules generated when excavation is enabled.

original system described in Section 3.1. At this point we will now consider how excavation might affect the process of post-rule generation.

Consider the rule and meta-rules illustrated in Figure 5.14. To the left we have the 'normal' post-rule meta-rules, i.e. the meta-rules representing those rules which might place a brick as a direct result of the environmental modification producing when this rule fires. If bricks can be removed during simulation, we must also consider that by placing a brick, we may have created a simulating configuration for a rule which *removes* a brick. Since the scope of our environmental modification is limited to the local neighbourhood, the candidate bricks for removal are only those bricks in our local neighbourhood.

Therefore in Figure 5.14 we can see that after the rule fires, there are 4 bricks present in the local environment. To generate the *excavation meta-rules* we select a brick in the neighbourhood as the centre cell, where instead we would select an empty cell during the regular procedure. That brick will be removed (indicated by the **E** in the build cell of the meta-rule) in the excavation rule.

When excavation is allowed, the number of meta-rules for a given rule r is equal to the number of empty cells, plus the number of filled cells within the rule – in other words, the total number of cells within the rule. The number of excavating post-rules from an excavating meta-rule with c undefined cells is given simply by s^c, where s is the number of valid states in the stigmergic system.

It should be clear now that the process and result of generating excavating post-rules is identical to that required to generate pre-rules. When including both building and excavating behaviours, the number of post-rules is given by:

$$((s-1) \times (s^c)) + s^{n-c}$$

where s is the number of valid cell values, c is the number of `EMPTY` cells in the rule, and n is the size of the neighbourhood (including the centre cell).

It is important to note that by including excavation, the space of possible stigmergic algorithms is greatly increased; the number of possible rules increases from $(s-1) \times s^{c-1}$ to s^c. This would clearly have a significant effect on the systematic exploration of possible stigmergic algorithms.

Building and Excavation – The General Case

From an abstract perspective, the obvious general case is to consider rules as defining arbitrary cell value modifications, rather than only:

- modifying a cell from empty to any other cell value (filled)
- modifying a cell from filled to empty

In a generalised environmental modification system, when a rule is matched the value defined in the centre cell is directly applied to the current location, regardless of the current state of that cell in the spatial lattice. For example, we are now free to change a cell from `RED` to `BLUE`, without having to remove a 'brick' as an interim step. This behaviour encapsulates both building and excavation in a single model.

5.4.5 Pre- and Post-rules – Summary

We now have an algorithmic means to explore the behaviour of stigmergic systems – by considering the post-rule links implicit within the rule set, we can determine likely sequences of rule-firing without being forced to simulate the system and analyse its behaviour from above.

From this new perspective we see again that 'building stages' are simply an approximate labelling of a system provided by the observer. Furthermore, the separation of rules into stages has no bearing on the behaviour of the system – it is *only* useful to the observer. Meanwhile, the dependencies highlighted by *post-rule links* define exactly how the system can behave.

5.5 Automatic Generation of Stigmergic Algorithms using Post-Rules

In this section we will now reconsider the original exploration of the space of stigmergic algorithms undertaken in [156]. We will then attempt to apply some of the practical understanding we have gained of the nature and behaviour of these abstract stigmergic rule sets, in the form of *post-rules*.

5.5.1 An Intractable Problem Space

As Bonabeau et al.[158] attempted to understand these stigmergic systems by generating a set of rules and examining the architecture they produce, they were forced to confront a space of possible stigmergic algorithms which is intimidatingly large. For a three-dimensional hexagonal simulation and 2 allowable brick colours, the number of different rules becomes $2 \times 3^{20} = 6,973,568,802$, or almost 7 billion rules. In general, the number of possible rules within a given system can be found as follows:

$$\text{number of rules} = (s-1) \times s^{c-1} \tag{5.1}$$

where s is the number of valid brick states, and c is the number of cells in a local neighbourhood: the neighbourhood consists of c cells, but one of these is the central cell, which indicates the colour of brick to build. This cell, therefore, cannot be empty, and so accounts for the $(s-1)$ term above. The remaining $(c-1)$ cells may hold any value.

Furthermore, if they limited their search to algorithms of up to 10 rules, the total number

of algorithms they are faced with is given by summing over the binomial coefficient:

$$\text{number of algorithms} = \sum_{i=1}^{m} \binom{n}{i} \equiv \sum_{i=1}^{m} \frac{n!}{i!(n-i)!} \tag{5.2}$$

where we wish to choose m elements from a set of n. In this case, n is the total number of rules, as defined above, and m is the limit of our algorithm size. As an illustration of the scale of this problem space, the number of three-dimensional hexagonal algorithms with between 1 and 10 rules is approximately 4.426×10^{96}. At a rate of one million algorithms considered per second, it would still take 1.4×10^{83} years to consider every algorithm. Furthermore, the number of algorithms vastly exceeds the number of atoms in the universe, estimated[6] between 1×10^{66} and 1×10^{79} atoms. Even maintaining a record of which algorithms have been considered poses a significant problem in the face of the memory constraints of even the largest computing resources currently available.

Clearly the space and time constraints prohibit a direct consideration of this problem; the possible number of algorithms not only grows combinatorially as the number of rules allowed in an algorithm increases, but even the exhaustive consideration of the subset of relatively small algorithms (containing only a handful of rules) is, at least for the moment, a computationally intractable task. If we wish to explore the space of stigmergic algorithms in any systematic manner, this overwhelmingly large pool of potential rules must be constrained in some manner.

5.5.2 Rule Selection in Algorithm Generation

In [156], it is noted that the vast majority of stigmergic algorithms randomly sampled did not produce interesting results. While some of these 'uninteresting' results may be architectures which did not pass the 'coherency' criteria laid down by Bonabeau and Theraulaz, the vast majority of these algorithms will have produced absolutely nothing during simulation.

Consider the two 'random' rule sets in Figure 5.15. Algorithm a will produce no building behaviour, while Algorithm b will produce a randomly-staggered line structure. In a random sampling, these two algorithms are equally likely to be selected for simulation. If we

[6] http://www.sunspot.noao.edu/sunspot/pr/answerbook/universe.html

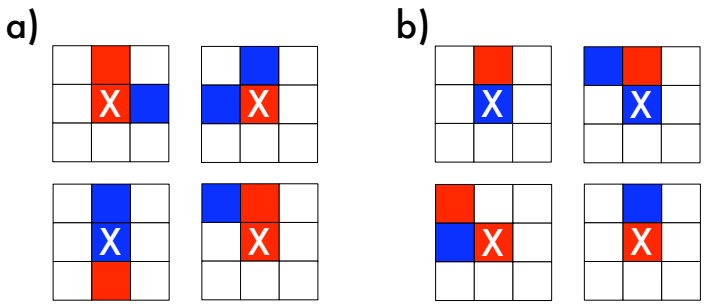


Figure 5.15: Two 'random' algorithms. Algorithm a produces no structure, placing no bricks. Algorithm b places bricks indefinitely.

constructed these algorithms by selecting m rules in sequence, each choice is made from the set of all rules (Equation 5.1), minus those rules already selected for the algorithm.

Figure 5.15 highlights the fact that selecting rules in this random manner is a very poor method for generating an algorithm. Some means of guiding the algorithm selection towards those rule sets which contain sequences of rules that will actually fire will clearly increase the value of our simulation time dramatically.

The First Rule

At the beginning of a simulation, the environment contains a single brick. Therefore, for *any* construction to proceed, there must be some rule in the algorithm which matches against this single brick. In more general terms, construction in *every* algorithm must begin with a rule that contains exactly **two** bricks – the central cell, which indicates the brick to be built, and the brick which matches that single brick in the initial environment.

The number of rules with b bricks in a system of s states and neighbourhood size in cells c is given by:

$$\text{Num. Rules with } b \text{ Bricks} = s!\binom{c}{b} \equiv \frac{s!c!}{b!(c-b)!} \tag{5.3}$$

Using this, we can see that the initial rule choice in a three-dimensional cubic system has been reduced from 47,525,504 to only 7,800 – a dramatic reduction.

It would seem intuitive that the *next* rule must match against **two** bricks, and therefore contain **three** bricks (and a corresponding reduction in choice from 47,525,503 to 62400). Unfortunately this is not the case – referring back to the single-rule systems in Figures 5.6

To create an algorithm with n rules:

1. Create the rule set for the algorithm, $R = \emptyset$
2. Insert the initial rule r_i into R.
3. Until $|R| = n$
   (a) Select a rule $r \in R$
   (b) Generate the *post-rules* of r, P_r (see Figure 5.10)
   (c) Select a post-rule $p \in (P_r - R)$, and add p to R

Figure 5.16: Generating stigmergic algorithms using post-rules.

and 5.8 we can see that at any point in the course of these simulations, three, four or more bricks may exist in the environment. In these situations it is clear that a rule which contains any number of bricks could potentially match in these simulations. Instead, we need some other mechanism of systematically determining which rule could fire next during simulation.

5.5.3 Post-Rules and Algorithm Generation

We have seen earlier in this chapter how the identification of *post-rules* allows the observer to determine, without simulation, which rules from an existing algorithm can fire in sequence during a simulation – the *post-rule links* between existing rules. However, we can also attempt to apply this implicit constraint information to the generation of random algorithms, resulting in simulations which are far more likely to produce *something* rather than *nothing*.

The algorithm presented in Figure 5.16 demonstrates a simple method to generate a stigmergic algorithm of any given size.

At each point during rule selection, we consider which rules could fire next during simulation by generation a set of post-rules and adding the next rule from within that set.

Should we wish to, we can define 'building stages' within this process, by selecting the next post-rule p in the algorithm above such that it is only the post-rule of a single-rule already within the algorithm. The slight modification to the algorithm in Figure 5.17 shows how to create a new building stage using post-rule information.

An example of an algorithm derived in this manner is presented in Figure 5.18. From

3. Until $|R| = n$
   (a) Select a rule $r \in R$ to trigger the next stage
   (b) Set the set of *all* post-rules $P_R = \cup_{q \in R} P_q$ where $q \neq r$
   (c) select the next rule $p \in P_r - P_R$ and add p to R

Figure 5.17: Adding a new stage using post-rules.

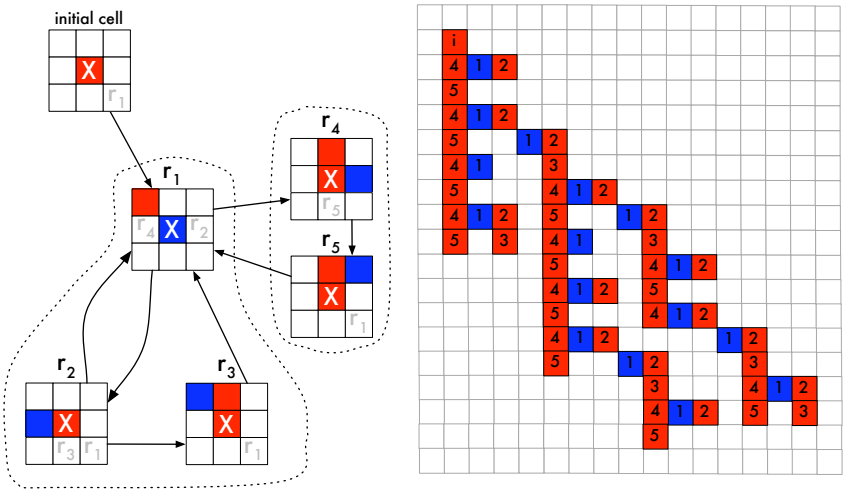


Figure 5.18: A stigmergic algorithm generated using post-rule information. In the constructed architecture, bricks are labelled with the rule *id* which placed the brick.

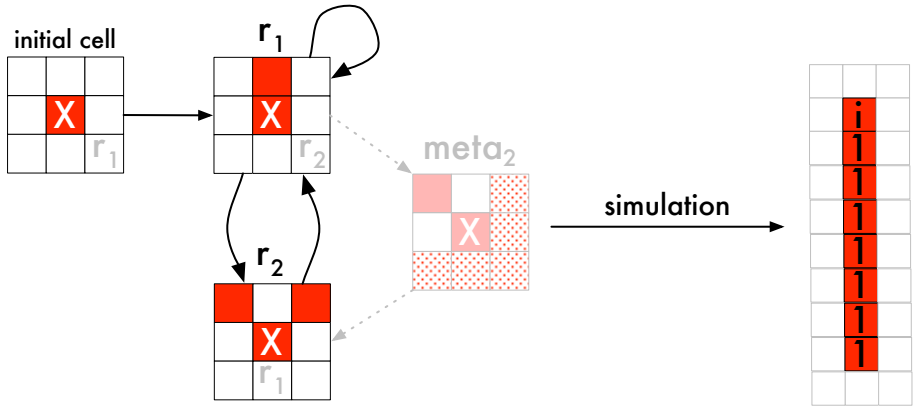


Figure 5.19: A second stigmergic algorithm generated using post-rules. During simulation, one of the post-rules never fires.

the initial cell, a single post-rule r_1 is selected and added to the algorithm. From there, a post-rule of r_1 is generated – r_2 – and then subsequently r_3 is added. These three rules could be considered a 'building stage'. To start a new stage, rule r_4 is added as the post-rule to *only* r_1 and *no other rules* in the algorithm thus far. Finally, rule r_5 is added to this second 'stage'.

The results of a simulation using this algorithm are also shown in Figure 5.18. The construction is quite erratic; to the observer it is obvious that there may be elements of structure within the resulting architecture, but the architecture is not completely regular. Repeated simulations produce similar, 'incoherent' results.

We have generated an algorithm which satisfies all the criteria presented in [158, 156, 19] describing a 'coordinated algorithm', being:

> "a finite number of building steps, with the necessary condition that the local configurations that are created by a given stage and trigger building actions differ from those created by a previous or a forthcoming building stage so as to avoid the deorganization of the building activity."[158]

but which does *not* appear to produce a 'coherent' structure, at least according to the definitions based on intuition provided by Bonabeau et al.

Another simple post-rule generated algorithm is shown in Figure 5.19. The derivation of each of the rules is outlined within the diagram in place with the final algorithm itself.

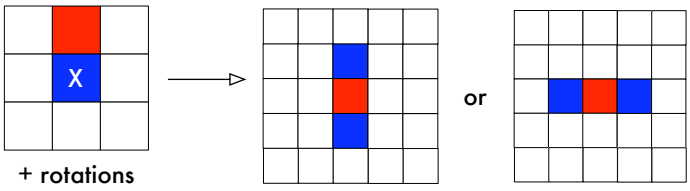


Figure 5.20: A single-rule stigmergic system that does not built continually, but instead produces one of two possible 3-brick architectures.

The results of simulation of this algorithm are also shown. We can see that while rule r_1 fires repeatedly, rule r_2 does not fire once during the entire simulation. This simple system brings to our attention a more important aspect of the application of post-rule techniques to algorithm generation, as will be discussed in the following section.

5.5.4 Single-Rule Systems and Post-Rule Uncertainty

The simple single-rule stigmergic algorithms shown in Figure 5.8 are all examples of **self-activating** rules – rules whose post-rule set includes themselves, or more formally those rules r where $r \in P_r$. This is clearly exhibited in the simulated behaviour of the systems, which all build structures without limit, despite their apparently simplicity.

In fact, only single-rule stigmergic systems where the rule contains *only a single brick* will produce any structure given the initial environment specified in Section 3.1. If the initial environment contains only a single brick, then the rule which matches this must match against one and only one brick. Only single-rule systems in which the brick built is the same colour as the brick matched build continually: as shown in Figure 5.20, with a rule which builds an alternate brick type, and with rotation enabled, two limited structures are possible. The rule in this figure is only self-activating when rotated rules are allowed to match. If no rotations were valid in this system, the rule would only match once, as in Figure 5.7. By allowing rotations, this rule can match itself when rotated 180 ° around the centre cell, and thus fire a second time.

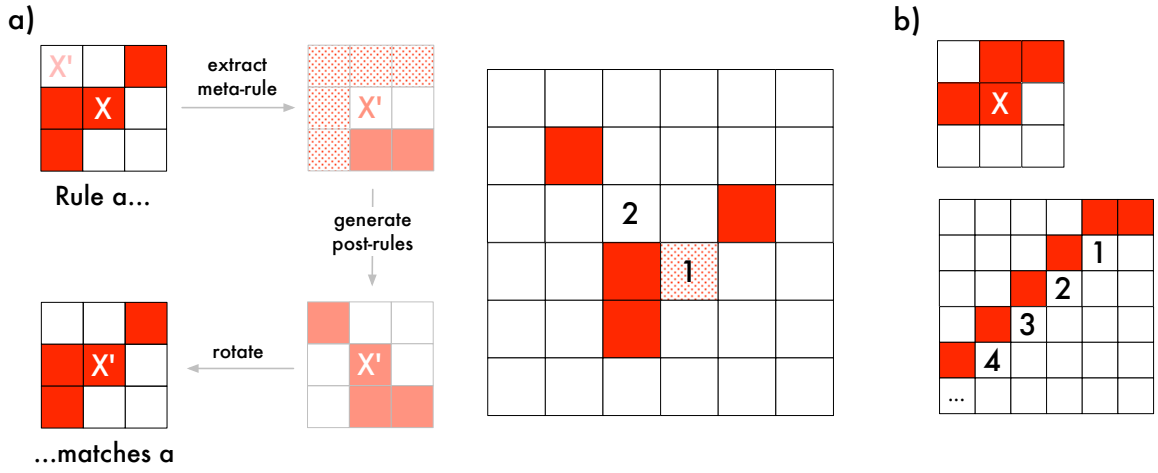


Figure 5.21: Two examples of rules which are post-rules of themselves, but require specific environmental configurations to trigger.

Post-Rule Uncertainty

Identifying a rule as being self-activating does **not guarantee** that it will be able to fire at time $t+1$ if it also fired at time t. When considering the rules presented in Figure 5.21, it becomes clear that post-rules are only an indication of which rules **could** fire in the next simulation cycle. Rule a in Figure 5.21 is clearly self-activating; by extracting a meta-rule and subsequently producing one possible rotated post-rule, it is clearly the case that $a \in P_a$. However, if a stigmergic algorithm consisting of only rule a is simulated, no building activity will occur. Instead, a specific environment is required (as shown in the figure) before rule a can enable itself to fire again at time $t+1$. Only when the rule matches in location 1 will a stimulating configuration be produced around cell 2 that also matches a rotation of rule a.

An even simpler example is presented as system b) in Figure 5.21, in which no rotation is required. Only in the presence of an existing diagonal line of bricks, will the rule in this system cause itself to fire repeatedly, at least until the end of the diagonal line of bricks is reached.

From this, in conjunction with the simple automatically generated system shown in Figure 5.19, we can now see clearly that the post-rules of any rule are those rules which *could* fire directly after this rule, given the correct external circumstances. They do not indicate which rules *will* certainly fire. Therefore using post-rules can only be used as a rough guide for the

selection of rules to be added to a stigmergic algorithm, and an algorithm generated in this manner cannot be guaranteed to produce any structure at all.

5.5.5 The Real Benefit of Post-Rule Selection

Since selection based on the post-rules of existing rules within the algorithm cannot guarantee that each rule added will be active (and therefore worthwhile) during simulation, what can be said about the benefits of post-rule based selection over the simpler, random choice?

It should be clear that the probability that a post-rule p_r will fire after rule r is always equal or greater than the probability of a randomly selected rule firing. Let R be the set of all rules within a system. If rule r fires at time t, let the set of all rules which actually can fire at time $t+1$ be N, and P_r is the set of all post-rules of r, such that $N \subset P_r \subset R$, and therefore $|N| < |P_r| < |R|$. It is then clear that if we select the next rule n, $P(n \text{ from } P_r) = |N|/|P_r|$ and $P(n \text{ from } R) = |N|/|R|$. Since $P_r \subset R$, we know that $|P_r| \leq |R|$, so $|N|/|P_r| \geq |N|/|R|$.

However, what is not clear is just how much of a reduction in choice post-rule based selection affords. Since each rule is unique, and the number of post-rules depends to a large extent on the layout of bricks and empty cells within that rule, it is not feasible to make a precise determination of the exact size of P_r for any rule. An approximation may be determined as follows.

Given a two-dimensional, cubic system, with two possible brick colours (and therefore three possible cell values), the total number of rules in the system, $|R|$, is given by Equation 5.1 as follows:

$$(s-1) \times s^{c-1} = (3-1) \times 3^{9-1} = 13122$$

If we imagine an 'average' two-dimensional cubic rule, of the eight cells in the neighbourhood which aren't the build cell half (four) may be filled, and the remaining four be empty. Of those four cells, two might be placed on the edge of the rule, and the remaining two on corners (see Figure 5.13). We can therefor estimate that the total number of postrules for our 'average' rule, $|P_r|$ is given by:

$$2 \times (s-1) \times 5^s + 2 \times (s-1) \times 3^s = 4 \times 5^3 + 4 \times 3^3 = 608$$

There is clearly a significant reduction, especially when it must be noted that the total of 608 certainly contains duplicated rules as a result of the 'expansion' of the meta-rules during post-rule production.

However, this method of estimation becomes less reliable as the number of states available in a system increases. We have also idealised the characteristics of the 'average' rule within a system: initially the number of post-rules will be significantly higher, since a larger number of empty cells will be present within the rule. As the rules present in the algorithm contain more bricks, the number of possible post-rules decreases according to the number and location of empty cells. Even if we assume that this estimate of post-rule numbers is accurate, the total number of algorithms of between 1 and 10 rules is:

$$\sum_{i=1}^{m} \binom{608}{i} = 1.796 \times 10^{21}$$

This naive estimate indicates that even using post-rules, the number of possible algorithms is still too large for us to exhaustively consider using the computational tools currently available.

5.6 Automatic Algorithm Generation – Summary

In this section the original approaches adopted by Bonabeau et al.[158, 156, 22, 19] to search for 'interesting' structures were considered, along with the assertions they made regarding what characteristics a stigmergic algorithm must have to produce such structures. The notions of 'coordinated algorithm', 'building stages' and 'coherent structures' have been examined and their various weaknesses exposed.

The concept of post-rules (along with pre-rules and meta-rules) was introduced as a means to avoid some of the ambiguity present in those concepts, and as a better means for automatically generated stigmergic algorithms for the systematic exploration of the space of possible rule sets. However, despite the promise post-rules would appear to hold for predicting algorithm behaviour, they can only indicate which rules *could* fire next, instead of which rules *will* fire next during simulation. Some rough numerical calculations indicate

that while the size of of the problem space has been hugely reduced, it is still computationally infeasible to systematically explore the entire space even for simple systems, as a means for examining the potential behaviour of stigmergic systems.

In the remainder of this thesis an alternative, more practical and potentially much more rewarding approach to the automatic generation of stigmergic systems will be explored.

Chapter 6

Automatic Algorithm Extraction

Overview

As has been suggested by the original *Nest-2.11.1* work and explicitly shown in previous chapters, systematic consideration and generation of coordinated stigmergic algorithms presents significant problems in terms of computational tractability. Such an investigation is directed at addressing the question: *what is the range of structural features that can be generated using stigmergic algorithms?* Like many such fundamental questions, despite a clearer understanding of the nature of the problem, in this case provided by examining the potential causal relationships between rules via *post-rules*, such a comprehensive understanding of the system as a whole remains beyond our grasp.

However, the lessons learned from the consideration of this question have applications beyond the problem they were designed to address. A "bottom-up" comprehensive understanding is most useful when it can be used to answer specific questions about individual situations. Thus, perhaps a more important question to address is *given a specific problem, how can the principles of stigmergic construction be applied to solve it?* The problem in this case is the construction of arbitrary structures by simple agents sharing a single, stigmergic rule-set. As noted in [156]:

> "The space of local configurations that is likely to be encountered by a given agent is rather huge... and the space of local rules cannot be explored system-

> atically... One has to discover the minimum set of rules necessary to produce a given architecture in a more clever way."

Is it possible to automatically extract a stigmergic algorithm from an existing structure? In following chapters, the feasibility of such a process will be examined, and one solution to this problem presented.

6.1 The 'Holy Grail'

Research using stigmergic and swarm techniques is primarily concerned with applying behaviours seen in nature to artificial problems. What is crucially important in this process is finding suitable problems for the biologically-inspired approach. For instance, one of the most successful examples of biologically-inspired problem solving is the application of ant foraging and path-finding techniques to routing problems in the internet and telecommunications domains[51, 69, 17, 20]. The parallels between each of these domains are quite clear, each requiring the production of shortest-paths with a large degree of robustness.

However, the application of other biologically-inspired models is not so clear. Certainly the true range of real-world applications of these solutions is far less obvious. Here we are referring to swarm behaviours such as sorting[46, 12], collecting[149, 34, 54], task allocation[21] and construction[158, 156, 17, 29, 95, 94, 115].

6.1.1 From Modelling to Manufacturing: Applied Stigmergy

The experiments discussed previously [156, 19] clearly show that the model of stigmergic construction used here is capable of producing interesting structures, some of which do appear very similar to those found in nature. While this is a result in itself, it is still of little more than curiosity value. For this model to become useful in itself, we must determine if we can apply these stigmergic construction techniques to *real-world* problems: while we rarely have need to build structures which resemble the nests of social insects, we do have significant construction needs, and the possibility of using stigmergic systems to fulfil these

needs is of course, very attractive[1].

Thus far we have considered the following questions:

Is the model capable of producing structures? Clearly the answer is yes, based on the architectures presented in previous work, along with our own experimentation.

Is the model capable of producing *interesting* structures? Despite the subjectivity of this question, it would be difficult to dispute the merits of those structures which do indeed appear to mirror forms commonly seen in nature. Clearly not only does the model produce *something*, it can, under the right circumstances, produce something *potentially desirable*.

What structures *can* or *cannot* be produced using this model? Can we produce arbitrary architectures using stigmergic techniques? A systematic exploration of this model is, of course, impossible, but we have begun to understand how the system will behave in the presence of certain types of rule.

Unfortunately, beyond a limited subset of simple algorithms, the tools we have developed so far to explain and understand this model are not powerful enough to predict the behaviour of all the stigmergic algorithms we might wish to consider.

Finally, we must consider what might be regarded as the 'Holy Grail' at the end this line of research – **how can we discover stigmergic algorithms which produce *specific, arbitrary* structures?** In answering this question we will have taken the most significant step towards practical application of stigmergic construction as a means of manufacturing structures which are specified by our needs as designers, engineers and ultimately the users of the products created.

6.2 Simple Stigmergic Algorithm Extraction

In order to begin investigating the possibility of an algorithm extraction process, we first consider a completed architecture, shown in Figure 6.1. This structure has been produced

[1]See 1.2.2 for the motivations behind using emergent systems

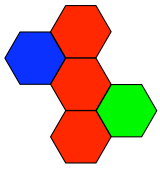


Figure 6.1: A simple architecture, as produced by the *Nest-3.0* system.

previously by the a *Nest* system, and we are required to derive the rules used during its creation.

6.2.1 Ordering and Rule Extraction

By considering the order in which bricks were placed, we can derive the exact structure of the rules which placed those bricks[2]. During each simulation cycle, an agent considers the arrangement of bricks surrounding its current position, and if a match is found within the set of rules, a brick is placed. This process is described fully in Section 4.7.

At each simulation cycle, we know the current simulation 'time', t. If when the bricks are placed each brick is tagged the current value of t, when the structure is completed each brick will be labelled with its order, n, within the overall construction process. This can be seen in Figure 6.2.

Since we now know the order in which the bricks were placed, we can 'rewind' and 'replay' the construction at will, simply by removing and adding bricks from the architecture in the order specified by the brick tags. To see the the state of construction at time t, we can simply remove all the bricks whose order tag $n \leq t$.

If we select a single brick b with order n_b within the structure, and create a copy of its local neighbourhood including only those bricks where $n < n_b$, this resulting neighbourhood must match exactly the rule R_b which placed b during the simulation.

This point is worth making very clear: all of the cells (both filled and empty) surrounding the location where brick b was placed must have matched exactly for the rule R_b which placed b to have fired. Because we know that R_b did indeed fire, we can extract the exact structure

[2]It should be noted that in this abstract system, bricks can never be placed truly *simultaneously*, since each agent is given the opportunity to examine its surroundings and build a brick before any other agent may act. Truly simultaneous action is beyond the scope of this simple abstract system.

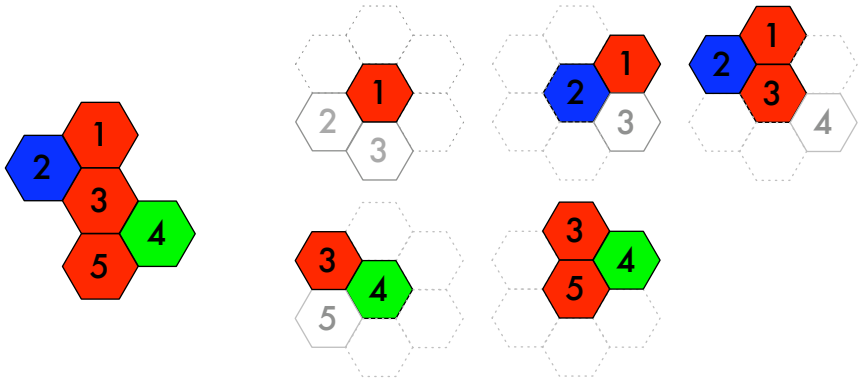


Figure 6.2: Using brick ordering to extract rule structure from a pre-generated architecture.

```
rules = [] # a new empty array
architecture.eachBrick { |brick|

  # create a new rule centred on this brick and including
  # its entire neighbourhood
  r = Rule.new(brick)

  r.eachBrick { |ruleBrick|
    # if any brick within this rule is ordered after the central 'build'
    # brick, remove it from the rule
    if (ruleBrick.order > r.buildBrick.order)
     r.deleteBrick(ruleBrick)
    end
  }

  # add the new rule to the set of all rules
  rules << r
}
```

Figure 6.3: Ruby code for creating a set of rules from an architecture in which each brick has been assigned an order.

of R_b by considering the state of the architecture at that location immediately before b was placed. Thus all bricks with an order $n < n_b$, and *no others*, must be present in R_b. Within the *Nest-3.0* system, this operation is performed by the code shown in Figure 6.3.

6.2.2 Brick Colours

Given that each rule will already match only certain configurations of brick colour, and will build a brick of a specific colour, the cell value information for each rule is already encoded explicitly within the final architecture. As the rules are extracted according to

brick orderings, the original states for rule's build cell and neighbourhood brick states are thus present in each rule.

This process is shown clearly in Figure 6.2. For each brick, we create a copy of the neighbourhood, including only those bricks whose order is less than the order of the brick under consideration. For each brick, we produce a rule which matches *exactly* the local configuration required to place the brick at that point during construction.

6.2.3 Simple Algorithm Extraction – Summary

Despite the obvious simplicity of this example, it is clear that we can move from completed architecture to the corresponding rule-set, and in this case with relative ease. Following the instructions given above, any similarly-tagged architecture can be deconstructed successfully. Where two bricks were placed by the same rule, two identical rules will be extracted. In this case we follow the conventions set out in the *Nest-2.11.1* software, and remove any duplicate rules from the algorithm (since the result of either firing will be identical).

6.3 Simple Algorithm Extraction Assumptions

The operation above is sufficient to successfully extract a stigmergic algorithm from a structure, given the following assumptions:

Ordering The order tagging process exists, was performed correctly during simulation, and has not been altered in any way.

Brick State The colour of each brick within the architecture is unchanged from the brick placed during construction.

That it to say: *if we know the order in which the bricks were placed, and the colours of those bricks, we can easily extract the stigmergic algorithm which constructed the architecture under consideration.* But what if these cannot be guaranteed?

Any modification to the order will necessarily alter the structure of the rules extracted and produce a set of rules which does not correspond to those which originally built the

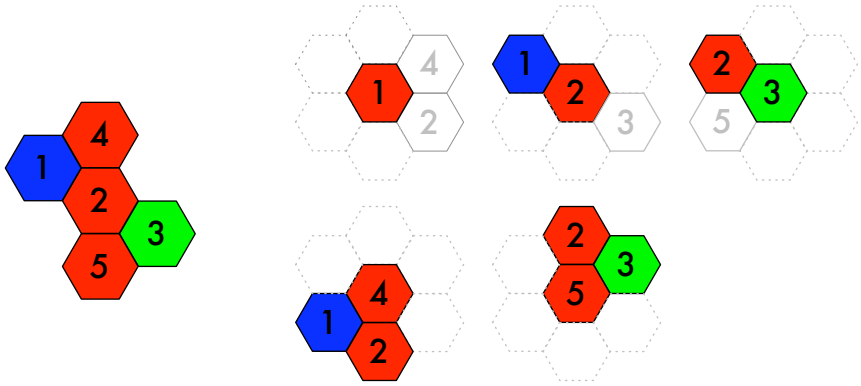


Figure 6.4: An alternative ordering on the bricks from the simple structure shown in Figure 6.2 produces different rules.

structure. This can be clearly seen in Figure 6.4, where a different ordering has been applied to the architecture from Figure 6.1. Due to the small size of this architecture, the space of possible rules is quite constrained, and only a single rule differs in this case. Larger architectures will provide greater scope for deviation.

Similarly, the states of the bricks in the structure directly (and obviously) correspond to the brick states built by each rule. If these are modified after construction, but before rule extraction, the resulting rules cannot correspond to those used during the simulation.

6.3.1 The Assumption of Coordination

The following assumption is also implicitly present if any undertaking of rule extraction is to be successful:

Existing Coordination The stigmergic algorithm which generated the structure is known to reliably produce the same structure each time it is used.

If we were to submit a properly ordered maze structure as shown in Figure 5.4, we would certainly successfully extract the structure of the rules used in those simulations[3], but as demonstrated in that figure, subsequent simulation using the same stigmergic algorithm might produce a radically different structure.

[3]Actually distinct rotated versions would be produced, since the specific simulations shown in Figure 5.4 allowed rotations. In other words, where two rules were allowed with four valid rotations, up to eight rules might be extracted.

The usefulness of such an extracted algorithm is directly tied to the perceived merits of this type of architecture. Just as was demonstrated using Figure 5.4, such structures may exhibit constant properties in terms of density ratios or particular structural features, and such properties might be desirable regardless of the exact form of the resulting architecture.

However, we have stated that our goal is "to discover the minimum set of rules necessary to produce a given architecture", rather than a given architectural feature (something much harder to identify), and so we will be attempting to reproduce the input structure *exactly*.

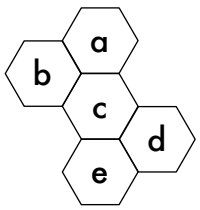
6.4 Practical Algorithm Extraction: A First Attempt

Clearly the extraction process as it currently stands falls some way short of being useful; it relies on the previous existence of a stigmergic algorithm to do most of the conflict management already. Furthermore, most of the assumptions we have relied on thus far become invalidated if we do not presume that the structure we are presented with was originally the product of a stigmergic simulation: no ordering will have been applied if the architecture was not produced during simulation, and brick colours may be modified without our knowledge. Despite these large assumptions this process will be the basis for our automatic algorithm extraction approach.

Since the act of matching rules during simulation implicitly defines a brick ordering, by determining a brick ordering *a posteriori* we are implicitly defining the structure of the rules. Similarly, the presence of differing brick states serves to manage the firing of conflicting rules – rules which match against identical arrangements of bricks should use alternative brick states to prevent the firing of rules where it would cause deviations from the desired output architecture.

From this point we no longer assume that out input architecture is the product of an existing stigmergic algorithm. As a result of this, the bricks carry neither the order tags nor the state information that could only have been created as this architecture was built during such a simulation. Effectively, we must operate on architectures with only information regarding brick positions, without any knowledge of ordering, or brick colour.

In this section we will consider the simplest method for extracting a reliable stigmergic



1) a–b–c–d–e
2) c–a–b–e–d
3) e–c–a–b–d

4) a–e–b–c–d

Figure 6.5: Random brick ordering for a simple 5-brick architecture. Orderings 1, 2 and 3 are valid while ordering 4 is invalid for the given labelling/architecture.

algorithm from a such an architecture.

A Note Regarding Brick Colours and Architectures

In this investigation, as in the original *Nest* experiments, brick colours are only present as a stigmergic aid, and do not represent any desired final aesthetic. Therefore, two architectures which are structurally identical – that is, the relative arrangements of bricks are the same – are considered to be identical, regardless of the colours of individual bricks.

6.4.1 Ordering

The simplest method of ordering the bricks with an architecture is to simply assign an ordering at random. The number of possible brick orderings for an architecture of size N is given by $N!$, so for example the number of ordered brick sequences for a 10-brick architecture is $10! = 3628800$. Being factorial, for larger architectures the number of brick orderings rapidly increases as the architecture size grows ($20! \approx 2.433 \times 10^{18}$). However, not all permutations of bricks can be considered valid for the purposes of rule extraction.

Invalid Brick Ordering

In the *Nest* simulation systems, bricks can only be placed in direct contact with either a face or an edge of an existing brick (aside from the first brick placed, which is a special case). However, following order 4 in Figure 6.5, brick e is placed before either b or c, the only cells which are adjacent to it in the final architecture. In this ordering, brick e is an *orphan* brick.

It is assumed in *Nest* simulations that the initial environment contains a single brick. This single brick corresponds with the rule extracted for brick 1 in Figures 6.1, 6.4 and 6.6.

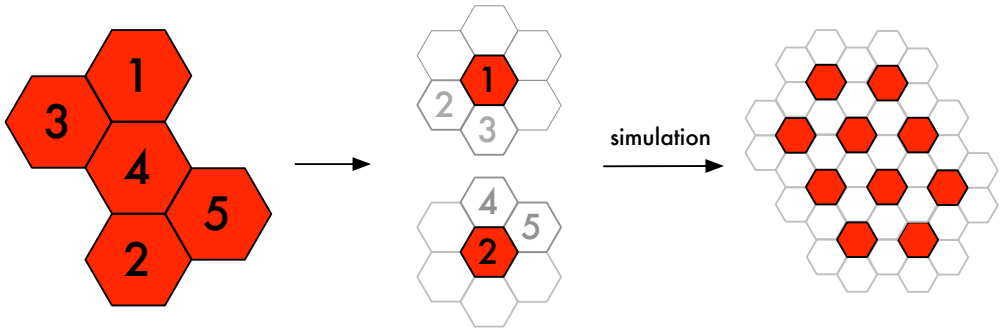


Figure 6.6: Two single-brick rules are produced with the random brick ordering of the simple architecture. The resulting simulation features many unconnected orphan bricks.

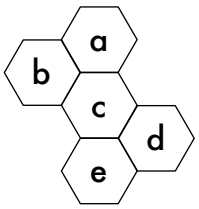
During simulation, this "starting" rule is typically disabled, because it can fire anywhere in the lattice where there is no adjacent brick. Instead, it is assumed that this rule fired at time $t = 0$, and produced the single brick present in the initial environment.

If rules of this form (which match against *empty* neighbourhoods) are present and active during the simulation, they could fire at almost any location within the empty lattice. Because there are no structural features (in the form of bricks of specific colours within the neighbourhood) to constrain firing of this rule, its behaviour is *unmanageable*. Figure 6.6 demonstrates a random order which produces two single-brick rules rather than one.

Generating Valid Brick Orderings

To avoid this, ordering the bricks within an architecture must take into account their structural relationship. The simplest way to produce a *valid* random ordering is to select the first brick at random and then construct a random tree from the graph formed by the directional links (i.e. the adjacent faces) between bricks, ordering the bricks as they are added to this tree.

This is illustrated in Figure 6.7: brick b is selected randomly as the initial brick. From b we can reach either a or c, giving us our initial frontier. We can select any brick from the frontier as the next brick in our ordering. After choosing a, our frontier remains the same as there are no bricks connected to a that were not already present in the frontier. Therefore, c must be the third brick in our ordering. Because both d and e are adjacent to c, they can



Order	Frontier
b	a, c
b, a	c
b, a, c	d, e
b, a, c, e	d
b, a, c, e, d	$\emptyset$

Figure 6.7: Producing a random valid order by performing a random walk over the graph of bricks.

be added to the frontier, and selection continues until the frontier is empty.

At each step we select the next brick in the order from the search frontier (the set of bricks which have yet to be visited but which are adjacent to bricks we have previously ordered). In this manner, a *valid* ordering and thus a valid set of rules can reliably be generated. The Ruby code which produces these orderings is shown in Figure 6.8.

6.4.2 State Assignment

As stated in Section 6.3, once the bricks have been ordered the structure of the rules can be determined by a very simple extraction process. Given our revised assumptions, all bricks within the structure are assumed to have a single, uniform value, and therefore all of the rules we will extract will build bricks of the same colour. We can see the results of applying the ordering from Figure 6.7 to our simple architecture in Figure 6.9.

At this point, any undesired *post-rules* (see Section 5.4.1) which may fire out of sequence must be managed. Since we are now experienced at identifying simple post-rules, we can clearly see that rules 2 and 4 in Figure 6.9 are *self-activating* (see Section 5.5.4), given the uniform brick colours present in the rule.

Futhermore, rule 4 is a post-rule of rule 1 (the initial brick), rule 2 is a post-rule of rule 5, rule 3 is a post-rule of rule 5 and vice-versa. Each of these conflicts arises purely from the arrangement of bricks in the rule neighbourhoods, and if we are to succeed in precisely replicating the original structure, each one of these undesired post-rules must be managed such that during simulation the rules will fire in proper sequence and construction will not continue once the final architecture has been produced.

This is most simply achieved by assigning each brick a *unique colour*. This guarantees

```
# choose a random start cell
startcell = architecture.brickArray.randomElement

orderArray = [startcell] # create a new array with this single element
frontierArray = [startcell] # create the frontier array

currentOrder = 0

# start a random walk
until frontierArray.empty? do
  # select a random element from the frontier
  tmpCell = frontier.randomElement

  # get an array holding all the neighbouring brick from this brick
  # remove all elements from this array which are empty cells, or
  # which are bricks that are already present in the order
  neighboursArray = tmpCell.neighbourhoodArray
  neighboursArray.delete_if { |cell|
    (cell == nil) or (orderArray.include?(cell)) or (cell.isEmpty?)
  }

  if neighboursArray.empty?
    # if there are no neighbouring bricks which are currently
    # unordered, remove this cell from the frontier
    frontierArray.delete(tmpCell)

  else
    # otherwise, pick a random neighbour and add it to the
    # array of ordered cells, and to the frontier and set the
    # order of this brick, incrementing the counter
    nextCell = neighboursArray.randomElement
    nextCell.order = (currentOrder += 1)
    orderArray << nextCell
    frontierArray << nextCell
  end

end # (of random walk)
```

Figure 6.8: The algorithm for assigning a random, valid ordering to an architecture

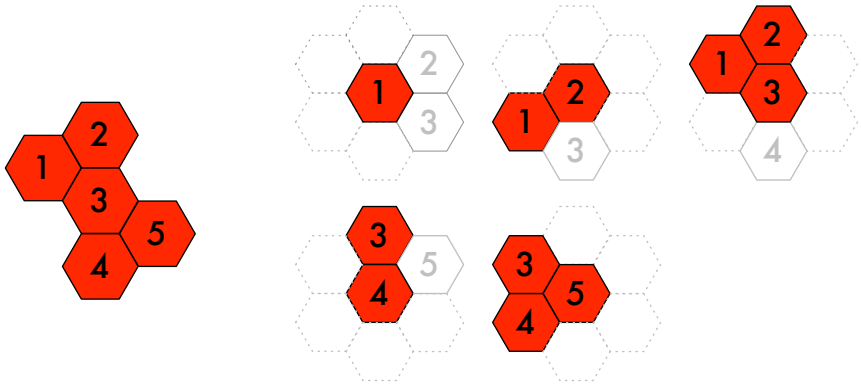


Figure 6.9: The random ordering from Figure 6.7 is applied to the simple architecture from Figure 6.1 (minus brick colour information), and rules are extracted.

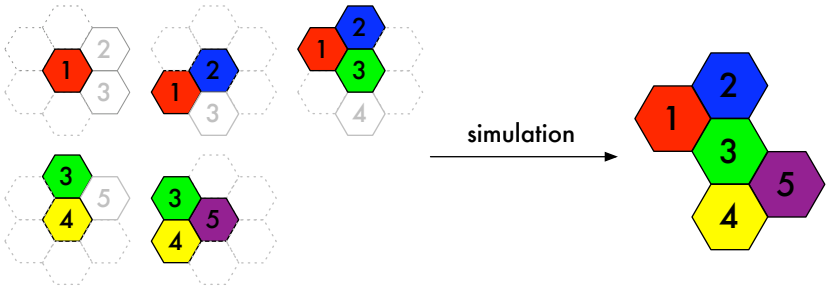


Figure 6.10: Applying a unique colour to all bricks removes all post-rule conflicts and self-activating rules from the stigmergic algorithm. The simulated architecture matches exactly the input architecture specified in Figure 6.1.

that every rule matches a unique neighbourhood, and therefore can only fire at one specific point during the construction of the architecture. Figure 6.10 illustrates this, and it can be clearly seen that all post-rule conflicts have been removed. Finally, a simulation using this extracted rule-set does indeed produce exactly the architecture given as input to our system.

6.5 Simple Stigmergic Script Extraction: Summary and Evaluation

The simplest method of extracting a stigmergic script from an architecture has been outlined above. The two keys steps in this process are

1. **Assign a random *valid* order to the bricks within the architecture**

2. **Assign a unique colour to each brick within the architecture to avoid any**

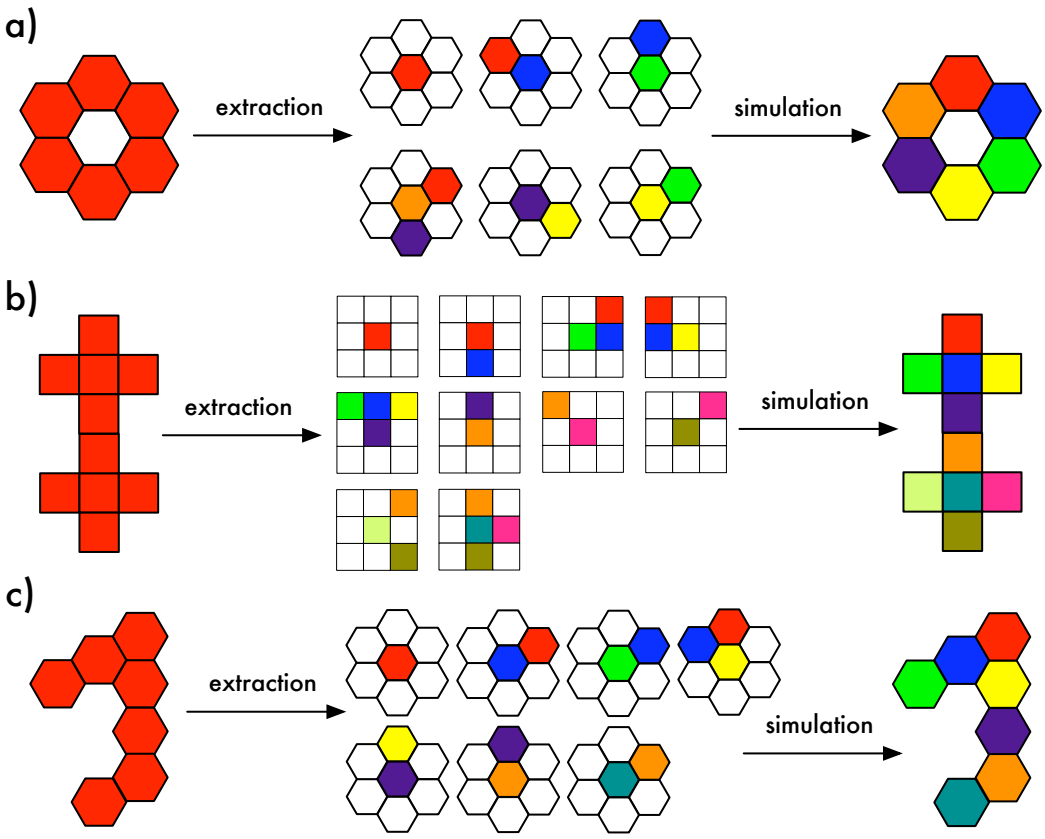


Figure 6.11: Some examples of architectures to which rule extraction has been applied.

potential rule conflicts during simulation.

The two steps of ordering and state assignment are sufficient to produce a set of rules which *reliably* reproduce the architecture supplied to the process. Most importantly, this process shows that **any architecture can be decomposed into a set of stigmergic rules**. This is a very important point if any further progress in the development of automatic extraction methods is to be successful: *no* architecture exists, representable within the limits of the *Nest-3.0* system, that cannot be decomposed using this technique. Some examples are shown in Figure 6.11.

These processes are almost independent, but it can be seen in Figure 6.12 below that the ordering has a significant effect upon brick state assignment. Because brick ordering defines the structure of the rules within the algorithm, a different number of colours may be required depending on the relationships between those rules.

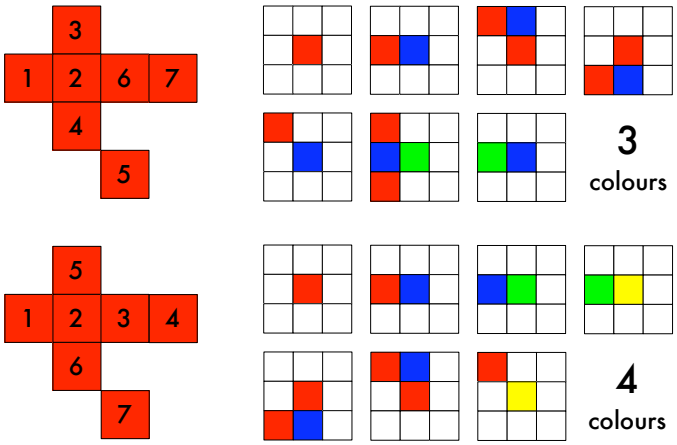


Figure 6.12: Differing brick orderings produce different minimal orderings.

6.5.1 Properties of Extracted Stigmergic Algorithms

If this process is used, several properties of the resulting stigmergic script can be predicted:

Number of Rules – The number of rules in the script is exactly equal the number of bricks in the architecture.

Number of Unique Brick Colours – The total number of brick colours in the script is exactly equal to the number of bricks in the architecture.

While we can now guarantee valid, working output for any architecture, the quality of this output is hardly comparable to those 'biological'-like scripts from which this entire system was inspired. If we intend to "discover the minimum set of rules necessary to produce a given architecture"[156] then we have fallen short: the algorithms in Figure 6.11 produced by this simple technique are certainly not minimal. As can be seen in Figure 6.13, it is possible to devise a stigmergic algorithm which builds a perfect ring of six hexagonal bricks using *less* than six states. The algorithm shown as a^1 demonstrates a reduction of states required by that particular rule-set based on a hand-calculated consideration of the interaction between the rules, and algorithm a^2 shows the optimal algorithm, derived by hand. Both of these algorithms represent improvements over our initial attempt.

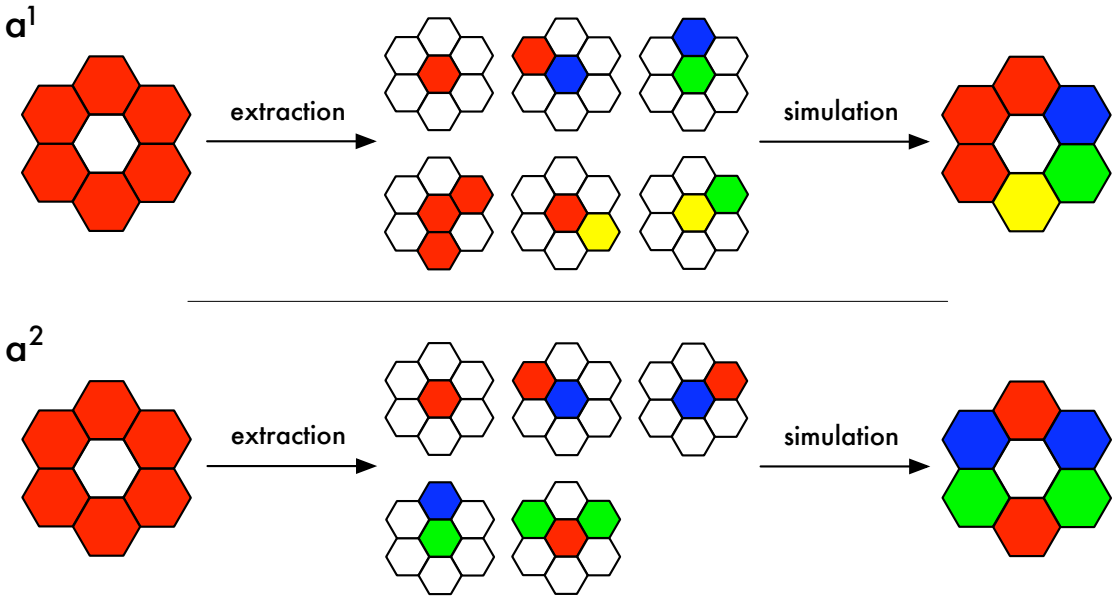


Figure 6.13: A ring-building algorithm which uses only four brick colours, rather than the six required by algorithm **a)** in Figure 6.11, and a manually-derived optimal algorithm for the same structure requiring only five rules and three brick colours.

6.5.2 Quality of Extracted Algorithms

Using this initial automatic algorithm extraction technique, the set of rules extracted precisely maps to a step-by-step plan for the construction of the entire structure, a feature which appears to be incompatible with the 'spirit' of stigmergy; there is no global blueprint present within ants and bees during the construction of their nest architectures.

On the other hand, this mechanism provides a single guarantee which does not seem to be present within the biological examples of stigmergy: the desired architecture is reproduced with *total accuracy*. Reliable production of specific architectural features is certainly an important consideration if stigmergic techniques are to be successfully applied to complex tasks.

Can the complexity (in terms of either numbers of rules, numbers of brick values, or both) of the extracted stigmergic script be reduced, without removing the accuracy of reproduction? What is the exact relationship between these two potentially-conflicting yet desirable traits? An understanding of these aspects of stigmergic algorithms is fundamental to an understanding of the behaviour and design of stigmergic systems.

6.5.3 Improving Algorithm Extraction

As mentioned above, two processes are involved in the automatic extraction of stigmergic algorithms from existing structures. In order to improve the quality of the extracted algorithms, we must consider each of these processes working both independently and in union. In the following chapters, optimisation of both the state assignment and brick ordering processes will be considered.

Chapter 7

State Assignment

As seen in Chapter 6, the basic process of stigmergic script extraction can be split into two near-independent processes – brick ordering and brick state assignment. We will now consider these separately before looking in more detail at how they interact.

When comparing two stigmergic scripts targeted towards solving the same problem, all other aspects being equal the script which requires the least information is preferred – a lower number of brick colours and smaller algorithms allow the agents implementing these algorithms to be simpler and cheaper to produce. In the simple extraction method described in the previous section, the number of unique brick colours required for an architecture was exactly equal to the number of bricks in that architecture. Below we will discuss in more detail the role that brick colours play in stigmergic scripts, and begin to exploit some of the theoretical insights we obtained from the generative algorithm investigation in Chapter 5 to reduce the complexity[1] of the produced stigmergic algorithm.

7.1 Brick States and Rule Conflict Management

As discussed in Section 6.4.2, the only purpose of brick states is to ensure that rules which match local brick environments during construction only fire when they are required. It follows that differentiating rules according to brick states is only important when there is an actual threat that some rule may fire out of sequence. Fortunately we have already at

[1] See Section 10.3 for a consideration of the *complexity* of stigmergic algorithms.

our disposal a mechanism for analysing and reasoning with the causal dependencies between rules – *post-rules* (see Section 5.4.1).

Typically a rule will have a large number of post-rules. When considering these from the perspective of systematic generation of sequences of rules, the combinatorial explosion is, as we have seen, prohibitive of significant further investigation. However, when considering an architecture which already exists, we have gained a significant advantage over our previous position: for each rule we have extracted from the input structure, we are now capable of identifying exactly which rules present within our algorithm could fire next. For each rule we have extracted based on our brick ordering, we can determine which of the set of all *extracted* rules are post-rules.

For each brick placed in the desired architecture, we know those bricks which are neighbours in the final construction. Therefore, we know which rules ought to fire to place each of those bricks, but more importantly we can identify those rules which should be *prevented* from firing and placing neighbours in this position. Only these undesired or *conflicting post-rules* must be managed using differing brick values to ensure that the resulting structure does not diverge from our desired output.

7.2 Post-Rule Conflict Resolution

In this section we will explore how we can use post-rule generation to assign brick colours and enable stigmergic algorithms to build precise architectures.

7.2.1 The Simplest System, Revisited

Let us first consider another simplest-case example. Figure 7.1 shows a simple two-brick architecture and the results of simple rule extraction. For convenience, algorithms may no longer show the initial rule containing a single red brick (such as that which would place brick **0** in Figure 7.1, since this rule is present in all algorithms.

As can be seen from the simulation of this extracted algorithm, the resulting architecture does not match the input structure. The single rule continues to fire and produces a

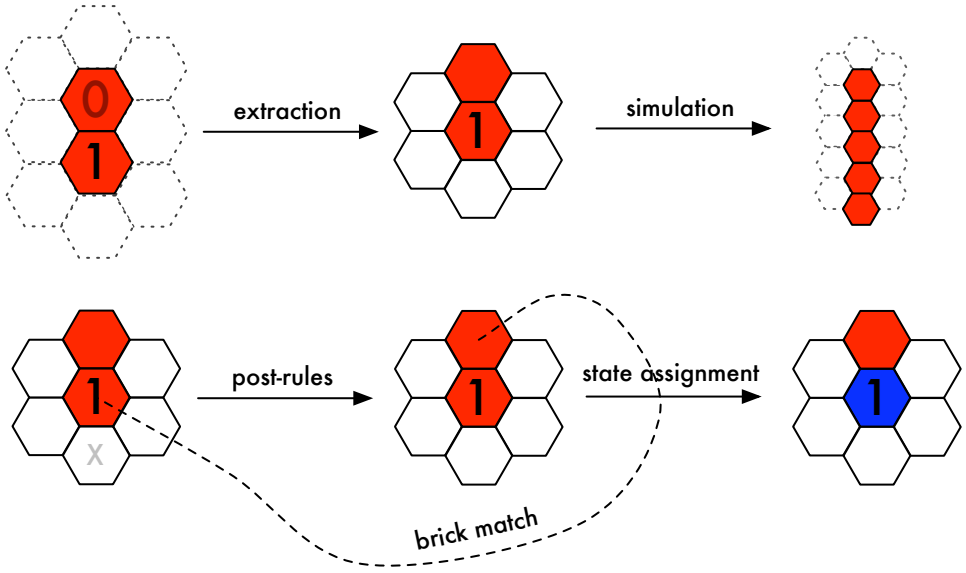


Figure 7.1: State Assignment in a simple, single-rule algorithm.

potentially-endless column of bricks. It can be clearly seen (and automatically determined, as described in Section 5.4.1) that this rule is self-activating[2], and this post-rule conflict must be managed if the algorithm's behaviour is to be contained.

During post-rule calculation, we determine which of the empty cells within a rule might be the site of a successive rule match. In situations where we *know* the actual set of rules which can fire during the simulation, we can further determine which of the bricks in a post-rule match against specific bricks in the original rule, and thus determine those bricks which can be used to control matching between these rules.

In Figure 7.1, the position of the post-rule match is indicated in the original rule by a greyed **x**. Knowing this, we can see the brick 'overlap' between the two rules, and the matching bricks are indicated on the figure. Since we want to *prevent* this match, we can see that we must alter the colour of one of these bricks in order to stop the rule firing again. We have modified the build-cell of the original rule, rather than the edge-cell of the post-rule, since in this case they are actually the same rule and the latter modification would also prevent the rule matching against the initial red cell. Now that the rule is modified, we are left with the same system shown in Figure 5.7 (disregarding the difference in geometries), and we can see that in simulation the correct structure will be reproduced.

[2]See Section 5.5.4

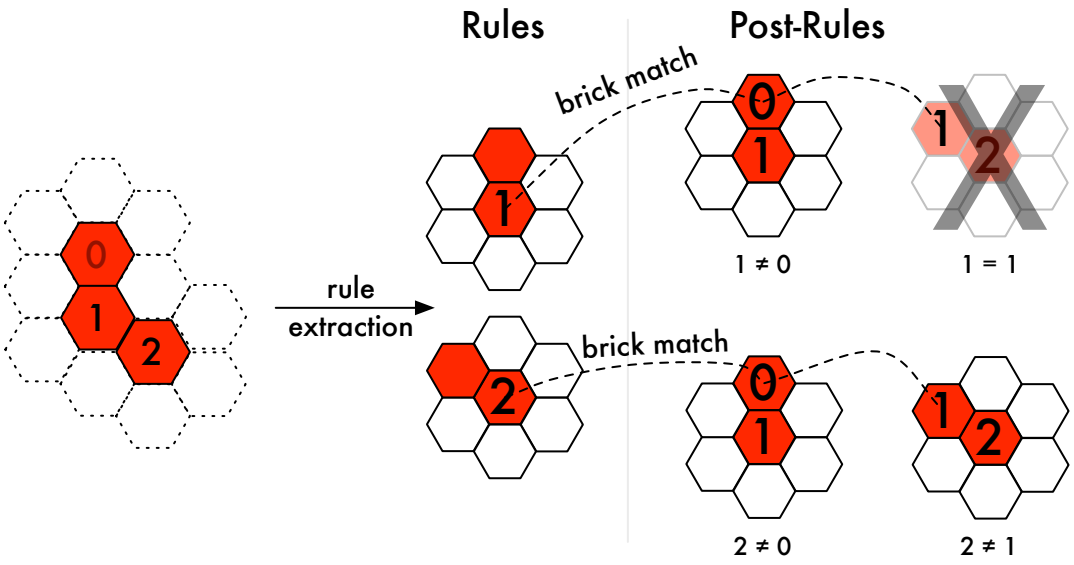


Figure 7.2: A stigmergic system with two rules. Post-rule conflicts are identified, along with the specific brick matches.

7.3 From One to Many: Desired Post-Rules

Let us now consider a slightly more complex system, such as the one depicted in Figure 7.2. In this system we now have two rules to consider: **Rule 1** and **Rule 2**. Each of these rules has two post-rules, as shown in the figure. Rule **1** has both itself, and Rule **2** as post-rules. We do not wish Rule **1** to fire repeatedly (as seen initially in Figure 7.1), but we *do* require that Rule **2** fire directly after Rule **1**.

We can determine this automatically by examining the brick matches for the post-rules of Rule **1**. For the first post-rule, brick **0** matches against brick **1** (the brick placed by Rule **1**). Because Rule **1** is *only* to match against brick **0**, and not brick **1**, we know that this represents a conflict which must be resolved. On the other hand, the second post-rule matches brick **1** against brick **1**, indicating that this post-rule *is* intended to fire next in the building sequence. This post-rule therefore does not produce any conflict, so we shouldn't consider it any further. Put simply, if the matched brick *id* is equal to the brick *id* of the rule, this post-rule is *desired* and does not require modification of brick colours to prevent it from firing.

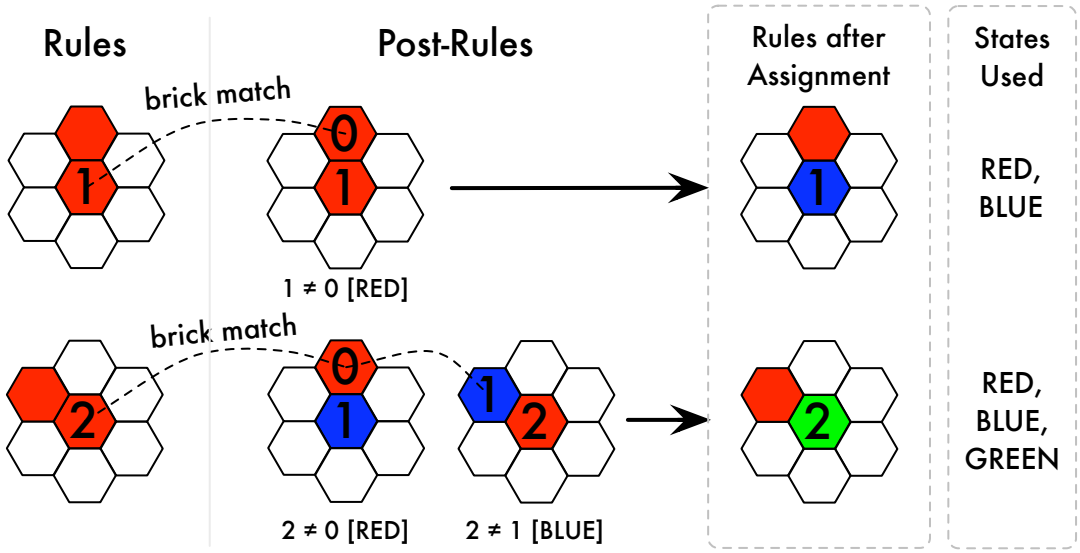


Figure 7.3: State assignment using the rules from Figure 7.2.

7.3.1 State Assignment with Two Rules

Once the desired post-rules have been eliminated, all remaining post-rules must be considered for brick colour assignment. This is shown in Figure 7.3. The remaining conflict for Rule **1** lies between brick **0** and brick **1**. Both bricks are red, so we set the build brick for Rule **1** to be `BLUE`, removing the conflict. We also add `BLUE` to the set of states in the algorithm.

Rule **2** has two undesired post-rules, with brick **2** built by the rule matching against bricks **0** and **1**. The conflicting bricks are `RED` and `BLUE`, so we must again add a new state – `GREEN` – and assign this value to the build brick of Rule **2**. The mechanism used to select the colour for assignment should now be quite clear: as we consider each rule, we have a set, S containing all of the brick states used by the algorithm thus far; to select the new state, we produce the set difference $S - U$, where U is the set of all states used by matching bricks. If $S - U = \emptyset$, we generate a new state n and add it to S for use as we continue.

7.4 Brick Tagging

Our state assignment mechanism is taking shape, but there are yet more complex examples which produce new problems to be considered. The state assignment of a ring structure, as shown in Figure 7.4, demonstrates the final significant failing in our current method. The

ring structure is ordered arbitrarily and rules are extracted, as shown at the very top of the figure, and each rule is then considered in turn. In this example, in order to be entirely general, all bricks are set to have the state `UNDEFINED`, rather than `RED` as in previous examples. This will clarify the assignment of states during the process.

The first rule places our initial brick within the environment. It has five post-rules, one of which can be safely ignored as it is the rule which builds an adjacent brick – a 'desired' post-rule. For the remainder of the rules, we determine which bricks within those rule matched bricks in Rule **1** (shown in the figure as white bricks with coloured *id*s, grey to indicate those bricks are `UNDEFINED` in this particular instance). We then collect the set U of all brick colours used in those matching bricks, minus the value `UNDEFINED`, which represents an *absence* of colour, rather than a colour itself, in order to determine what colour we may assign to the build brick of this rule. The reasoning behind this is as follows: in order for a post-rule to fire, the colour of the brick this rule places must be identical to that of the matched brick in the post-rule. By assigning the colour such that it matches *none* of the colours of those bricks within post-rules, we are ensuring that those post-rules can never fire as a result of the placement of this brick.

U is empty for the post-rules of Rule **1**, and subtracting this from the set of all available states, S (again initially empty) gives us yet another empty set. In this situation, as described above, we must create a new brick colour – `RED` – and both add this to the set of available states, and assign it to the build brick of the rule currently being considered. Thus at the end of processing Rule **1**, we have assigned `RED` as the colour of brick **1**.

Next, Rule **2** is considered, with only a single post-rule conflict. The matching brick in this case is `RED`, leaving us with no further available states, so a new state `BLUE` is added and assigned to brick **2**. Rule **3** continues in this manner, requiring a further colour `GREEN` to be added to the set of valid states for this algorithm.

When considering Rule **4**, we determine the states of the matching bricks within the post-rules to be {`BLUE`,`GREEN`}, leaving us with `RED` available for assignment to brick **4**. This process continues similarly for Rules **5** and **6**. In total, our algorithm has determined that this set of rules requires **four** states to operate as intended.

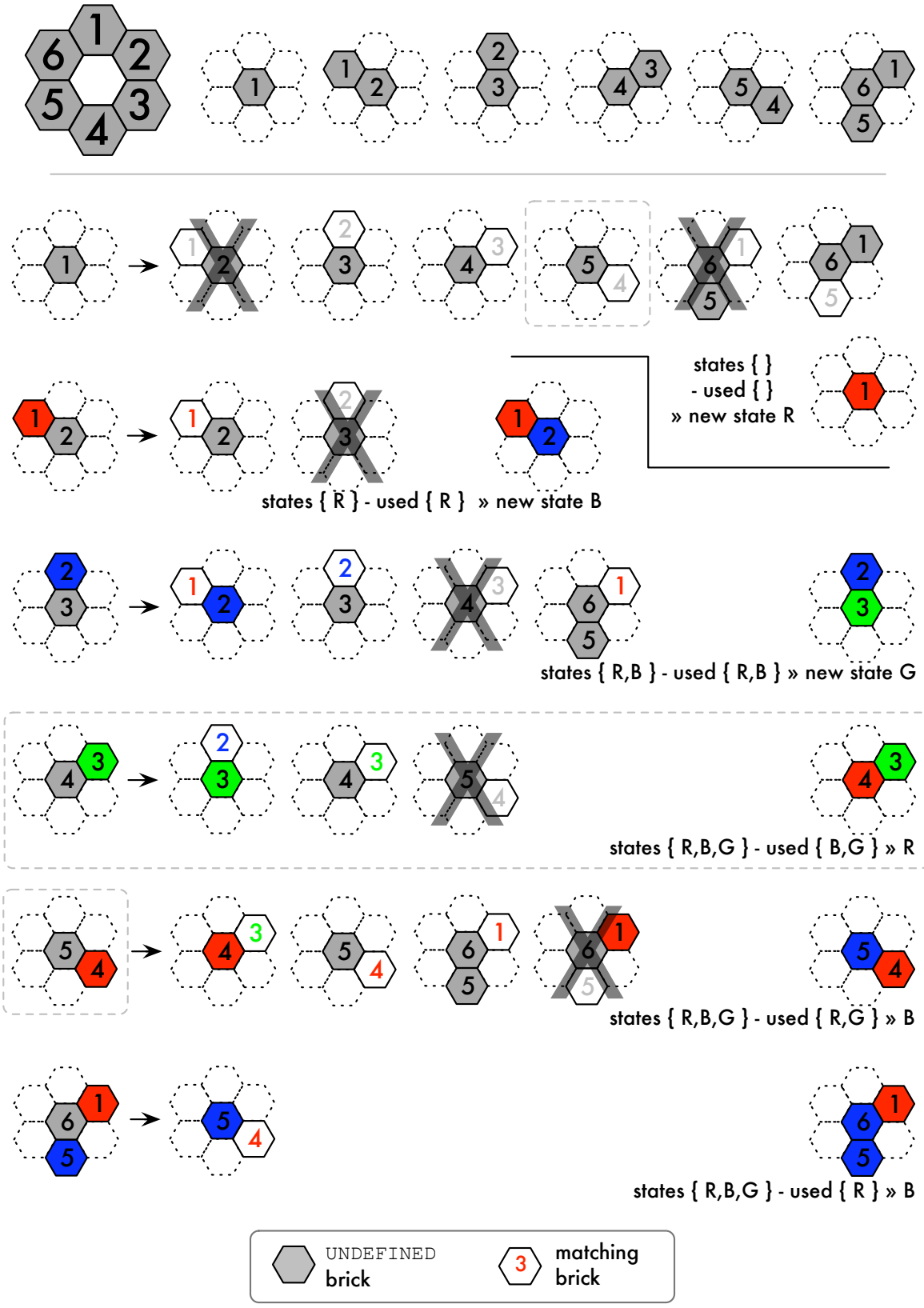


Figure 7.4: A more complex state assignment example. Rule **5** is detected as a post-rule of Rule **1**, but the later assignment of colour `RED` to brick **4** fails to take this information into account. The resulting algorithm is flawed. All bricks are initially of `UNDEFINED` (grey) state. Matching bricks in post-rules are shown as unfilled with coloured *id*. 'Desired' post-rules are crossed out.

7.4.1 Missing Post-Rule Information

However, closer inspection of the stigmergic algorithm we have produced shows that it contains a signficant flaw. Rule **5** (highlighted in Figure 7.4, which matches against a single red brick, could easily fire after Rule **1** places the initial brick. The reproduction of our original structure is therefore not guaranteed. How was this error produced?

Rule **5** was noted as a post-rule of Rule **1** when state assignment was performed on Rule **1**. At this point, we determined that brick **4** matched against brick **1**, but at that point in the algorithm brick **4** is `UNDEFINED` and thus not perceived as having a conflicting state. Later in the process when we are assigning a state to brick **4**, we cannot take into account the conflict between this brick as it appears in Rule **5**, and the brick placed by Rule **1**, since neither of these rules appear as post-rules of Rule **4**.

Because brick **4** was `UNDEFINED` at this stage, its state does not affect the colour assigned to brick **1**, but we have determined that these two bricks must not share the same colour. In order to reintegrate this information into the state assignment process, we can pursue two alternative methods of resolution, outlined below.

7.4.2 Assigning Values to Post-Rule Bricks

At the point where we realise that brick **1** must be differentiated from brick **4** (and in fact bricks **2**, **3** and **5**), any of those bricks which are `UNDEFINED` must be defined in order to remove the conflict. The results of such a process are shown in Figure 7.5. We first assign a colour to brick **1** – `RED`, as before. To resolve the remaining conflicting `UNDEFINED` bricks, we can assign them all identical values, as shown in part **a)** of Figure 7.5. The specific value derived using $S - U$ as before. We are now guaranteed that none of the remaining post-rules will fire after Rule **1**. However, it is obvious that this scheme is flawed, as in even this simple example, the resulting Rules **3**, **4** and **5** are all self-activating (See Section 5.5.4); we have introduced as much conflict as we attempted to resolve.

An obvious method to avoid the introduction of such new conflicts is by simply assigning each conflicting brick a unique colour, creating new colours where necessary. This is illustrated in part **b)** of Figure 7.5. Here we can see there are no self-activating rules, and using

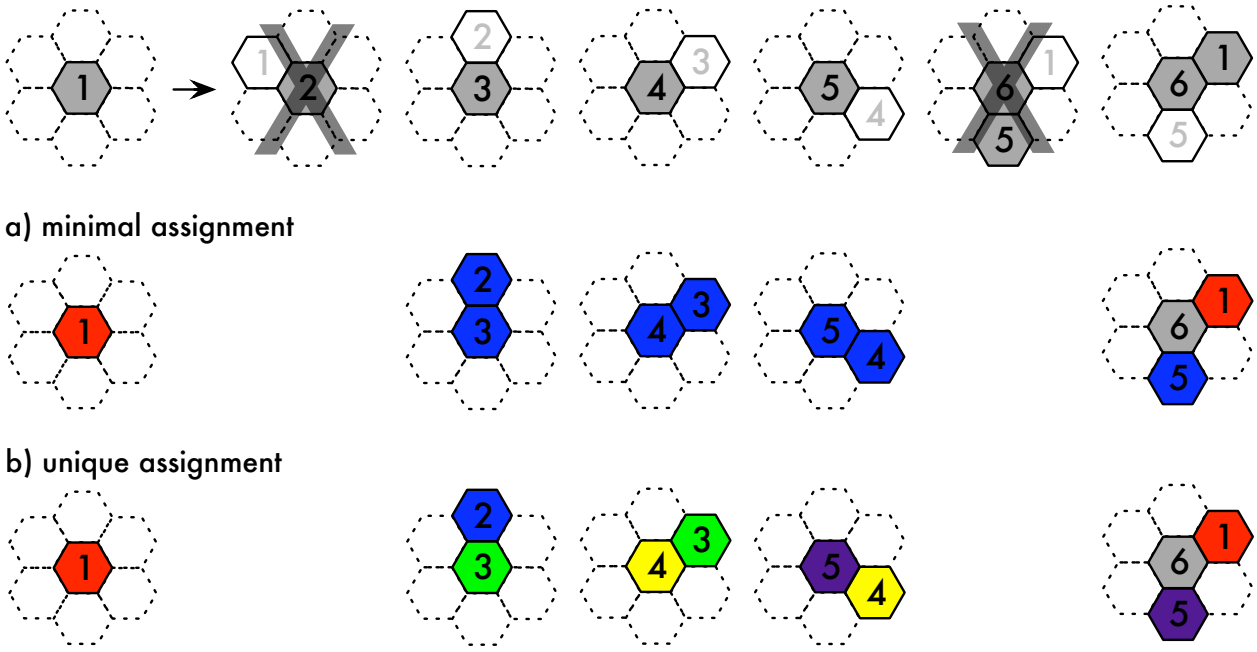


Figure 7.5: An illustration of post-rule state assignment during algorithm state assigment. The resulting assignments **a)** fail to consider the dependencies between post-rules, and **b)** introduce more states than are required.

this process will result in an algorithm which successfully and reliably reproduces the input structure. However, this does not process an algorithm with the minimal number of states. Within the context of this example, we can see by referring to the hand-tested derivations in Figure 6.13 that only four colours are actually required for this particular rule set, certainly not the five we have already introduced in this first step of state assignment.

By assigning colours to bricks *other than the build brick of the rule currently being considered*, we do not take into account the intra-post-rule links, and so we cannot make informed decisions about how best to colour those bricks. The second approach amounts to assigning unique states as in our original, simple approach, and fails to produce optimal assignments. We must devise some means of resolving these conflicts without assigning colours to bricks within post-rules.

7.4.3 Tagging

The brick-match information determined at each stage throughout the state assignment process is the key to this problem. This information must be preserved when it cannot be

acted upon directly. We will now outline our chosen means to achieve this, and in doing so produce an algorithm which is capable of assigning the minimum number of brick colours to any stigmergic algorithm we might consider here.

When our algorithm is processing Rule **4**, if it was aware of the colour restriction between brick **1** and **4**, it could use this information to include the colour of brick **1** in the set of colours which are *not* available for assignment. By 'tagging' brick **4** with the *colour* of brick **1**, the algorithm now has this information available at the required stage of processing. This is depicted in Figure 7.6. Once the 'desired' post-rules have been removed, those matching bricks which are `UNDEFINED` are tagged with the *colour* of the build brick currently under assignment. In this particular example, our initial rule[3] has four unresolved post-rules, so the matching bricks in each rule are tagged with the colour of brick *id* **1**, `RED`.

In this example, the state assignments proceed as before (Figure 7.4) until we reach Rule **4**. Here we can see the matching bricks in the post-rules indicate that neither `BLUE` or `GREEN` can be assigned to brick **4**. However, upon examination of the *tags* within brick **4**, we also note the previous conflict with brick **1** (`RED`), and so this colour is added to the set of *used* colours. The selection of possible brick colours is modified from $S - U$ to $S - (U \cup T)$ where T is the set of all colours assigned to bricks within the *tags* of the current brick. Brick **4** is therefore assigned the new colour `YELLOW`, rather than `RED`. The remainder of brick state assignment proceeds in this manner, considering the previous post-rule conflicts with each brick using tag values.

7.4.4 Minimal State Assignment

The final algorithm and structure is shown in Figure 7.7. We can see our output stigmergic algorithm uses only *four* colours, rather than the six required by our simple extraction technique. Furthermore, the number of colours determined by this extraction algorithm is the *minimum* number of colours required to reliably and accurately reproduce the *rule set* given, since a new colour is **only** added to the stigmergic algorithm when required to resolve

[3]It is crucial that we consider the initial rule (even if we do not display this rule or explicitly describe it) – when performing state assignment we must consider its post-rules if we are to successfully derive a valid algorithm.

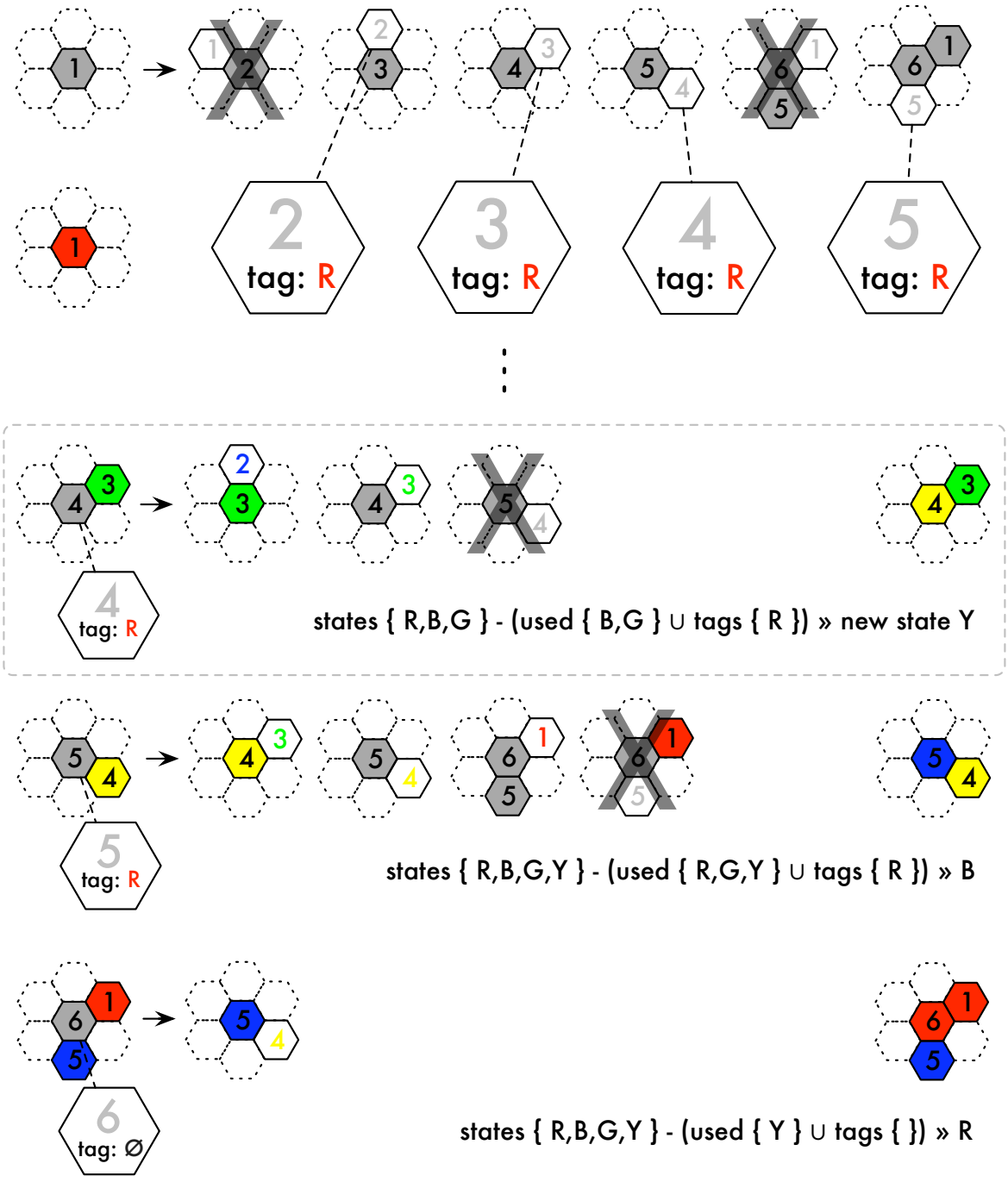


Figure 7.6: State assignment using 'tagging'. Assignment proceeds as Figure 7.4 until the consideration of Rule **4**. Previous brick match conflict information is combined with current post-rules to determine the new brick colour.

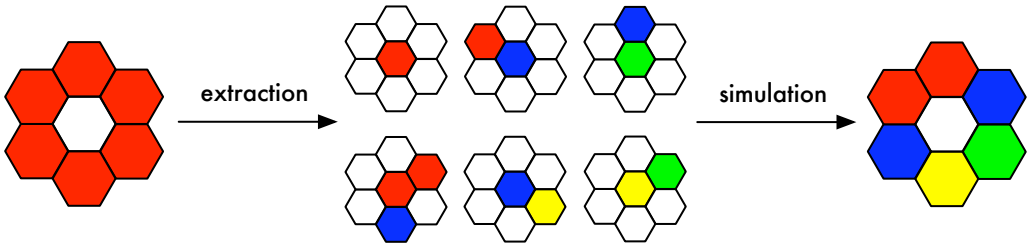


Figure 7.7: The final, state-optimised algorithm, shown with simulation results.

a conflict between post-rules already present within the rule set.

7.5 The Increasing States Algorithm

The algorithm presented above has been named the **Increasing States Algorithm**, due to the gradual addition of brick states as each conflict is resolved. A more formal description of the algorithm is found in Figure 7.8, the Ruby implementation is outlined in Figure 7.9

An arbitrary architecture is processed in Figure 7.10, which demonstrates each key stage during the Increasing States algorithm. It is important to note in this example that during the processing of Rule **5**, the `YELLOW` colour from brick **4** in Rule **7** is not included in U, because the post-rule conflict is already resolved, since this post-rule requires both a match between bricks **5** and **4**, but also bricks **2** and **3**. However, brick **2** is already set `BLUE` and brick **3** is set `GREEN`, so a match of this post-rule is already prevented. This is indicated by a large red cross through the rule. In total, four states are required to build this architecture.

7.6 State Assignment and Rotated Rules

Up to this point it has been assumed that rule rotations are not matched during simulations using these extracted stigmergic scripts. In principle there would appear to be no reason for this restriction; all we are required to produce for consideration is a set of post-rules, and rotated post-rules can be easily determined using the the rotation techniques described in Sections 4.5.2 and 4.6.1. However, as will be shown below, this is not the case.

Figure 7.11 demonstrates the fundamental problem produced by allowing rule rotations during simulation. The figure presents a small fragment of the rule extraction process,

To assign the minimal number of brick colours to a rule set R:

1. Let the set of all states in the algorithm $S = \{\texttt{RED}\}$
2. $\forall r \in R$:
   (a) Generate the post-rules P_r for r
   (b) $P_r = P_r \cap R$ – include in P_r only those post-rules actually in our rule-set
   (c) $P_r = P_r - D(r, P_r)$, where $D(x, Y)$ is the set of all post-rules in Y which are differentiated by state from x.
   (d) $\mathrm{t}(r_{build})$ is the set of values of all bricks for which this r_{build} was tagged
   (e) Let B be the set of those bricks from the post-rules P_r which match against the build-cell of r, r_{build}
   (f) $U = \cup_{b \in B} \mathrm{value}(b)$ – the set of all values of the bricks in B
   (g) $V = S - (U \cup T)$
   (h) If $V = \emptyset$ then
      i. Create a new state s
      ii. $S = S + s$
      iii. $V = \{s\}$
   (i) $\mathrm{value}(r_{build}) = v$ where $v \in V$
   (j) $P_r = P_r - D(r, P_r)$
   (k) Let B be the set of those bricks from the remaining post-rules P_r which match against the build-cell of r, r_{build}
   (l) $\forall b \in B : \ \mathrm{t}(b) = \mathrm{t}(b) + \mathrm{value}(r_{build})$ – tag each brick with the value of r_{build}

Figure 7.8: The *Increasing States* Algorithm.

```
# set all cell states to UNDEFINED
architecture.setBricks(CELL_UNDEFINED)

rules = architecture.rules.to_a.sort {|x,y| x.order <=> y.order }

tags = Array.new(@architecture.numBricks)

rules.each { |currentRule| # for each rule currentRule in rules

  # find all postrules Rp for Rx
  postrules, matchingBricks = currentRule.postrulesWithCells(@rules)

  # we create a simple array of index values so we know which
  # postrules still require some consideration
  postruleIndexes = []
  postrules.length.times { |x| postruleIndexes << x }

  # firstly remove any postrules which are already differentiated
  # with state
  removeDifferentiatedPostRules(postRules, postruleIndexes)

  # assign a value to the build brick of the current rule, based
  # on the matchingBricks, brick tags and available states.
  processRuleCells(matchingBricks, postruleIndexes,
                   currentRule, postrules)

  # Again, remove any newly-differentiated post-rules
  removeDifferentiatedPostRules(matchingBricks, postruleIndexes)

  # Tag any UNDEFINED bricks in the remaining post-rules
  tagRemainingPostRuleCells(matchingBricks, postruleIndexes,
                            currentRule, postrules)

} # end of 'rules.each'
```

Figure 7.9: The *Increasing States Algorithm* implementation, in Ruby.

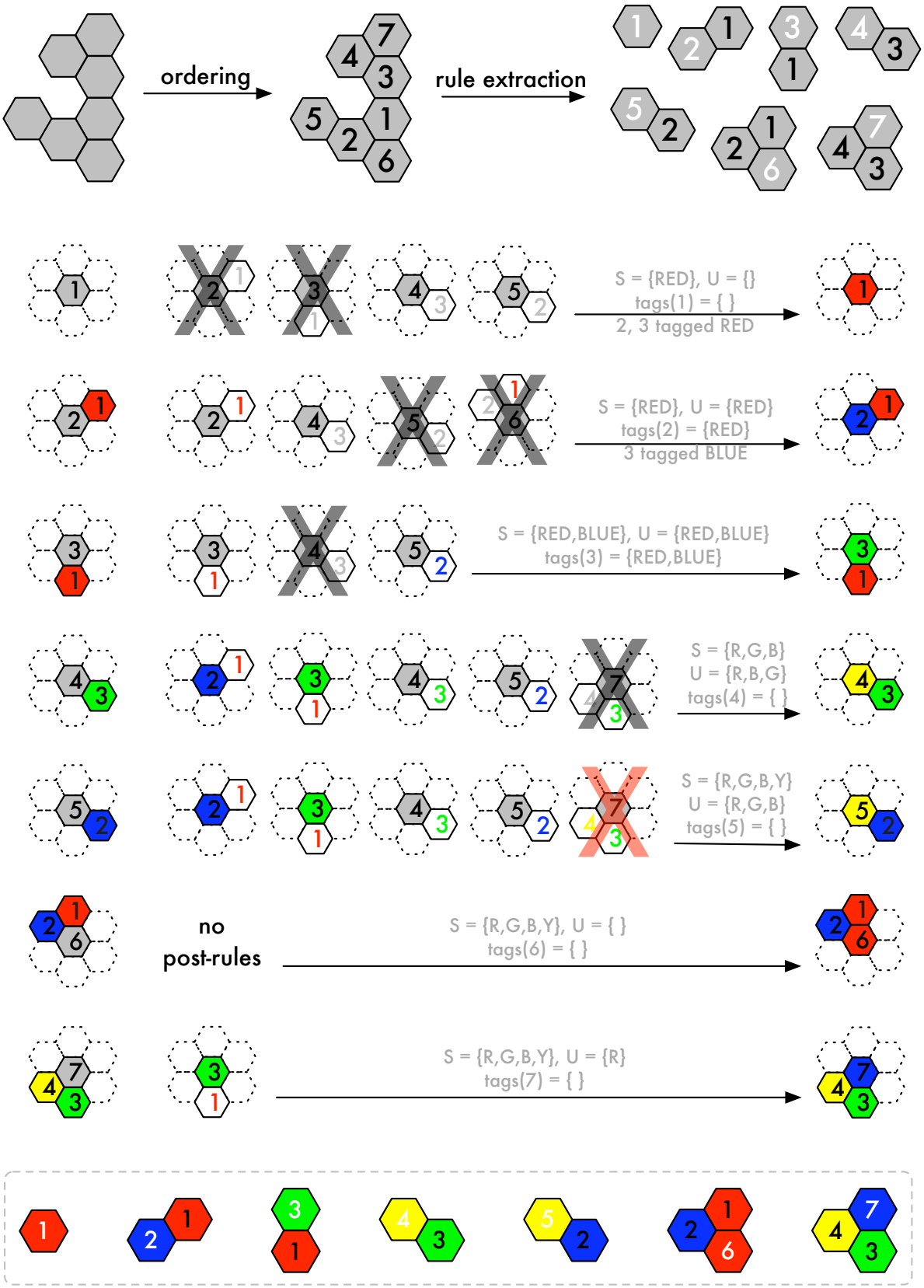


Figure 7.10: The *Increasing States* algorithm performed on an arbitrary structure.

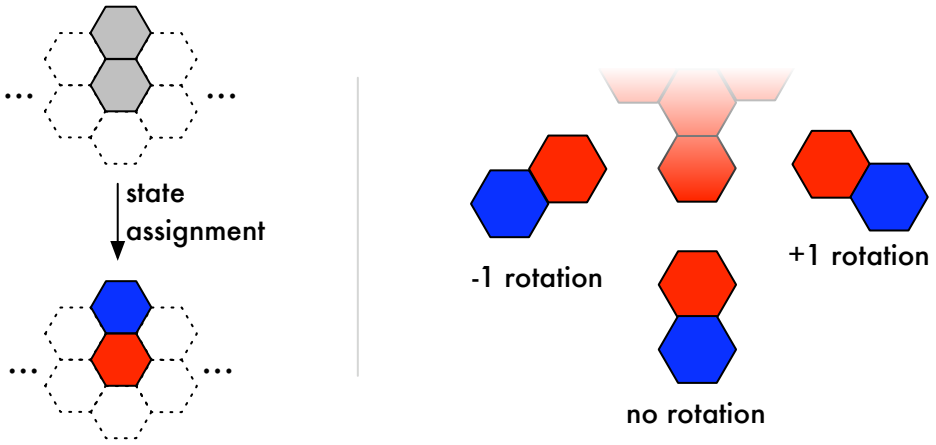


Figure 7.11: Despite state assignment, if rotations are allowed the progress of construction is not deterministic.

indicating that somewhere in this algorithm, the simple rule shown[4] is extracted. State assignment will ensure that differing colours are assigned to each of the bricks, as we have seen previously. However, if rotations of the rule are allowed to match, we see that if a situation such as that displayed on the right is encountered during the simulation, this rule can match in three different rotations, each producing a unique overall structure.

Most importantly, *there is no way to control which rule rotation is used in this situation.* To control construction in this abstract stigmergic system, the two mechanisms we have available to constrain building behaviour are *local structure* and *brick colouring.* It is clear in this situation that no combination of brick colours could prevent divergence from the desired architecture. This is an example of a conflict where brick state assignment cannot be used to control how the rule matches the local environment. The symmetries present in this situation prevent any effective differentiation.

For this reason, from this point forward extracted algorithms will not be simulated allowing rotations of their rules. This problem is a very specific instance of the larger issue of reproductive accuracy, considered in detail in Chapter 9.

[4]This rule has now been considered several times, along with its cubic and rotated counterparts: see Figures 5.6, 5.7, 5.8, 5.20, 7.1, 7.2, 7.3

7.7 Evaluation of Increasing States Algorithm

We have now seen that 'bottom-up' investigation – wherein we try to understand the global behaviour and capabilities of abstract stigmergic systems from the structure of the rules themselves – is computationally intractable. So, we have presented an alternative, more practical 'top-down' approach, wherein we exploit the information already present within an architecture to decompose it into a corresponding stigmergic algorithm. We have seen:

A stigmergic algorithm can be extracted from any architecture – as shown by the simplest extraction technique in Section 6.2, once a *valid* brick ordering has been established, in whatever manner, rules can be extracted and bricks coloured in such a way that guarantees accurate reconstruction.

The *Increasing States* algorithm – which produces minimal brick colourings for any given stigmergic algorithm, since new brick colours are only added to the algorithm when absolutely required. This is achieved whilst ensuring that the architecture will be constructed accurately.

As stated above, the *Increasing States* algorithm produces minimal brick colours for any specific rule set, but the structure of the rules depends completely on the ordering used to extract them. Therefore, different orderings will result in differences in the minimum number of colours for any given architecture.

The two steps required to extract a stigmergic algorithm were outlined as below in Sections 6.5:

1. Assign a *valid* order to the bricks within the architecture

2. Assign a colour to each brick within the architecture

We have considered in detail the latter, and produced the *Increasing States* algorithm as a solution to this half of the extraction process. In the following chapter, the orderings process will be examined, and we will begin to make our conclusions regarding the nature of stigmergic algorithm extraction.

Chapter 8

Ordering

In Section 6.4.1 the issue of providing a *valid* ordering was introduced, and it has been shown that once the order in which the bricks were placed is established, the structure of the rules can be easily extracted, and the minimum number of states required to build the architecture can be clearly established. In this section we will consider the ordering process itself, and how this affects both the rules extracted and the algorithm produced.

8.1 The Importance of Ordering

The relationship between ordering and extracting compact algorithms was briefly considered in Section 6.5. As was shown in Figure 6.12, changes in the ordering of the bricks within an architecture will produce rules with different structures. While the Increasing States algorithm will produce a minimal set of brick colourings for any *particular* rule set, that minimal number depends on the structure of the rules within that algorithm, which in turn depends on the ordering assigned. In order to determine the *absolute minimum* number of brick colours required by an arbitrary *structure*, we must therefore find the brick ordering which minimises the results of the Increasing States algorithm.

8.2 Ordering Problems

Many interesting ordering problems exist and have been the subject of extensive research. Perhaps the most famous is the Travelling Salesman Problem[93, 112][1], or TSP. Given a finite number of 'cities' with some pre-determined cost of travelling between each pair, we must find the cheapest route which allows the salesman to visit all cities and return to his starting point. If links exist between all cities, then the total number of tours is given by $\frac{(n-1)!}{2}$, where n is the number of cities the salesman must visit.

The TSP problem can be restated as a problem rooted in graph theory, where the cities are the vertices in a finite complete graph, and integer weights (distances) are assigned to each edge. The solution to the problem then becomes the derivation of the Hamiltonian cycle (that is, a cycle passing through all the vertices) with minimum weight.

The TSP problem is a simple-yet-general example of a combinatorial problem, and is analogous to many real-life problems such as route planning, circuit board fabrication and job scheduling. The TSP is very easy to describe and yet very difficult to solve since no polynomial time algorithm is known which can be used to find optimal solutions. As such, it has become the classic example of an *NP-complete* problem[71]. An excellent library of TSP software[136] has been collected, containing many examples and solutions to TSP problems.

8.2.1 Job-Shop Scheduling

The Job-Shop Scheduling problem[91, 62] is perhaps a closer analogue to the brick ordering problem we wish to consider in this chapter. In this problem a number of 'jobs' (processes, tasks, etc.) must be performed using limited processing resources. For example, with a limited number of factory machines, a number of 'jobs' must be scheduled on those machines; the optimal solution completes all jobs in the minimum amount of time.

The primary difference between this ordering problem and TSP problems is the additional constraint that the underlying graph is typically *directed* and *incomplete*. In order words, there are external constraints which insist that some jobs must be performed before others.

[1] An extensive bibliography of Travelling Salesman Problem information can be found at http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

Formally stated, the problem graph is described by an acyclic graph:

$$G = (V, E)$$

where the vertices represent jobs and the edge (u, v) implies that job u must be performed before job v.

This more closely matches brick ordering:

$$G = (B, L)$$

where B is the set of all bricks in the structure and L the set of neighbourhood links, such that

$$\forall b_1, b_2 \in B, (b_1, b_2) \in L$$

if and only if b_2 is a brick placed after b_1 and b_2 is in the neighbourhood of b_1.

Just as the TSP has become a standard problem in combinatorial analyis, job-shop scheduling can be similarly considered prototypical of all constrained combinatorial optimisation problems.

8.2.2 Genetic Algorithms for Combinatorial Problems

Due to the *NP-hard* nature of this family of combinatorial problems, approximate methods must often be used to search for acceptable solutions. Barring the advent of practical quantum computing[80] which might give means to search all solutions in parallel, there is little hope of finding a *tractable* optimal algorithm for solving this problem.

Instead, approximate methods such as simulated annealing[112, 101], genetic algorithms[127, 78, 74, 24] and various other approximate methods[59, 9, 4] provide a practical, tractable means of obtaining acceptable, if not optimal, solutions. Using a genetic algorithm to facilitate this search is now commonplace[6], and has been exploited to great success in both TSP problems[111] and job-shop scheduling[57].

8.2.3 An Overview of the Genetic Algorithm Approach

An outline of the typical sequence of events undertaken when using a genetic algorithm is shown in Figure 8.1. First a random population of individuals is generated. Each of these individuals is some representation of a solution to the problem under consideration. For example, in the TSP context, an individual solution might simply be a list of the cities to be travelled in the order they should be traversed. From a programming perspective, these individuals are strings of data. Next, each member of the population is evaluated to determine the fitness of that particular solution. Those solutions which are fitter than others are more likely to be selected to produce 'offspring' which will be included in subsequent generations. These offspring are produced using a *crossover* function, which uses a mix of data from two 'parent' solutions to produce 'child' offspring. The solutions are also subjected to a low probability mutation, in which some small part of the solution string is modified. The population size is maintained by stripping poor solutions, and the new generation of solutions is then evaluated and new offspring generated.

The motivation behind this approach is that the children of solutions of high quality should inherit aspects of their parents which were responsible for their high quality, and as such, the children of two good solutions may have a higher fitness than either of the parents. In this manner, the fitness of the population steadily increases until a solution of acceptable fitness is found.

8.3 Ordering Bricks using a Genetic Algorithm

The two most important steps to take towards reaching that solution are first defining a fitness function to evaluate individuals, and secondly selecting an appropriate representation for each ordering solution. Our fitness function is simply the minimum number of brick colours required by the algorithm produced by this ordering. This can be determined using the Increasing States algorithm developed in Chapter 7.

The information which must be encoded and tested when evaluating the orderings of bricks within a structure is the set of choices required to produce a fixed, ordered architecture.

1. Create a population P of random solutions
2. Until there exists a solution in P of acceptable fitness
   (a) Evaluate each solution s in P with some **fitness function** $f(s)$
   (b) Select the fittest solutions in the population and create offspring solutions using a **crossover** operator, to replace less-fit candidates
   (c) **Mutate** some solutions randomly

Figure 8.1: The typical workflow of a genetic algorithm.

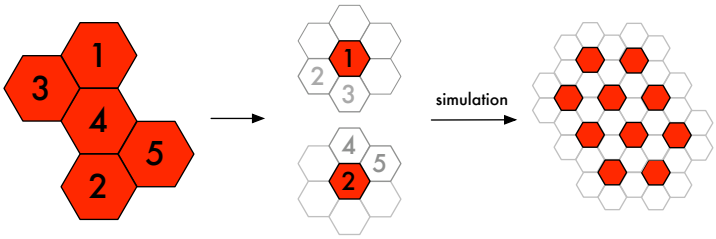


Figure 8.2: Two single-brick rules are produced with the random brick ordering of a simple architecture. The resulting simulation features many unconnected orphan bricks. *Repeated from Figure 6.6.*

Fundamentally each choice in this operation results in the selection of the next brick, and so for an architecture constituted of n bricks, there are always exactly n choices required to fully order the bricks. For this reason a fixed-length genome representation, or **gene string**, is appropriate to our needs.

The method of encoding a solution can have an enormous impact on the speed and effectiveness of this process. Typically encodings fall into two categories - **direct** and **indirect**. Each of these will be discussed below.

8.3.1 Direct Encoding of Brick Ordering

The simplest encoding for the ordering of bricks within an architecture is simply a unique labelling of each brick, followed by the ordering those labels within the genome string itself. This can be seen in a simple structure in Figure 8.2.

This example illustrates a potential problem with a direct representation - it is very easy

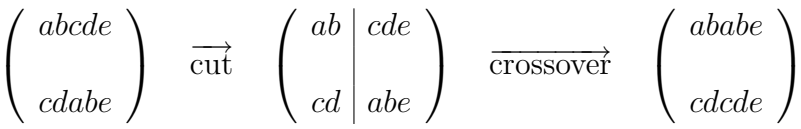


Figure 8.3: Simple crossover using a direct encoding representation. Invalid orderings are easily produced as a result of the crossover.

to produce an ordering of brick labels which does not map to a valid ordering of brick placing during the production of the given architecture.

8.3.2 Crossover using Direct Encoding

The implementation of the crossover operation depends entirely on the nature of the genome representation. For simple linear-string representations it is simple enough to specify a *cut-point* which describes the position in the string at which both candidates are cut, and latter parts replaced with the corresponding latter part of the partner (Figure 8.3). Many more complex types of crossover operation are possible, utilising for example a greater number of cut-points and sophisticated methods for determining the position of these points. However these are beyond the scope of this investigation; comprehensive surveys [111, 74, 84] are available for further information.

What becomes quickly clear when applying crossover to solutions within the population is that the offspring from two valid solutions is rarely a valid solution itself. Invalid brick ordering has been illustrated previously in Section 6.4.1. The two features of the children solutions which result in their invalidity are the repetition of bricks within the ordering, and the ordering itself being now invalid.

8.3.3 Repetition

As can be seen in Figure 8.3, both offspring contain repeated bricks (i.e. *ababe* features both brick a and b twice). Obviously this ordering cannot be valid since a brick cannot be built twice, and furthermore since the solutions are fixed-length some bricks are missing. In such a case, the duplicate brick labels must be removed and any missing bricks inserted. This is undesirable for two reasons. Firstly it requires additional computational effort to find

$$\begin{pmatrix} abcdef \\ \\ fedcba \end{pmatrix} \xrightarrow{\text{cut}} \left(\begin{array}{c|c} abc & def \\ \\ fed & cba \end{array}\right) \xrightarrow{\text{crossover}} \begin{pmatrix} abccba \\ \\ feddef \end{pmatrix}$$

Correction of **abccba**

Identify error labels	$\Rightarrow$	abc**cba**
Identify missing labels	$\Rightarrow$	**def**
Remove all duplicates	$\Rightarrow$	$abc\_\_$
Insert missing labels	$\Rightarrow$	abc**def**

Figure 8.4: Correction of repeated labels after a crossover operation. The final, corrected child ordering contains no information from the the second parent.

$$\begin{pmatrix} \cdots xbcdevk\cdots \\ \\ \cdots acbedgf\cdots \end{pmatrix} \xrightarrow{\text{cut}} \left(\begin{array}{c|c|c} \cdots x & bcde & vk\cdots \\ \cdots a & cbed & gf\cdots \end{array}\right) \xrightarrow{\text{crossover}} \begin{pmatrix} \cdots xcbedvk\cdots \\ \\ \cdots abcdegf\cdots \end{pmatrix}$$

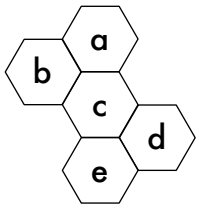
Figure 8.5: Advanced crossover pattern-finding with a direct encoding representation

duplicates and which brick labels are missing. More importantly, modifying the solution in this manner destroys information donated from one or both of the parent solutions, reducing the potential value of the cross-over operation as a whole. This is illustrated in Figure 8.4.

Techniques for implementing cross-over in constrained problems similar to this have been discussed in particular in the context of job-shop scheduling[169]. One particular technique[75, 160] which solves the problem of duplicate labels is outlined in Figure 8.5. This process identifies common subgroups of elements within the parent solution, and swaps these subgroups between the parents to produce offspring solutions. In the example in Figure 8.5, a subgroup containing the elements b, c, d and e has been identified in both parents. The order these elements appear is unimportant, only that they appear together as a group. These subgroups are swapped between the two parents, resulting in offspring with no brick label duplication.

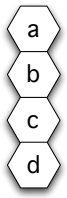
8.3.4 Structurally Invalid Orderings

It is possible that even when an offspring is created which features no duplication (or where duplication has been corrected somehow), the ordering will remain invalid due to the structural constraints of the architecture. For example, ordering **1** presented in Figure 8.6 features



1. a–e–b–c–d
2. a–b–c–d–e

Figure 8.6: An invalid random ordering for the given labelling/architecture. *See also Figure 6.5.*



$$\begin{pmatrix} abcd \\ \\ dcba \end{pmatrix} \xrightarrow[\text{cut}]{} \left(\begin{array}{c|c|c} a & bc & d \\ d & cb & a \end{array}\right) \xrightarrow{\text{crossover}} \begin{pmatrix} acbd \\ \\ dbca \end{pmatrix}$$

Figure 8.7: Label-grouped crossover resulting in offspring solutions with invalid orderings

no duplication, but is not a valid ordering. Brick **a**, being first in the ordering, is designated as the *initial brick*. Each brick subsequently placed must then be adjacent to an existing brick within the structure. Brick **e**, which is scheduled next in the ordering, can only be placed when one of the adjacent bricks has been placed previously. In this case, either brick **c** or **d** (or both) qualify, but since the ordering indicates neither of these bricks yet exists within the architecture, brick **e** cannot be placed and the ordering is *invalid*.

We have seen that the simplest crossover mechanism can easily introduce label duplication into offspring solutions. While the 'common-subgroup' method illustrated in Figure 8.5 guarantees no duplication of labels, neither the normal crossover mechanism, nor the 'common subgroup' method are guaranteed to produce valid offspring solutions. It is trivial to construct an example in which this form of crossover results in an invalid order, as can be seen in Figure 8.7.

8.3.5 Mutation using Direct Encoding

Mutation provides genetic diversity and enables the genetic algorithm to search a broader space. It can also be considered to 'nudge' the fitness of the population of solutions away from local maxima, and is usually applied according to some low probability (the exact value normally being determined experimentally). It is mutation which most clearly exposes

the fundamental problem with a direct encoding scheme for ordering bricks within a *Nest* architecture.

Within a direct encoding of orders, mutation can only be achieved through swapping one or more elements within the ordering. If we consider ordering **2** in Figure 8.6

$$[abcde] \quad \overrightarrow{mutate} \quad [ebcda]$$

However, this order is now invalid, because brick b cannot be placed before either a or c. To avoid this, elements must be carefully (and expensively) selected for mutation, or the resulting errors corrected, again incurring further computational expense. While for the small example structure in Figure 8.6 this is trivial, it is quite possible that a small change at the start of an order will require the remainder of the order to be modified to become valid again.

This pinpoints the crucial problem with using a direct encoding scheme for this type of problem: each choice made during the ordering process dramatically alters the set of subsequent choices which will result in a valid ordering.

8.3.6 Summary of Direct Encoding

It is clear now that a direct representation of brick ordering is far too brittle to be used within a GA. Even when means are devised to correct the errors which are almost certain to appear after cross-over and mutation, a large amount of computational effort must be expended and the resulting ordering after correction may share nothing with the original (as shown in Figure 8.4). The relationship between parent solutions and offspring is effectively broken, significantly compromising the fundamental strength of the genetic algorithm approach.

8.4 Indirect Representation of Brick Ordering

An indirect encoding scheme specifies a solution in some intermediate language, requiring reinterpretation before it can be considered for evaluation. By distancing the solution representation from the specific architecture, the flexibility which was missing in the direct

Let B be the set of all bricks within the architecture. The sequence of bricks representing the build order $O = \emptyset$. To produce a valid ordering:

1. Select a brick b from B at random, and append it to O. This brick is the *initial brick*
2. Until O contains all the bricks in the architecture:
   (a) From O, select a brick b, where N is the set of neighbouring bricks of b and $(N \cap O) \neq \emptyset$
   (b) Select a neighbour b' of brick b from $(N \cap O)$ and add it to O

Figure 8.8: The sequence of decisions taken when creating a valid ordering from an architecture.

encoding scheme described above allows us to alter solutions without completely destroying their validity.

To produce an indirect encoding, it is necessary to consider the process which is used to create a *valid* ordering in the first place. Essentially this process consists of 3 steps, outlined in Figure 8.8.

This process can be broken down into three distinct choices. The first choice, listed as step 1 in Figure 8.8, determines which brick is the *initial brick* – that is, the single brick which is not placed by a rule, but already present in the lattice. The remaining processes consists of selecting a brick we have already ordered which has unordered neighbours (step 2.(a)), and then adding one of those neighbours to the ordering (step 2.(b)).

If set B is represented as an ordered array (the ordering based on whatever unique brick labelling has been selected) and consists of n bricks, the initial choice is simply a random index within that array[2]: a random number between 0 and $n - 1$. To select a brick b from O (step 2.(a)), we can simply treat O like an array and indicate which element we should choose by its index $i_b, 0 \leq i_b < |O|$, where $|O|$ is the number of bricks already ordered at any point. Finally, we must select an unordered neighbour b' of b (step 2.(b)). If we examine the neighbourhood of b in a specific order, such as the cell indexing shown in Figure 4.6 in Chapter 4, we can add neighbours to N in a consistent manner and thus ensure that any index will always refer to the same brick for one particular neighbourhood. The choice of a

[2]Array indexes almost universally begin at zero within programming languages. In this instance indexing from zero is crucial since we will be utilising remainders.

1. Create a sorted array B containing all bricks in the architecture.
2. Create an empty array O which holds the ordering of brick labels
3. Select the brick at index *start* from B ($B[start]$), and add it to a list O. This brick is the *initial brick*. Remove $B[start]$ from U.
4. For each pair of indexes (i_x, n_x):
   (a) U is the set of all bricks in O which have unordered neighbours
   (b) $b = U[i_x]$.
   (c) Generate the neighbour array N from the bricks around b.
   (d) Remove all brick labels from N which appear already in O
   (e) $b' = N[n_x]$.
   (f) Insert b' at the end of O

Figure 8.9: The sequence of decisions taken when creating a valid ordering from an architecture.

neighbour b' therefore becomes the selection of an index n_b from an array representation of N, where $0 \leq n_b < |N|$.

This sequence of steps can be concretely represented as a list of indexes of the form:

$$start, (i_1, n_1), (i_2, n_2), \ldots, (i_{(x-1)}, n_{(x-1)})$$

where n is the total number of bricks in the architecture. Each pair (i_x, n_x) are indexes used to select the next brick in the ordering. This sequence of indexes identified a precise set of brick choices, and can thus be used to reliably encode specific architecture brick orderings.

With such a sequence of array index choices, we can use the algorithm implementation outlined in Figure 8.9 to produce O, the final global ordering sequence of each brick within the architecture. This ordering can then be used to extract rules using the rule extraction process outlined in Section 6.2.1.

8.4.1 Crossover using Indirect Encoding

In order to justify the extra processing required to convert this indirect encoding into an architecture ordering, the problems exhibited by a direct encoding during crossover must be

somehow avoided. By modifying the algorithm in Figure 8.8 a means of achieving this goal becomes readily available.

For an architecture of n bricks, each element within the gene string of this form of indirectly encoded solution is a number. However, as was seen with crossover using direct encodings, modifications to the representation can easily render a solution invalid. For each element within that string, the conditions below must be satisfied for it to be interpretable as a brick ordering:

start In the case of the first element, any number between 0 and $n-1$ is valid. If the number is greater than $n-1$, it will be out of bounds of the array representation of B.

Solution – Determine the actual value of *start* by taking the remainder of $\frac{start}{n}$, i.e. *start* mod n.

i_x For brick index i_x to be valid, $0 \leq i_x < |O|$, since the index must be an element in the set of previously-ordered bricks. If the value is greater than $|O|$, it is out of the bounds of the array O, leading to an error.

Solution Determine the *actual* index within U by taking the remainder of $\frac{i_x}{|U|}$, i.e. i_x mod $|U|$, where $|U|$ is the size of array U at the point during the ordering process when we encounter i_x.

n_x The size of N at any point depends entirely on which brick is selected by $U[i_x]$. If $n_x > |N|$ then the next brick in the ordering cannot be determined, again producing an error.

Solution The index of the next brick to be added to O is then determined by n_x mod $|N|$.

In this manner *any* string of numbers of length $2n+1$ can be processed into a *valid* ordering of an architecture of n bricks. Furthermore, this string can be modified in any way through cutting and replacing whilst remaining valid as above, requiring none of the error correction necessary when using a direct encoding.

8.4.2 Mutation using Indirect Encoding

Mutation is now simple to achieve as any element within the encoded genome can be modified in any way without compromising the validity of the solution it represents. The simplest form of mutation is selecting one digit at random from the string, and incrementing or decrementing its value.

8.4.3 Indirect Encoding Example

Below a concrete example of indirect ordering is presented. In Figure 8.10, two indirectly encoded orderings are presented. The first can be converted into an ordering without significant consideration. The second is a random string of numbers, which can also be converted deterministically into a valid brick ordering for the architecture. By using a random string of numbers, it is clear that *any* gene string, regardless of modifications produced through crossover or mutation, can be used to generate a *valid* brick ordering.

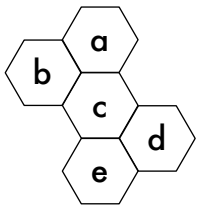
8.5 Genetic Algorithm Implementation

The indirect encoding scheme above has been implemented as part of the *Nest-3.0* system, and a number of simulations run to evaluate the effectiveness of using a genetic algorithm to order bricks. Six random architectures were generated, containing 10, 20, 30, 40, 50 and 100 bricks each. These architectures were then used as the subject of a genetic algorithm simulation as follows:

1. A random initial population of 50 valid ordering *solutions* is generated.

2. Each of these orderings is then applied to the architecture:

    - The architecture is then evaluated using the **Increasing States** algorithm to determine the minimum number of brick colours required to guarantee accurate construction with the rule set derived from that ordering.
    - The number of brick colours returned is assigned as the fitness of that particular ordering.



1. 2,0,0,1,0,0,0,1,0
2. 6,1,5,7,3,2,5,4,7

B = [a, b, c, d, e]

1. 2,0,0,1,0,0,0,1,0 ⇒ c-a-b-d-e

O = [**B**[*start*]] = [**B**[2]] = [c]
U = [c]

Step	(i_x, n_x)	**U**$[i_x]$	**N**	**N**$[n_x]$	**O**	**U**
1	$(0,0)$	**U**[0] = c	[a,d,b,e]	**N**[0] = a	[c,a]	[c,a]
2	$(1,0)$	**U**[1] = a	[b]	**N**[0] = b	[c,a,b]	[c]
3	$(0,0)$	**U**[0] = c	[d,e]	**N**[0] = d	[c,a,b,d]	[c,d]
4	$(1,0)$	**U**[1] = d	[e]	**N**[0] = a	[c,a,b,d,e]	[∅]

2. 6,1,5,7,3,2,5,4,7 ⇒ b-c-a-e-d

$start = (6 \bmod |B|) = (6 \bmod 5) = 1$
O = [**B**[*start*]] = [**B**[1]] = [b]
U = [b]

Step	(i_x, n_x)	**U**$[i_x \bmod \lvert U\rvert]$	**N**	**N**$[n_x \bmod \lvert N\rvert]$	**O**	**U**
1	$(1,5)$	**U**$[(1 \bmod 1 = 0)]$ = b	[a,c]	**N**$[(5 \bmod 2 = 1)]$ = c	[b,c]	[b,c]
2	$(7,3)$	**U**$[(7 \bmod 2 = 1)]$ = c	[a]	**N**$[(3 \bmod 1 = 0)]$ = a	[b,c,a]	[c]
3	$(2,5)$	**U**$[(2 \bmod 1 = 0)]$ = c	[d,e]	**N**$[(5 \bmod 2 = 1)]$ = e	[b,c,a,e]	[c,e]
4	$(4,7)$	**U**$[(4 \bmod 2 = 0)]$ = c	[d]	**N**$[(7 \bmod 1 = 0)]$ = d	[b,c,a,e,d]	[∅]

Figure 8.10: Examples of indirect encoding processed into a brick orderings. Each step shows the ordering of another brick, based on the interpretation of previous parts of the genome. In this example the order of precedence when generating N is `N,NE,SE,S,SW,NW`.

3. The population of ordering solutions is then sorted into ascending fitness order, and statistics recorded regarding the fitness of the population.

4. The best solution in the generation is passed through without modification by crossover or mutation. This procedure is known as *elitism*, and ensures that the best solution is never lost.

5. Until the new population contains the same number of solutions as the existing population

   - Two solutions are selected from the old population. The probability of any particular solution being selected is proportional to its fitness.
   - The crossover operator (see Section 8.4.1) may be applied to these solutions to generate two new child solutions, which are placed into the next generation instead. The probability of this operation occurring is set at 0.9.
   - As each solution passes into the next generation, it may be mutated (see Section 8.4.2) to produce a new solution. The probability of any solution being mutated is set at 0.04.

6. The process continues at Step **2**, unless:

   (a) A maximum number of iterations has been reached, or

   (b) The fitness of the best solution has not changed within a given number of iterations

8.6 Experimental Results

The results of these experiments are shown in Figure 8.11. On each chart, the y-axis delineates the *fitness* of the population, whist the progress through time from one generation to the next is represented along the x-axis. Each chart also displays three lines, representing the **best**, **worst** and **average** fitness within the population at each generation. Results are taken from an average of 10 simulations for each particular size of architecture (with the

exception of the 100-brick simulation whose results are an average of 5 runs due to the time taken for each simulation).

The best solutions for the 20- and 30-brick architectures from a randomly-selected run are shown in Figure 8.12 for illustration.

8.6.1 Fitness Trends

Several trends become apparent through examination of the charts in Figure 8.11. By considering the average fitness through time, we can confirm that the genetic algorithm succeeds in increasing the fitness of the population. This important result confirms that our chosen representation and genetic operators allow for effective evolution towards better results. This trend is echoed by the worst fitness score in each generation, which typically decreases, although following a far less smooth curve than the average.

The high variability of the worst solution over time suggests that while our operators do allow for successful evolution, a high degree of variability in fitness can be caused by crossover, mutation, or both these processes. This is most likely an artefact of the smoothness (or lack of smoothness) of the mapping between gene strings and the brick orderings produced. Since each element within the gene string affects the contents of O and U, a single difference between gene strings can result in different contents of U at later stages, causing the remaining gene instructions to be interpreted differently. In this manner, a single mutation can result in a significantly different brick ordering, and thus the mapping between the gene string representation and the resulting orderings is not entirely smooth.

The fitness of the best solution, as was intended and can clearly be seen, never decreases. It is interesting to note that in those simulations with smaller architectures (10- and 20-brick structures in particular) the best fitness is present within the initial population, and is not bettered during the course of the experimental run. However, in simulations using 50- and finally 100-brick structures, the final best fitness is improved over the best solution in the initial population. In these larger-architecture simulations, the genetic algorithm's effectiveness can be more clearly seen.

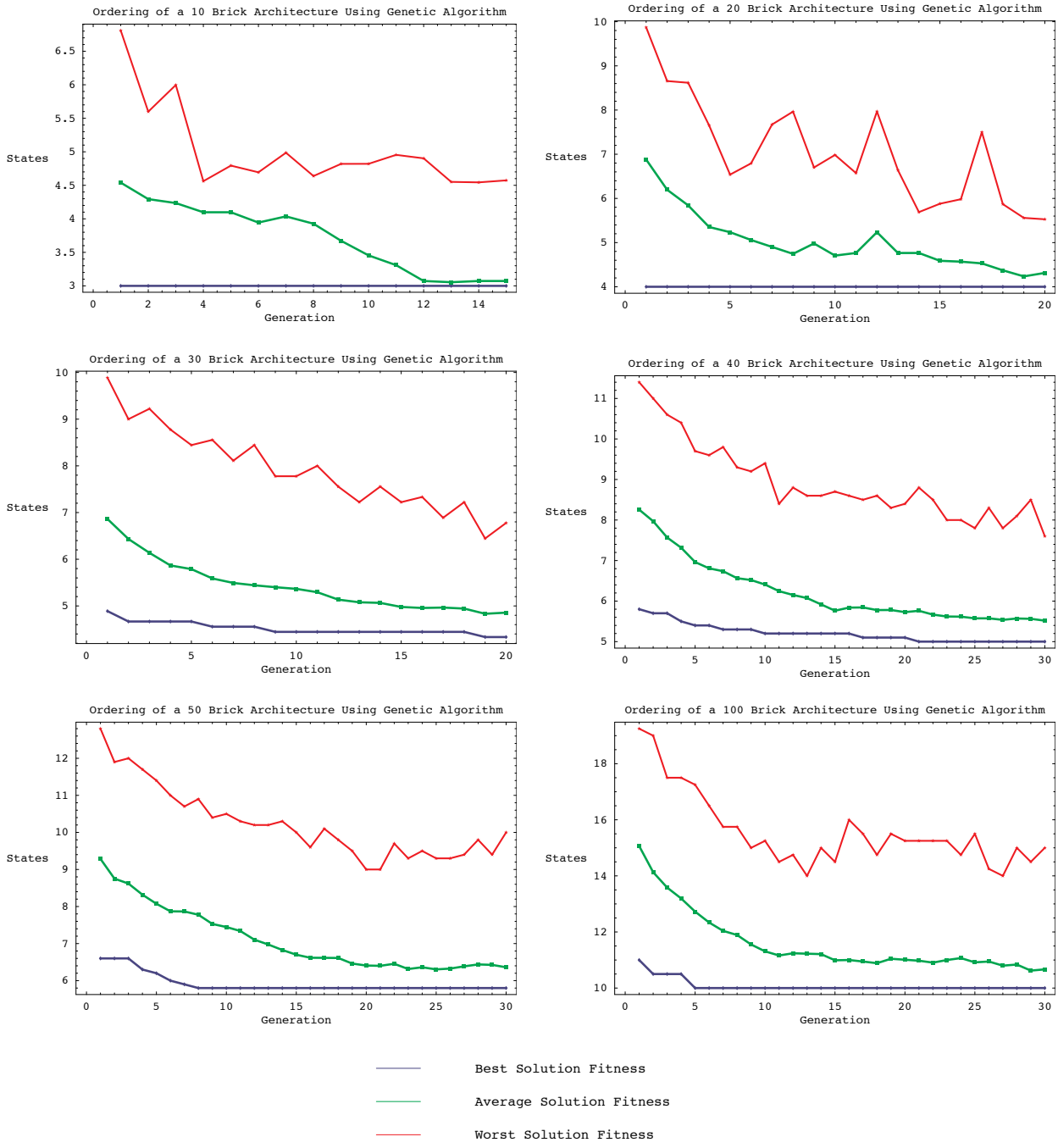


Figure 8.11: Experimental results using a genetic algorithm to evolve brick orderings.

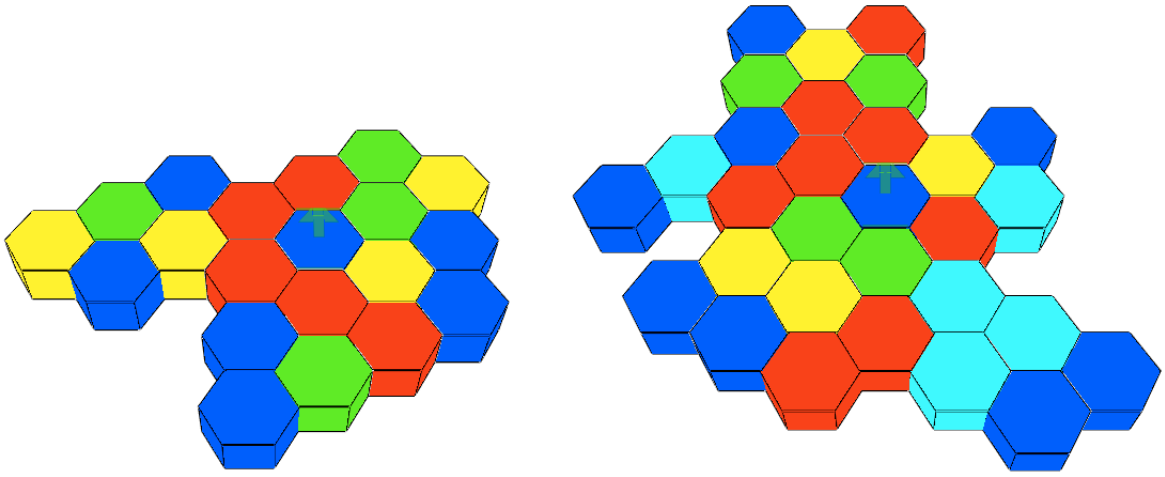


Figure 8.12: Examples of 20- and 30-brick randomly generated architectures, after ordering using the genetic algorithm and brick colour assignment by the Increasing States algorithm.

8.6.2 Experimental Parameter Selection

The crossover probability was set at 0.9 to encourage a thorough mixing of solution material, whilst letting some adequate or even high-fitness solutions enter the next generation unmodified. Finally, mutation occurs with a relatively low probability, as is traditional in most GA experiments, since mutation may have a significant detrimental effect on the fitness of a solution.

These parameter values were selected after a brief series of test runs in order to ascertain the viability of this approach as a whole. The precise values do not significantly impact the performance of the genetic algorithm when operating on architectures of this small size, although it was clear that higher values of mutation adversely affected the worst solution fitness. Larger architectures, on the other hand, may benefit from hand-tuning of these parameters in order to improve the genetic algorithm's performance and convergence of fitness. However, since the optimisation of this process is not relevant to this investigation (only that such an approach is viable), such experimentation is left as future work.

8.7 Evaluation of Ordering Using Genetic Algorithms

It can now be seen that it is quite feasible to use genetic algorithm techniques to produce brick orderings which work towards optimising the number of brick colours required to build an architecture, when used in conjunction with the Increasing States algorithm described in the previous chapter. The selection of an appropriate representation for manipulation by the genetic algorithm process has been shown to be crucially important, as is true in related sequencing problems such as the Travelling Salesman Problem, and in Job-Shop Scheduling. By selecting an *indirect* method of encoding the brick order, expensive genetic repair operators have been avoided.

The results of this approach as a mechanism towards stigmergic algorithm extraction have also been presented, and can clearly be seen to be effective. However, the results also highlight several problems. The crossover and mutation operators can still have significantly destructive results on the fitness of the solutions on which they operate. More inspiration can certainly be drawn from the extensive work in the area of Job-Shop Scheduling[91] to deal with this.

8.8 Algorithm Extraction – Summary Review

The two steps required to extract a stigmergic algorithm were outlined as below in Sections 6.5:

1. Assign a *valid* order to the bricks within the architecture
2. Assign a colour to each brick within the architecture

We have now presented solutions to both of these component problems, and thus have to a significant extent reached the goal of both the original work by Bonabeau, Theraulaz et al., and the continuation of those aims outlined in Chapter 5 and Chapter 6 here. Given *any* structure (within the constraints of the original lattice system) it is possible to:

- Extract a stigmergic algorithm;
- Determine the minimal number of brick colours required by that algorithm;

- Guarantee that the desired structure will be reproduced perfectly.

However, as will be seen in the remainder of this work, optimisation of this particular genetic algorithm process is not the biggest obstacle in achieving the *original* goal of this research: to produce *minimal* stigmergic algorithms capable of constructing arbitrary structures.

8.9 Limitations of Increasing States and Ordering

Only one aspect of this solution seems to be lacking: the number of rules in an extracted algorithm remains equal to the number of bricks in the original architecture (see Section 6.5.1). To further illustrate, by considering the 6-brick ring structure in Figure 6.13, the extraction process we have developed will at best produce an algorithm equivalent to $\mathbf{a}^1$ in that figure. However, algorithm $\mathbf{a}^2$ in that same figure depicts an algorithm which uses only 5 rules to build the same structure – the number of rules $|R|$ is less than the number of bricks built $|B|$, and the resulting algorithm is therefore smaller, simpler, and more efficient at describing the construction of the output structure.

As $|R|$ increases for more complex structures, the simplicity of an extracted stigmergic solution is compromised. An agent which must recognise thousands of different local configurations of building material has arguably ceased to be simple, and there may be a threshold where the other benefits of a distributed solution (as described in Section 1.2.2) no longer outweigh the complexity of the control system within each individual agent required to distinguish between a larger number of local configurations.

In other words, it is clear that a significant portion of the perceived power underlying stigmergic solutions is derived from the production of seemingly complex structures from *obviously simple* rule sets, and thus producing stigmergic algorithms with a number of rules less than the number of bricks in the resulting architecture is a highly desirable goal.

One means to produce algorithms where $|R| < |B|$ will now be considered: the identification of smaller, repeated sub-structures within an architecture, and the exploitation of this repetition to reduce the total number of rules required to build the structure. The introduc-

tion of repetition creates new problems and conflicts within the algorithm regarding control of building behaviour. The production of algorithms in which $|R| < |B|$, and implications of allowing the repeated firing of rules during simulation, are the subjects of the following chapter.

Chapter 9

Pattern-based Ordering

> "Therefore, if one wants a swarm of agents to build a given architecture, one has to decompose it into a finite number of building steps, with the necessary condition that the local configurations that are created by a given state and which trigger building actions, differ from those created by a previous or a forthcoming building step so as to avoid the dis-organization of the building activity." [156]

Overview

The previous chapters have presented an effective and tractable method of extracting a stigmergic algorithm from an existing architecture. However, in every algorithm produced by this process the number of rules in the resulting algorithm is exactly equal to the number of bricks in the original structure.

In this chapter, the use of repeated substructures or *modules* is considered as a means to reduce the number of rules in the extracted stigmergic algorithm. The consequences of construction using repeated modules are considered in detail, and finally the feasibility of the process is evaluated.

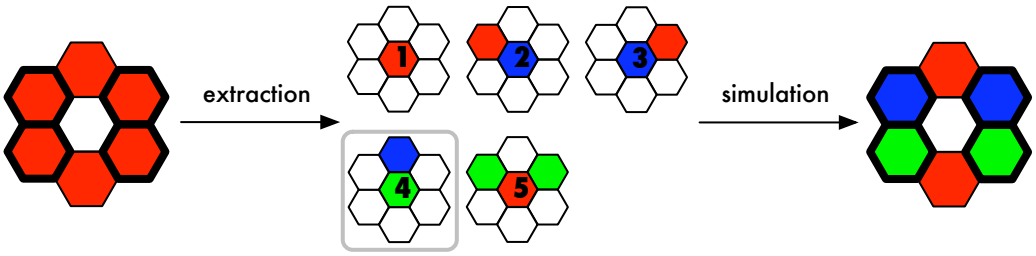


Figure 9.1: A stigmergic algorithm in which $|R| < |B|$, taken from Figure 6.13. Rule 4 in this algorithm fires twice, at separate locations, to produce the two, short vertical columns highlighted in the final construction on the right.

9.1 Patterns and Substructures

The simplest consequence of attempting to produce an algorithm where the number of rules in the algorithm, $|R|$, is less than the number of bricks in the input structure, $|B|$, is that in order to build more bricks than you have rules, one or more rules must fire more than once.

This is a crucial point which deserves to be repeated: if in *any* way it is possible to produce an algorithm with $|R| < |B|$, it is *necessary* that one or more rules within the algorithm will fire at least twice during the simulation. Considering the basic nature of the simulation system, a rule will fire when a particular stimulating configuration is present within the environment. For the same rule to fire more than once, the same stimulating configuration must therefore be created within the environment more than once.

9.1.1 A Simple, Minimal Example

In Figure 9.1, we can see a simple algorithm which produces a six-brick ring structure, using only *five* rules. The highlighted rule fires twice during the course of simulation, producing the two green bricks which the final rule eventually matches against. The extraction processes described previously would always produce stigmergic algorithms with six rules, given that $|R| = |B|$ in that process. While the reduction from six to five rules might not seem significant, this new five-rule algorithm is now minimal[1], and cannot be optimised further. The production of *minimal* algorithms is clearly the most valuable potential outcome of this research, so any method which might reduce algorithm size should be considered.

[1] See section 9.8.1.

The key to the production of this minimal algorithm lies in the identification of *patterns* or *substructures* which are present more than once within the architecture. Within the ring structure in Figure 9.1, the repeated substructure of a short column of two bricks has been highlighted. Since this small structure appears identically twice in the overall construction, it can be built using the same set of rules (or single rule – rule 4 – in this simple case).

9.1.2 Repeating Rules

The repeated execution of stigmergic rules has been considered previously in this thesis (see Section 5.5.4), and in particular the 'problem' of *self-activating* rules. These rules, given the correct local environment, can fire more than once in direct succession. This type of repetition contrasts with that demonstrated in the simple example above, where a single rule fires in two distinct neighbourhoods, and fires only once at each point.

This is illustrated in Figure 9.2, using the two rules from figures 5.6 and 5.7 in Chapter 5. The situation presented on the left is an example of *independent* repetition, similar to that given in Figure 9.1. Rule A will fire *only* in positions $\mathbf{X}^1$ and $\mathbf{X}^2$. These two positions must be in distinct neighbourhoods; i.e. $\mathbf{X}^1$ must be a cell which is not included in the neighbourhood of $\mathbf{X}^2$. If this is not the case, the neighbourhood for $\mathbf{X}^2$ will be modified by the new brick in $\mathbf{X}^1$, and thus the rule will not match in this modified neighbourhood.

Rule B is the archetypal *self-activating* rule, which in this form builds a straight line of cells. In contrast with *independent* repetition, each subsequent cell in which the rule fires is a part of the neighbourhood of the previous. To clarify, the distinction between independent and self-activating repetition is a consequence of the positions in which the rule can fire (if it can fire at all) immediately after it has itself fired. If the rule can place a brick somewhere within the neighbourhood of the brick it originally placed, the rule is *self-activating*, since it is therefore a post-rule of itself.

If a rule *is* repeated, either the subsequent firing occurs directly after the previous one, and within the same neighbourhood, and is therefore a *self-activating* repetition, otherwise it is then an example of *independent* repetition.

Just as the repetition of rules can be divided into two clear categories, we can consider

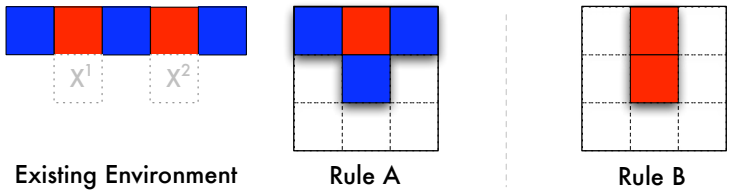


Figure 9.2: Two types of repetition in simple rule systems. Rule A will fire once at site $\mathbf{X}^1$, and then once again at site $\mathbf{X}^2$. Rule B is *self-activating*, and will fire repeatedly until stopped.

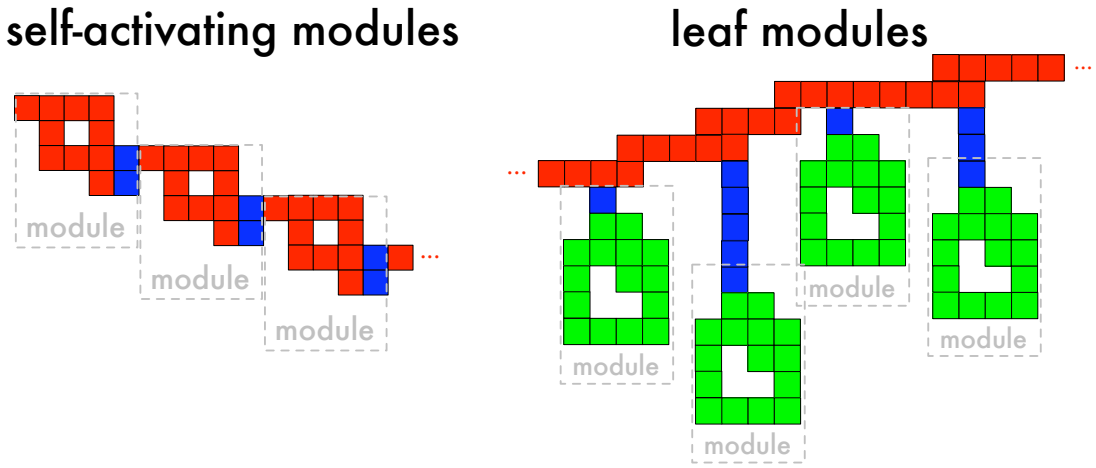


Figure 9.3: Two types of architecture module repetition: *self-activation* and *stem-and-leaf*.

the construction of larger repeated units – *modules* – in a similar manner.

9.1.3 Modularity and Types of Repetition

The stimulating configuration which triggers the repeated production of a substructure or **module** in an architecture may be produced either as a direct result of the construction of the repeated unit, or by construction external to this module. This is illustrated in Figure 9.3. These two types of repetition may be termed *self-activating* and **stem-and-leaf** repetition.

Self-Activating Repetition

As can be seen in Figure 9.3, self-activating modules are built in a chain, with one modular substructure built directly onto the previous one. An important consequence of this form of repetition is that in the absence of any external structures (see Section 9.7) the construction

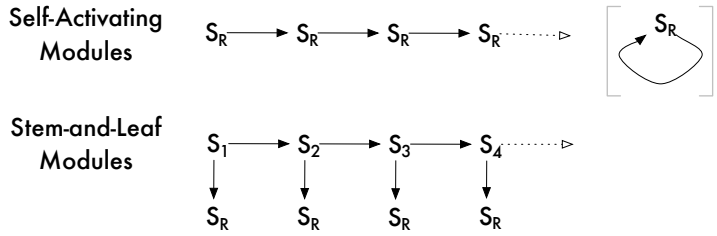


Figure 9.4: Self-activating and stem-and-leaf modular construction shown as building states. $\mathbf{S}_R$ indicates the particular module/state which is repeated.

of such chains of modules cannot be controlled: the sequence of modules will be built forever.

Repetition of this type allows 'coherent' structures of infinite size (without the bounds of the simulated space) to be produced from a stigmergic algorithm of fixed size. The construction cannot be halted unless external structures are present in the environment or a more sophisticated agent behavioural system is used. Controlling this type of repetition is discussed further in Section 9.2.1.

Stem-and-Leaf Repetition

In contrast to *self-activating* modules, leaf modules do not fire continuously. Instead, the stimulating configurations which trigger the construction of a leaf module are generated by a separate set of stigmergic rules. This is illustrated on the right in Figure 9.3.

9.1.4 From Repeated Modules to Building Stages

These two types of modular construction are distinguished by the generation (or lack of generation, in the case of *leaf* modules) of stimulating configurations which will initiate the repeated construction of another identical module. This exhibits itself during the simulation as a series of cycles within the construction graph (Section 3.4.1). If the types of modular repetition are visualised in a similar style, shown in Figure 9.4, the difference between self-activating and stem-and-leaf repetition is clearly presented.

Just as two types of modular construction have now been identified, two distinct behavioural patterns of *coordinated algorithms* were asserted in [158, 156] (and discussed in Section 5.1):

"Such [building] states are at the root of the modular structures that appear in the architecture. One may have the two following cases (where R_i denotes the set of responses generated in state S_i):

a strictly linear chain of building states:

$$S_1 \xrightarrow{R_1\downarrow} S_2 \xrightarrow{R_2\downarrow} S_3 \xrightarrow{R_3\downarrow} S_4$$

a chain of building states that can have recurrent states:

$$S_1 \underset{R_4\uparrow}{\overset{R_1\downarrow}{\longrightarrow} S_2 \overset{R_2\downarrow}{\longrightarrow} S_3 \overset{R_3\downarrow}{\longrightarrow}} S_4.$$"

These state transition diagrams do not correspond directly to the two types of modularity depicted in Figure 9.4. This may account for the issues in clarity raised in Section 5.1 regarding these building behaviour descriptions. In this section, the concept of a 'building state' was discussed in detail, and the definitions given in [158, 156] found to be too vague to be useful for the 'bottom-up' stigmergic algorithm generation process adopted originally by Theraulaz and Bonabeau.

However, the aspect of *modularity* indicated by Theraulaz and Bonabeau's 'building states' now appears be central in the 'top-down' extraction of compact ($|R| < |B|$) stigmergic algorithms.

9.1.5 Architecture Construction using Repetition

The repetition of larger structures can be divided into two distinct types, as shown in Section 9.1.2; however, it should **not** be inferred that the construction of a whole architecture must be achieved by only one of these types. Both types can exist together comfortably, as shown in Figure 9.5. In this example, the red central spine is built using a single, self-activating rule, while the blue and yellow modules are repeated along the spine as leaf modules.

Clearly, self-activating repetition can be extremely useful, and may co-exist as a complementary mechanism with stem-and-leaf repetition (and certainly with other rules which may not repeat at all) for efficiently generating a structure. However, as is now obvious

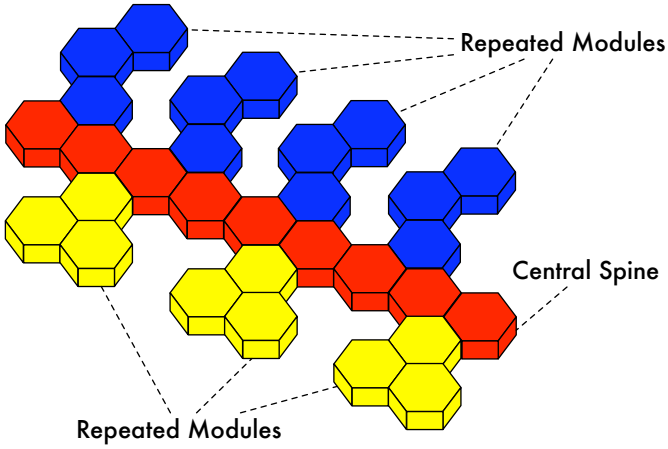


Figure 9.5: A simple architecture demonstrating both *self-activating* and *independent* or *stem-and-leaf* repetition using during construction.

the central spine of the structure shown in Figure 9.5 will grow to be infinite in length in the absence of external factors such as existing structures within the environment, or user intervention.

To effectively exploit repetition in an algorithm which must *accurately* construct an architecture, some means for controlling the behaviour of such rules must be found. The issue of accurate construction with regards to minimising algorithm size therefore becomes an important factor to consider in this research.

9.2 Accuracy and Control in Architecture Construction

Numerous example systems have been previously presented in which construction continues until stopped by the user. The most notable of these are the single-rule systems discussed in Section 5.3.1, which are examples of what is now termed *self-activating* repetition. If stigmergic construction techniques are to be successfully applied in arbitrary, non-biological situations, it is almost certain that the production of structures of exact sizes will be required, and therefore if the repetition of architectural modules is to be employed in order to produce more compact stigmergic algorithms, some mechanisms must be devised to *control* this repetition, particularly modules and rules which are self-activating.

9.2.1 Limiting Stigmergic Construction in *Nest*

As noted in [29, 52, 151], a significant problem with qualitative models of stigmergy such as that being examined here is that they fail to provide mechanisms which define how construction ceases. Two methods of inhibiting construction are suggested in [29]:

- Additional *meta-rules* which stop construction, governed by external factors such as swarm population size or various other functional parameters.
- The sequence of construction steps results in a final architecture with a complete absence of stimulating configurations matching any rules within the stigmergic algorithm.

An additional mechanism which is fairly obvious involves the use of quantitative environmental cues, such as pheromones:

- Rules are only matched if there is some sufficient strength of an environmental factor which exhibits a decreasing-strength gradient as the distance from the source increases.

While the addition of a pheromone-like[2] element is simple to implement, it is beyond the scope of the simple model used here, which must remain purely qualitative if prediction of rule firing (via post-rules, for instance) is required. The *meta-rule* method is far beyond the scope of the simple model considered here: the functional simulation of a structure requires a vastly more complex behavioural model for the agents, and almost certainly the gross abstraction of structure employed here would not sufficiently support such detailed simulation. Instead, a pure qualitatively-stigmergic solution must be derived if the current model is to be successfully exploited.

9.2.2 Precisely-Sized Structures

The dimensions of a structure can be defined in two ways – using absolute terms, or relative to existing external structures. Considering the latter, such a structure might be defined

[2]While pheromones represent the most obvious form of quantitative stigmergy, many other forms can exist. Behaviour may be regulated by light intensity (greater activity in regions where there is little material between the agents and the outside world), temperature gradients, or concentration of carbon dioxide[40, 121]. This is discussed further Section 10.2.1.

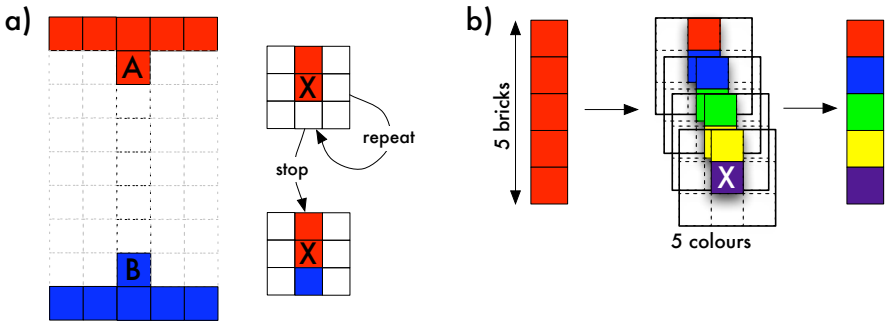


Figure 9.6: Two methods of limiting the size of constructed architectures: external structures and discrete brick colouring.

as "build a column of bricks from existing point **A** to existing point **B**", or some similar expression which ties construction to objects *already present in the environment*. This is demonstrated by part **a)** in Figure 9.6, where a 'bridge' must be built between two existing structures. Only two rules are required in this particular instance – one which builds the bridge itself, and a final rule to build the brick joining the bridge to 'structure **B**'.

Independent Structural Measurement

However, in the simple system we are considering, the environment is initially empty, and so all constraints must be defined without reference to external structures: in *isolation*. A trivial example is the construction of a simple column of *exactly* 5 bricks (as illustrated in part **b)** of Figure 9.6). To successfully achieve this goal, the agents must have some method of measuring the construction so as to ensure they neither fail to complete the architecture, nor build more than 5 bricks.

Brick Colours for Measurement

An agent's perception is limited to the size of a single neighbourhood, which in this particular system only includes the bricks immediately adjacent to the agent's current position. Since an agent cannot detect the presence of bricks beyond this boundary, and lacks any internal state to compensate, it is unable to measure architectural features of a size greater than or equal to the size of a neighbourhood in any direction. This is illustrated in Figure 9.7, wherein the agent cannot know if the row of bricks it is moving along consists of exactly, or

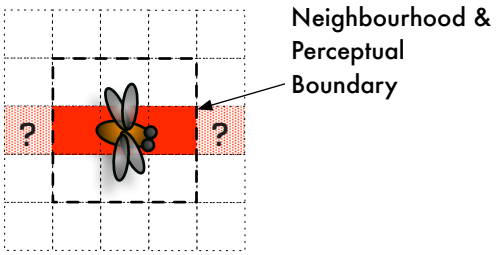


Figure 9.7: The size of an agent's perceptual area is limited to a single neighbourhood as defined by the abstract stigmergic system. The agent therefore cannot determine the size of structural elements equal or greater than the neighbourhood size.

more than, three bricks.

Since the agent has no means to internally 'count' the number of bricks in the structure, this information must be stored in the state of the environment itself; the agent needs some method of indicating that the current brick is the n^{th} brick in the sequence. As indicated in part **b)** of Figure 9.6, the means available for the agent to store this information in the environment is using different brick colours. In this example, the 'blue' brick is the 2^{nd} brick, the 'green brick the 3^{rd} and so on. Since no rule matches against a 'purple' brick, construction halts and the required 5-brick architecture has been precisely replicated.

9.2.3 Minimal Brick Colours and Accurate Replication

A simple conclusion to take from part **b)** of Figure 9.6 would be as follows: In order to build an architectural feature of length n, without the existence of external structures, n brick colours are required.

However, by initiating the construction of structures at a central point and building outwards, rather than starting at an extremity. By doing this, the maximum length agents must measure is half the total length of the structure. For example, the minimum number of brick colours required to build a column of n is reduced from n to $n/2$ (rounded up to the nearest integer). This is illustrated in Figure 9.8. We can therefore revise our assertion to become: **In order to build an architectural feature of length n, without the existence of external structures, *at least* $n/2$ brick colours are required.**

This assertion is further supported by the minimum number of brick colours found by

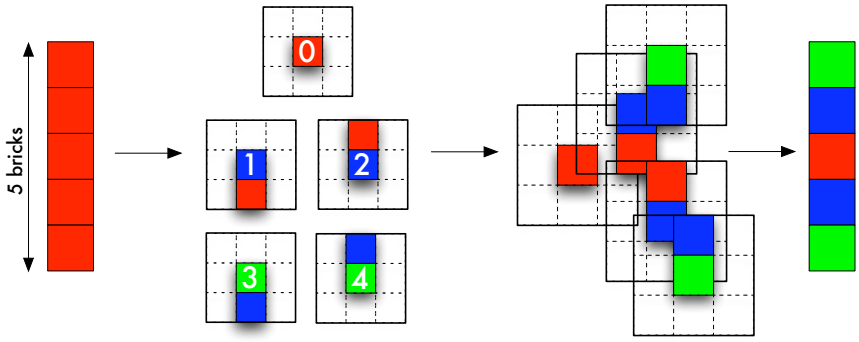


Figure 9.8: Exploiting the exclusion of rotation to reduce the number of brick colours required to build a column.

the genetic algorithm (GA) in Section 8.6. The 50-brick structure whose order was evolved previously is shown in Figure 9.9. The maximum span of this structure is 9 bricks, and so $n = 9$. The minimum number of brick colours found by the GA was 6, which lies between n and $n/2$. It should not be expected that the GA would find an algorithm where the minimum number of brick colours is actually $n/2$, since the other features of the structure might require additional colours to be introduced.

9.3 Accurate Construction with Repeating Modules

In the previous section, use of brick colours to allow agents to "measure" architectural features was introduced. For instance, to build a column or line of bricks of an absolute length n then $n/2$ brick colours are required. This same technique can be applied to the construction of repeated units, as shown in Figure 9.10. In this architecture, a leaf module is required every three bricks along a column structure. This is achieved by ensuring that the particular stimulating configuration which initiates the leaf module (Rule **D**) appears at this regularly repeating interval.

The repetition of this configuration is created by splitting the column structure into repeating units of three bricks each, with the deposit of the final green brick triggering (Rule **C**) the construction of another three-brick column segment. In this manner, the differing brick colours along the column are used to measure the correct interval for the placement of leaf modules, as the construction progresses.

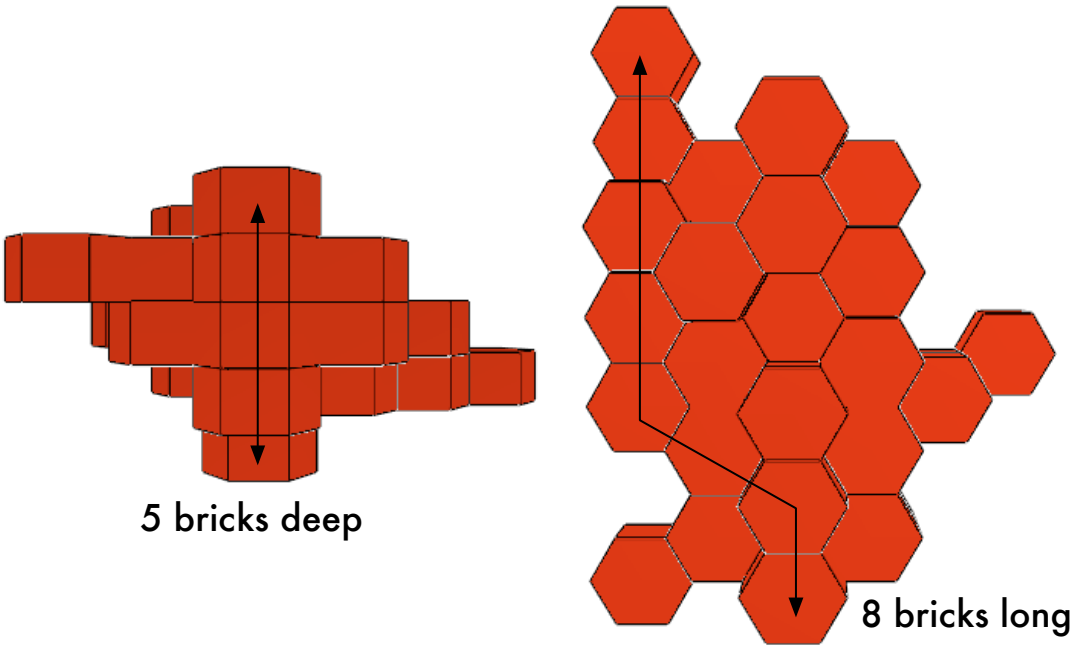


Figure 9.9: Dimensions of the structure for which an ordering was evolved in Section 8.6 (see also Figure 8.11).

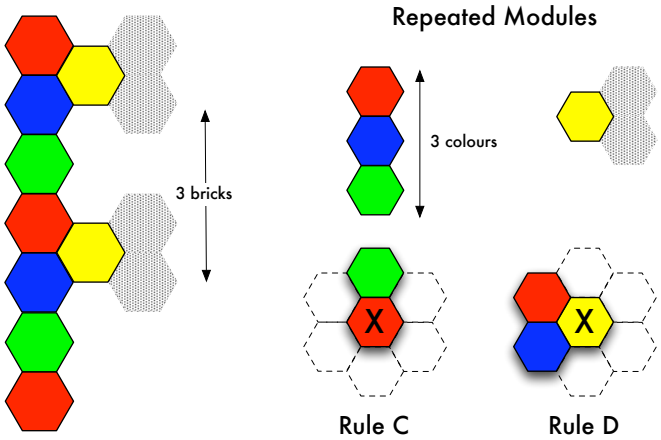


Figure 9.10: A simple structure featuring the controlled repetition of modules.

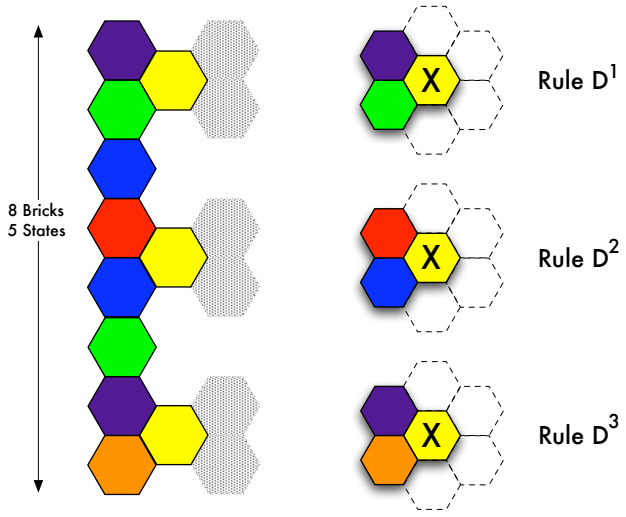


Figure 9.11: Controlled, limited repetition of leaf modules.

9.3.1 Local vs. Global Measurement

The example above demonstrates the use of repeated modules to accurately build features at given intervals during the construction of an architecture, and the use of brick colour as an additional tool to effectively 'measure' the relative distance between these modules.

However, further consideration of the algorithm outlined above will show that while the repeating column unit controls the placement of the leaf modules initiated by Rule **D**, the column module itself is a self-activating module. In the absence of external factors an infinite series of leaf modules will be built, each spaced 3 bricks apart on an infinite-length column of bricks. While the brick colouring on the column introduces some local measurement and control, there remains no means to limit the *number of repetitions* of this unit, and therefore no mechanism to limit the construction of the overall architecture in this case.

The *only* way to guarantee an arbitrarily-sized structure in the presence of self-activating rules or modules is by ensuring the presence of some external absolute-sized structure to halt the cycle of self-activated construction. The limited-construction form of the stigmergic system presented in Figure 9.10 is shown in Figure 9.11. In order to halt the construction of leaf modules, the central spine is built with a limited length, using the minimum number of brick colours possible (see Section 9.2.3).

9.3.2 Accurate Construction With Repeated Modules – Summary

We have now seen that allowing the repeated firing of rules during construction using a stigmergic algorithm allows for a reduction in the algorithm size, but the potential for loss of control during construction is also significant. The presence of self-activating rules and modules must be countered by the existence (or construction) of precisely-sized structures to inhibit the further replication of this structural features. In light of the need for some globally-measured structure of absolute size, the assertion made in Section 9.2.3 still appears to hold true: **in order to build an architectural feature of length n, without external intervention, *at least* $n/2$ brick colours are required**.

9.4 Pattern-based Rule Extraction from Existing Structures

Thus far the separation of architectures into sub-units or modules has been considered from the perspective of reconstruction. It was shown that it is possible to build an architecture using an algorithm with fewer rules than bricks, if the repeated firing of rules is appropriately managed.

What remains to be investigated is the feasibility of the automatic determination of these structural modules from an existing structure, and the integration of any such process into the existing algorithm extraction technique presented in previous sections. The original process is split into two stages, as below:

1. Extract rules from the structure by ordering bricks.

2. Assign colours to each rule using the Increasing States Algorithm (see Chapter 7).

The state assignment process operates only on a set of rules, without concern as to how those rules were obtained, so consideration of how it might accommodate controlled repetition can be postponed for the moment. The number of rules within an algorithm is determined wholly by the rule extraction process, which up to this point has consisted of a simple ordering of the structure's bricks. This must now be augmented as follows:

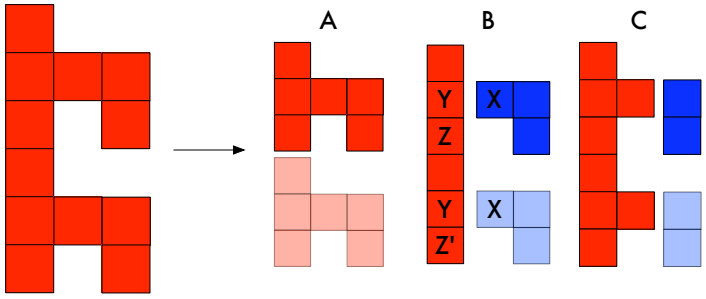


Figure 9.12: A simple architecture, and possible divisions into patterns.

1. Extract rules from the structure by ordering bricks:

   (a) **Identify and divide the architecture into sub-architectures, some of which may be repeated.**

   (b) **For any modules which are repeated, retain only one instance.**

   (c) **Order the bricks within each module, and then extract rules for the final algorithm from all modules.**

2. Assign colours to each rule using the Increasing States Algorithm.

A simple example of how such a process could work is presented below, after which the construction of an algorithmic process which might automate the process is considered.

9.4.1 A Simple Example of Pattern-based Rule Extraction

In order to illustrate how pattern-based rule extraction might operate, a simple example is now presented, using the simple structure in Figure 9.12. The first step in the rule extraction procedure outlined above is the identification of potential modules.

Module Selection

Three likely deconstructions are presented to the right of the simple structure. The first and most obvious, deconstruction **A**, splits the architecture into a single module, with a repetition directly below. However, such a module would be self-activating, and construction would repeat forever. This is clearly undesirable, and so another decomposition must be found.

Set **B** splits the architecture into a simple column with a repeated 'L' leaf node. It has previously been shown that the construction of an absolute-length column is simple if enough brick colours are used, and by doing so this second deconstruction can avoid the infinite-building problems of self-activating modules.

The rule which builds both bricks marked **X** in set **B** (shown below the modules) must fire twice along the column, matching against brick **Y**. This means that the same configuration of bricks and brick colours must be produced at each of these points. As indicated in the figure, the first brick **Y** is proceeded by brick **Z**, and the second **Y** by brick **Z′**.

For the same stimulating configuration, including brick **Y** and any neighbouring bricks which are already present, to be produced at two points along an effectively one-dimensional column, the column itself must be built using rules which will repeatedly produce bricks matching **Y** along its length. In other words, the column must be built using a self-activating module, and therefore brick **Y** will appear many times along the resulting column's length, and the same number of leaf modules will be built. Using this set of modules will not result in accurate construction of the original architecture.

The final decomposition (**C**) requires the construction of a column with 'spurs' along its length at each point where leaf module should be built. These spurs can be built at specific points along the column by using distinct rules for each spur, which only match against a single location along the column's length. However, these spur rules will deposit an identical-coloured brick as the spur itself. The leaf module can be assembled by rules which match against any such brick.

Rule Extraction

Having selected a set of modules, a set of rules must be extracted which can then be processed by the Increasing States algorithm. This process is shown in Figure 9.13. Firstly, since each instance of a module will be built using the same rules, we can retain only one instance of each unique module. In this example, only two distinct modules are present – the column with stubs, and the leaf, which is repeated twice. The second instance of the leaf module is therefore discarded.

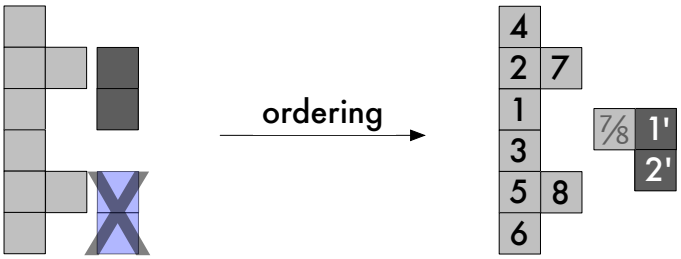


Figure 9.13: Ordering and state assignment of the selected modular deconstruction of the simple architecture presented in Figure 9.13.

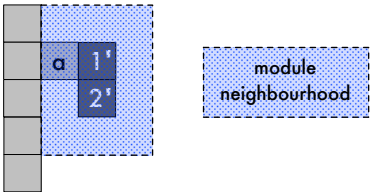


Figure 9.14: Module neighbourhoods must be considered when assigning brick colours.

At this point, each of the bricks in the modules under consideration will be built by a distinct rule. It is then simple to order the bricks in each module as described in previous chapters (either randomly or using a genetic algorithm), resulting in the orderings shown in the centre of Figure 9.13.

State Assignment, Repetition and Brick Colour Constraints

The procedure now becomes more complicated than previous post-ordering rule extraction. While the leaf module outlined in Figure 9.12 only contained two bricks, the first brick in any module is never built in isolation[3], so any surrounding bricks from other modules must be included as part of the rule extraction process. This is shown in Figure 9.14. Each brick which is within the combined neighbourhoods of all bricks within the module must be considered when the rules for that module are extracted.

Furthermore, since this module must match in more than one region of the architecture, the module's neighbourhood must correspond to more than one neighbourhood of bricks in the overall architecture. Brick **a** in Figure 9.14 must correspond to both brick **7** and brick **8** in the architecture shown in Figure 9.13. This is shown as the inclusion of brick **7/8** in the

[3]Bricks are never placed in isolation, with the exception of the very first brick to be placed.

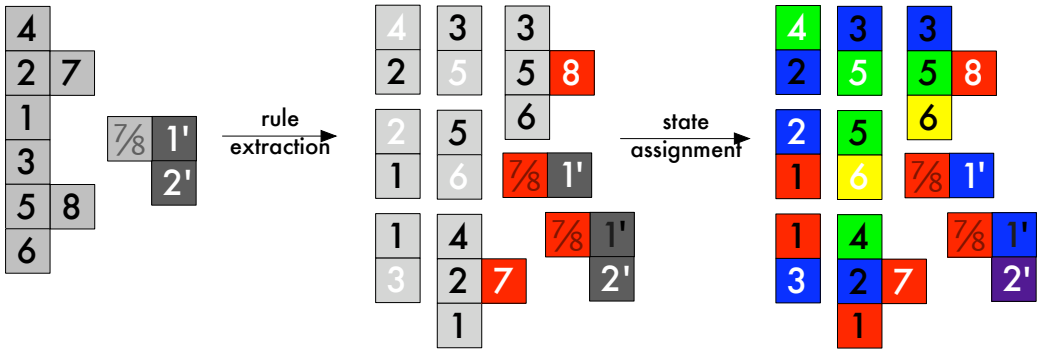


Figure 9.15: State assignment of extract rules, highlighting linked bricks.

leaf module, indicating that the rule extracted for brick **1′** in the leaf module must match against both brick **7** and **8** during construction.

The consequence of brick **a**'s correspondence to *both* bricks **7** and **8** is that these two bricks must share the same colour if rule **1′** is to match against both of them. If these bricks are different colours, the neighbourhoods where rule **1′** should fire cannot be identical, and so two rules would be required to build brick **1′** in each location. This clearly undermines the effort to reduce the total number of rules in the resulting stigmergic algorithm.

The complete process is shown in Figure 9.15. The first transition shows the individual rules as defined by the ordering presented in Figure 9.13. To ensure that bricks **7** and **8** are the same colour, they are both assigned the colour `RED`, after which state assignment is performed using the Increasing States algorithm. The resulting algorithm accurately builds an architecture of 12 bricks using 9 rules and 5 brick colours.

9.4.2 Automatic Pattern-based Rule Extraction

The section above demonstrated the possibility of producing a stigmergic algorithm based on the deconstruction of an architecture into modules or sub-architectures, but it is far from automatic. Several steps are not obviously reducible to an algorithmic process. The first and most obvious is the division of the architecture into modules by automatic means – can the architecture be split in a way which minimises the size of the extracted stigmergic algorithm? Additionally, the manner in which colours are assigned to the bricks connecting one module to another (the spur bricks, or module neighbours, above for instance) must be clarified and

clearly explained. Finally, the apparent requirement that self-activating modules must be avoided will be investigated. Each of these issues is now considered below.

9.5 Automatic Identification of Structural Patterns

The identification of repeating structures seems trivial for the human observer, but it is a significant problem to reproduce this ability within computer systems. Most research into automatic pattern detection takes place within the context of computer vision, where models of of relationships between objects within a scene must be determined by the analysis of image data.

In contrast, we already have a model of entity relationship in the graph of bricks. Each brick can be related to another by the statement of the direction in which the neighbouring brick lies. The statement of this relationship is clearly represented in Figure 4.13, presented during the discussion of the *Nest-3.0* software system. To recap, the data structure chosen to store hexagonal architectures maintains a *graph* linking each neighbouring brick. Each edge is also labelled with the direction in which the neighbouring brick lies. What is therefore required is some automatic means to search for non-overlapping repeated sub-graphs within this graph of bricks.

9.5.1 Substructure Discovery

By representing a structure as a graph, it is possible to take advantage of the significant body of existing research and literature regarding graphs[4]. A module or substructure within an architecture corresponds to a sub-graph within the entire graph of brick vertices. In traditional graph theory terminology, two modules which are structurally identical appear as two isomorphic sub-graphs. Two graphs are isomorphic if there is a one-to-one correspondence between their vertices and there is an edge between two vertices of one graph if and only if there is an edge between the two corresponding vertices in the other graph.

The problem of determining if a sub-graph is isomorphic to any other graph is famous,

[4]Excellent introductions to graph theory can be found in [79, 47].

and known to be NP-complete[5]. What differentiates the problem of isomorphic sub-graph identification within this particular context is that in the 'classic' problems, the edges are not labelled. Within a *Nest* structure graph, however, the direction in which one brick is located in relation to any others is significant. In this manner, each edge within the graph is labelled with the direction (`N`, `NE`, or `UP` for example) in which one brick is connected to the other.

The SUBDUE system developed by Cook and Holder[82, 83, 39, 81] employs a computationally constrained best-first search to perform substructure discovery on graphs with labelled edges. An example structure, taken from [83], is shown in Figure 9.16. On the left, a simple arrangement of shapes is presented with a trivial repeating unit. This structure is represented on the right by a directed, labelled graph in which the vertices represent each distinct component (T for triangle, S for square, R for rectangle, etc.), while the labelled edges between each vertex describe each shape's relationship. For instance, the edge marked `on` between `T1` and `S1` indicates that triangle 1 is on top of square 1. The most obvious repeating substructure is the simple 'house' shape formed by a triangle on a square. This is represented within the graph as a sub-graph of the form

$$(x : \mathrm{T}) \xrightarrow{\mathrm{on}} (y : \mathrm{S})$$

where x, y are the unique identifiers for two vertices within the graph.

The SUBDUE algorithm itself is outlined in Figure 9.17. The algorithm begins with a substructure matching only a single vertex within the structure graph. Until the amount of computation exceeds the given limit, each iteration selects the best substructures and enlarges each instance of these substructures by one neighbouring edge in all possible directions. The selection of 'better' substructures is performed by evaluating each substructure using a combination of four heuristics[83]:

Cognitive Savings – "... the net reduction in complexity after considering both the reduction in complexity of the input graph after replacing each instance of the substructure

[5]A comprehensive bibliography of papers considering this problem can be found at http://www.ics.uci.edu/~eppstein/bibs/subiso.bib

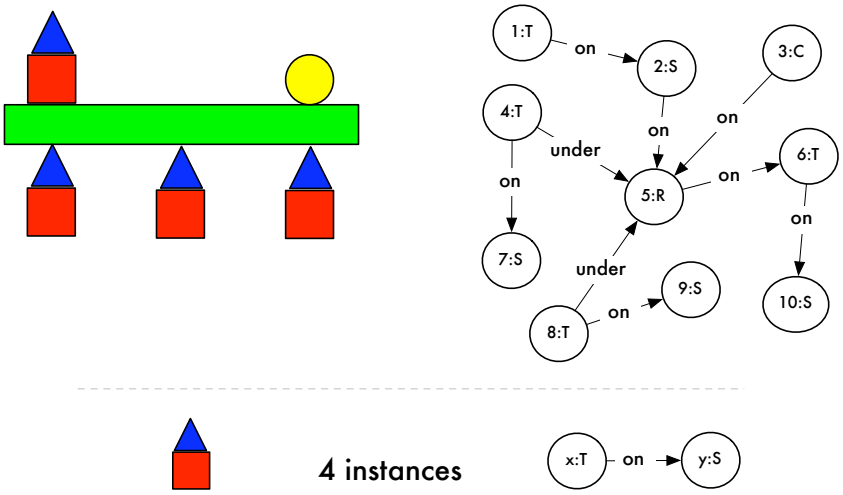


Figure 9.16: A simple configuration of objects on which SUBDUE can operate.

SUBDUE(G,*limit*,*beam*)
  $D = \{\}$
  $S =$ vertices(G)
  while (computation $<$ *limit*) and ($S \neq \{\}$)
    order S from best to worst using heuristic evaluation
    $b =$ first(S)
    $D = D \cap \{b\}$
    $E = \{b$ extended by one edge in all possible ways$\}$
    $S = S \cap E$
  return D

Figure 9.17: The SUBDUE substructure discovery algorithm (adapted from [81]).

by a single conceptual entity, and the gain in complexity associated with the conceptual definition of the new substructure."

Compactness – "...the ratio of the number of edges in the substructure to the number of nodes in the substructure."

Connectivity – inversely proportional to "...the amount of external connection in the instances of the substructure." In other words, "isolated" substructures are preferred over strongly connected ones.

Coverage – "...the fraction of structure in the input graph describe by the substructure."

A large proportion of the further work regarding the SUBDUE system focuses on fuzzy matching of sub-graphs. The goal of recent work by Holder and Cook[39, 81] is to providing

a compressed representation of the structure, by *iteratively* replacing identical sub-graphs with a single representative symbol. Where two sub-graphs are similar but not identical, the minimum number of transformations which will convert the first to the second can be stored – a *fuzzy* match – ultimately compressing the original structure into a hierarchical description of minimal length[138].

This type of transformation, from architecture to minimal-length description, seems very close to the ultimate aim of this research, which is to find minimal (or failing that, sufficiently small) stigmergic algorithms to produce given architectures. However, iterative substructure matching is not directly useful, because stigmergic algorithms are *not hierarchical*; there are no means within a *Nest* algorithm to specify a structure as an arrangement of repeating units, followed by a description of those units themselves. Instead, stigmergic algorithms are *flat*, with each rule as likely to fire at any time as any other rule within that algorithm.

Furthermore, there are no mechanisms within the *Nest* system which can arbitrarily apply changes to single instances of the same substructure *during construction*, in order to leverage any fuzzy pattern matching. If some substructure must be subjected to a number of transformations, it must be distinguishable from other instances within the architecture. However, repeated substructures produced by stigmergic algorithms are necessarily identical since the same rules are used to construct each instance.

Despite this, the core substructure discovery mechanism employed by SUBDUE remains a viable means for discovering possible repeated modules which might be used in the decomposition of an architecture into a modular stigmergic algorithm. More information regarding the SUBDUE system, along with executable and vastly extended versions of the algorithm, are available online at the University of Texas, Arlington,[6] at the time of writing.

9.5.2 The Application of SUBDUE to Stigmergic Algorithm Extraction

The SUBDUE graph of a very simple *Nest* structure is shown in Figure 9.18. The graph has been generated by including edges between two bricks for each neighbour link present

[6]SUBDUE can be found online at http://cygnus.uta.edu/subdue/

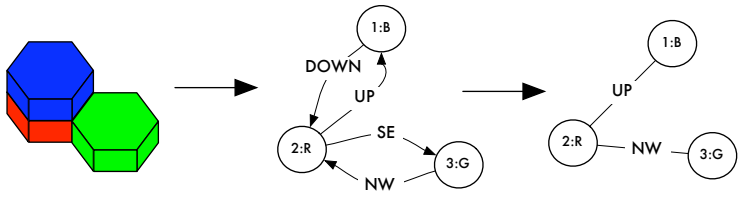


Figure 9.18: The subdue graph for a *Nest* architecture. Because the links are undirected, only half of the neighbour directions need be explicitly used – an `UP` link between two bricks *automatically implies* the corresponding `DOWN` link.

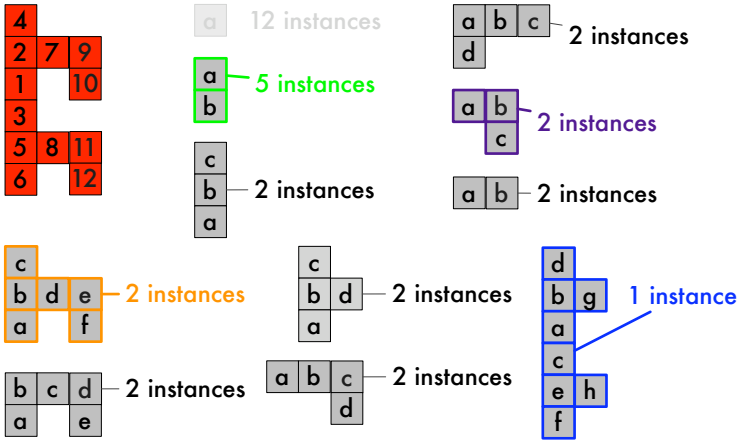


Figure 9.19: Structural patterns found by SUBDUE in the example architecture.

within the datastructure. It is immediately apparent that this graph is more complex than necessary: every vertex is joined by two edges, each in opposing-direction pairs. Rather than using a directed graph to represent a *Nest* structure, it is simpler to select a *sub-set* of the possibly neighbour directions and produce an undirected graph, as shown on the far right. The sub-set selected is arbitrary, but should include exactly one direction of each opposite pair (i.e. one of `UP/DOWN`, one of `N/S`, one of `NE/SW`, and so on).

The example architecture from Section 9.4.1 was processed using the SUBDUE algorithm. Nine patterns were found before the computational limit was reached (one second of processing time for this simple architecture), after limiting the minimum pattern size to be two vertices (bricks), and therefore removing the single-brick 'pattern'. Each of these patterns is shown in Figure 9.19.

The most common pattern is simply a short column of two bricks. This appears in the original architecture as bricks $(4, 2)$, $(1, 3)$, $(5, 6)$, $(9, 10)$ and $(11, 12)$. In the simple example,

two instances of this pattern were selected, ((9, 10) and (11, 12)), along with the only pattern with a single instance shown (highlighted in blue). Both instances of the pattern highlighted in orange fully cover the original architecture, and were the initial choice during the manual pattern identification above.

9.5.3 Pattern Selection and Sets of Patterns

Clearly, the SUBDUE system is very capable of identifying repeating structural patterns within the architecture graph. However, what this process does not produce is an analysis of the *relationship between patterns*. These relationships are crucially important for the accurate construction of the input architecture.

If it is decided that the patterns which are selected as the basis for decomposition of the architecture are those with the maximum number of instances within that architecture, then the small, two brick column (highlighted in green in Figure 9.19 would be chosen. While this particular pattern is suitable for the leaf nodes, using a small, repeating unit to build the long column requires the derived construction module to be assigned brick colours that would render it *self-activating* – a situation which will later be shown to be extremely undesirable (see Section 9.7).

If the patterns are selected on the basis of the amount of architecture covered by the sum of all instances of the pattern, the six-brick 'chair' (highlighted in orange) is the obvious candidate, and would appear to remove the need to consider any further patterns. However, the manual investigation of this problem reveals that because of the relationship between the two instances of this structural pattern, the resulting module must again be self-activating, and the produced architecture would again deviate from the input. Clearly there is much more to be considered than just the identification of possible repeating units within the structure.

9.5.4 Pattern Set Selection and Intractability

The experimentation and simple examples up to this point show that typically a mix of repeating modules and 'basic' rules[7] is required to successfully reproduce an architecture. It is the selection of this set of modules and individual rules which proves to be the fundamental problem facing any pattern-based ordering technique. As with any problem where "between 1 and x items must be selected from a set of size y", the complexity of this selection is the sum over a combinatorial. The relationships between this pattern and any other selected patterns must also be considered:

Is the pattern self-activating? – if so, can any other construction be used to constrain the repetition of this module? This is discussed further below in Section 9.7.

Does this pattern overlap with any other selected patterns? – since each brick should only be placed once, overlapping modules cannot be selected as part of the same algorithm.

How many instances of this pattern should be built? – as in the simple example, while the small two-brick pattern was used to build the leaf modules, it was not used to build the column *despite* its presence within that structure.

It is clear that without some mechanism to intelligently select patterns, this process rapidly becomes intractable. Unfortunately, further complications – outlined in the following sections – increase the complexity of this operation significantly. These problems will be summarised in Section 9.8.

9.6 Modular Overlap and Brick Colour Assignment

In the example presented in Figure 9.12, the leaf modules are connected to the stem by a single brick. By placing identical bricks along the spine module, multiple leaf modules can be constructed. In order to trigger the construction of a leaf module however, this brick

[7] 'Basic' rules in this context are those which build only a single, unique brick within the final structure, as would be extracting using the simple ordering techniques described in Chapter 8.

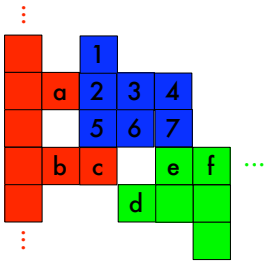


Figure 9.20: A more complex module neighbourhood. The ordering dependencies between multiple bricks and multiple modules considerably complicate the assignment of consistent colours to bricks within a module's neighbourhood.

must be identical with regards to both its colour and neighbourhood if the first rule of the leaf module is to fire at each location a leaf module should be built.

Ensuring that the shared elements of the spine module and leaf modules in Figure 9.12 are appropriately coloured is simple, but where there are a larger number of bricks in the overlapping neighbourhood – as shown in Figure 9.20 – arbitrarily assigning colours to such bricks will not satisfactorily resolve this dependency.

9.6.1 Modular Overlap and The *Increasing States* Algorithm

Brick colour assignment, as performed by the *Increasing States* algorithm (Section **??**), ensures the accurate construction of a particular structure by examining the underlying rules and guarantees that any undesired construction behaviour is eliminated.

However, there is no reason to assume that the *Increasing States* algorithm will assign identical colours to bricks **7** and **8** of the column structure shown in Figure 9.13. The probability of bricks **a**, **b** *and* **c** in Figure 9.20 being assigned identical colours each time they appear within the global architecture is even more remote.

The states for these bricks will be assigned to ensure only that the construction of the module, or rather the set of rules the *Increasing States* algorithm is operating upon, remains controlled and precise.

It does not take into account any dependencies between this set of rules and any other modules or architectures.

Assigning the colours of these bricks external to the *Increasing States* algorithm can, in

contrast, prevent the algorithm from successfully controlling the execution of the rules, since the external assignment does not consider the dependencies and post-rule links between the rules which will build those bricks, and the rest of the module.

9.6.2 Modular Overlap and Ordering

It was implicit in the simple example presented in Section 9.4.1 that construction of the column structure would begin before any of the leaf modules are built. However, this decision may not always be simple. The structural fragment shown in Figure 9.20 shows three modules, each with overlapping neighbourhoods. Because of this overlap, the rules for each module must be extracted according to the predicted presence (or absence) of any bricks in the overlap regions. In other words, the order in which *modules* are built is critical in the extraction of the rules for each of those modules.

Using Figure 9.20 as an example, when extracting the rules for bricks **1** and **2** it must be determined – using some hypothetical algorithmic process – if brick **a** has already been built, since these rules may match against the presence (or absence) of that brick. Similarly, the rule which builds **5** may match against bricks **a**, **b** and **c**, and so the existence of these bricks at the time rule **5** should fire must be similarly be discovered by this algorithm. In other words, the modules themselves must be ordered before rules can be extracted for individual bricks.

Modules and Ordering

The benefit gained by establishing a modular stigmergic algorithm lies in the fact that each module will be build using the same rules. Therefore, the ordering each brick is build is identical in all instances of that module. Consequently, once the ordering of bricks within a module is determined, it cannot be modified for any other instances of that module. This can present significant problems as the module extract process proceeds.

Consider the rules presented on the right in Figure 9.21. Multiple instances of the three modules originally shown in Figure 9.20 are now presented within the larger context of a structure, and these modules have been ordered using the hypothetical algorithm discussed

above. From these module orderings, it is implicit that:

- brick **c** is ordered after brick **d** (Rule 1) – that is, brick **c** will be present in the neighbourhood of the rule that builds brick **d** (see right of Figure 9.21);
- brick **6** is ordered after brick **c** (Rule 3);
- brick **e** is ordered after brick **6** (Rule 4).
- brick **d** is ordered after brick **e** (Rule 2);

Each rule is extracted using the relative orderings of *all* bricks within the neighbourhood to determine which bricks will be present when the brick in question is to be built. However, given this ordering, a cyclic dependency is created between the three modules which can never be satisfied. As modules 5, 6 and 7 are built each of the rules shown in Figure 9.21 must be applied. According to these rules, however, brick **c** must be present before brick **6** will be built, and brick **6** built before brick **e** can be placed. Brick **d** can only be placed after brick **e** (since **e** appears within the neighbourhood in the rule for brick **d**), and finally, the *original brick* – **c** – can only be built after brick **d** has been placed. Clearly this will lead to an effective 'grid-lock', and no further rules will be applied in this area.

This example shows that despite the attempted use of modularity, the global order of brick placement remains crucial in the accurate construction of an architecture. Attempting to algorithmically order the bricks within individual modules, and additionally manage the constraints and relationships which exist *between* modules, introduces a significantly complex new layer to the problem of pattern extraction. The ordering process can only consider bricks and their local neighbours: it cannot respect the notion of modules, since in essence a stigmergic algorithm is *flat*, with each rule treated as the *only* significant element.

9.7 Self-Activating Modules and Endless Construction

Self-activating modules – those substructures whose construction produces the same set of stimulating configurations as initiated the construction of the substructure originally – must

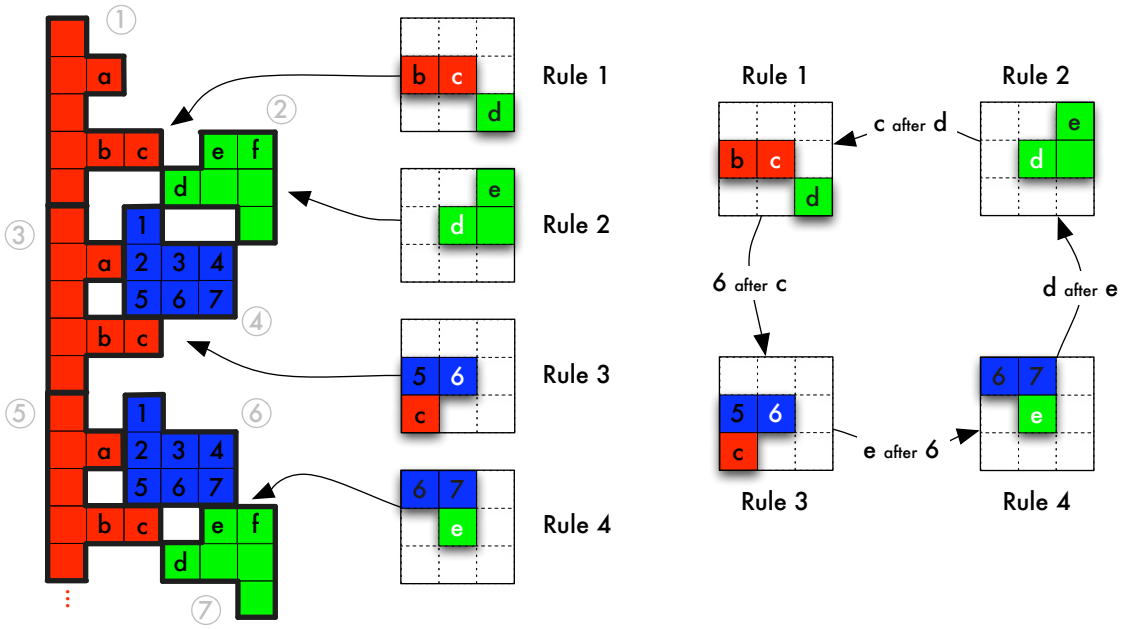


Figure 9.21: Ordering of the inner bricks within a module can be problematic if the relationship between modules is not consistent throughout the entire architecture.

be carefully considered if they are to be included in any stigmergic algorithm in which a specific architectural output is expected. It has been shown previously that without external intervention by the user, or in the absence of any other structure, self-activating rules or modules build forever. Clearly this is unacceptable if exact replication of a specific architecture is desired. In other words, the only occasion where a self-activating rule or module can be allowed is where one of these constraints exists. Since user interaction is outwith the specification of the *Nest* system, the latter must therefore be considered.

Figure 9.22 demonstrates an example situation in which a self-activating rule can be employed successfully, without jeopardising the precision of the construction process. In this construction, a self activating rule matches against brick **X**, and continues to build downwards until it encounters the *neighbourhood* of cell **Y**. In order to close the gap, an additional 'bridge' rule is required, since the rule must match against both the existing column of red bricks, and the newly-encountered blue brick below.

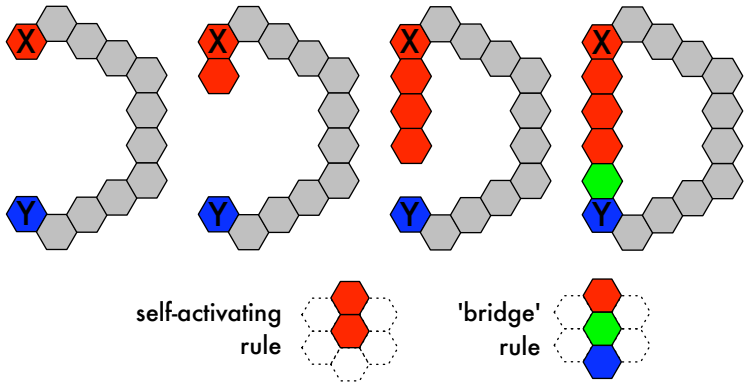


Figure 9.22: An example of a 'useful' self-activating rule/module.

9.7.1 Structural Dependencies and Ordering

While this approach seems to successfully reconcile the difficulties associated with self-activation, significant problems remain, because the red column's construction depends on the limitation provided by brick **Y**. The column building rule is self-activating, so as soon as brick **X** is present the column construction may begin. Brick **Y** must then be placed *before* three red bricks have been placed, at the very latest. If construction begins around brick **X**, there are no mechanisms available to ensure that brick **Y** will be placed before construction of the red column exceeds the number of bricks required to join as indicated.

To avoid this, it must be *guaranteed* that brick **Y** exists before brick **X**. This can be achieved by starting construction at **Y** and building each brick around to **X**, thereby ensuring that the relative ordering of brick **Y** is before that of brick **X**. Any self-activating module within a stigmergic algorithm must depend crucially on the ordering of other modules within the algorithm.

While this dependency is resolvable in the example presented in Figure 9.22, more complex architectures with non-linear building behaviour (i.e. where more than one brick can be placed at a given time during construction) will introduce more complex ordering. Resolving the structural dependencies between multiple self-activating modules may be impossible.

One significant exception to the situation described above is possible: *self-terminating* modules. However, these require the allowance of rotated rules. This type of module will be discussed in the following section.

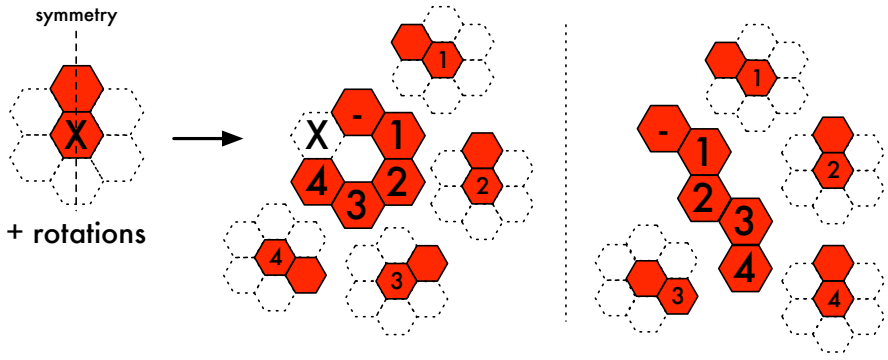


Figure 9.23: Building a ring structure using rotated self-activating rules. The particular rotation which matches cannot be controlled, and the output structure may therefore be inaccurate.

9.7.2 Self-Activating Modules and Rotation

One final possibility for safely deploying self-activating modules is if the structure which causes construction to terminate is generated as part of the self-activating construction itself – a **self-terminating** module. For instance, a 'ring' of construction, such as the simple ring shown on the left in Figure 9.23 might employ a single rule to build most of the bricks (other than brick **X**, which requires a 'bridge' rule, as above). However, to allow the construction to encounter itself, the units of construction must rotate to produce a curve.

The consequences of allowing rule rotations to match during simulation were discussed in Section 7.6. It was shown previously that if rotated rules were allowed to fire, stimulating configurations may appear in which a rule may be able to fire in a number of different rotations. In such situations, the progress of construction cannot be controlled or determined in advance, and so it is not possible to guarantee that the final architecture will be accurately produced. An example of such behaviour using a single, self-activating rule can be seen on the right in Figure 9.23.

Modular Rings

The problems highlighted above (and previously in Section 7.6) stem from *symmetries* within the structure, and matching rules. The single rule previously discussed is symmetrical through the line formed by the two bricks within it. If the brick matched against has sufficient empty cells surrounding it, the rule can fire in up to three different rotations in

any given neighbourhood.

A further example of this can be seen in Figure 9.24, part **A**. A single repeated module can be built in rotated forms (by allowing rotated rules) with the intention of constructing a ring[8]. However, if the module is symmetrical, there is no way for the rule which places brick **X** to ensure that the clockwise curve is maintained. It is equally likely that brick **X′** will be built.

The ring presented as part **B** of Figure 9.24 has been modified such that the symmetry of the module can be broken by introducing an alternative brick colour. The neighbourhood surrounding brick **X′** is now differentiated from that surrounding brick **X**, and so the direction of the curve can be maintained. However since rotated individual rules may fire, the rule which builds brick **A** may match against brick **X** in a rotated form, and place a brick where brick **B** should be.

It may be possible to contrive by hand a ring-structure built from repeating modules, but such a stigmergic algorithm would have to manage not only the symmetry issues highlighted here (and those in Section 7.6), but also the added complexity of increased post-rule conflicts since rotated rules may match in many parts of the construction.

9.8 Summary – Limitations of Pattern Extraction

While initially a repeating pattern-based approach was appealing, the previous sections have shown that this approach presents the most significant barrier between the algorithm extraction techniques developed thus far, and achievement of the 'holy grail' goal stated in Section 6.1: "...to discover the minimum set of rules necessary to produce a given architecture."[156]. While it is clear that compact stigmergic algorithms can only be produced where each rule is responsible for more than just a single brick within the final architecture, extracting such an algorithm reliably in any situation has been shown to be an enormous algorithmic undertaking.

[8]Closed-curve structures are commonly found in 'natural' structures, as demonstrated by the many cell and envelope structures observed in biological nests[156, 17, 29, 94]. This type of structure is doubtless of high functional worth, and therefore the replication of such features would certainly prove useful in any physical application of stigmergy systems.

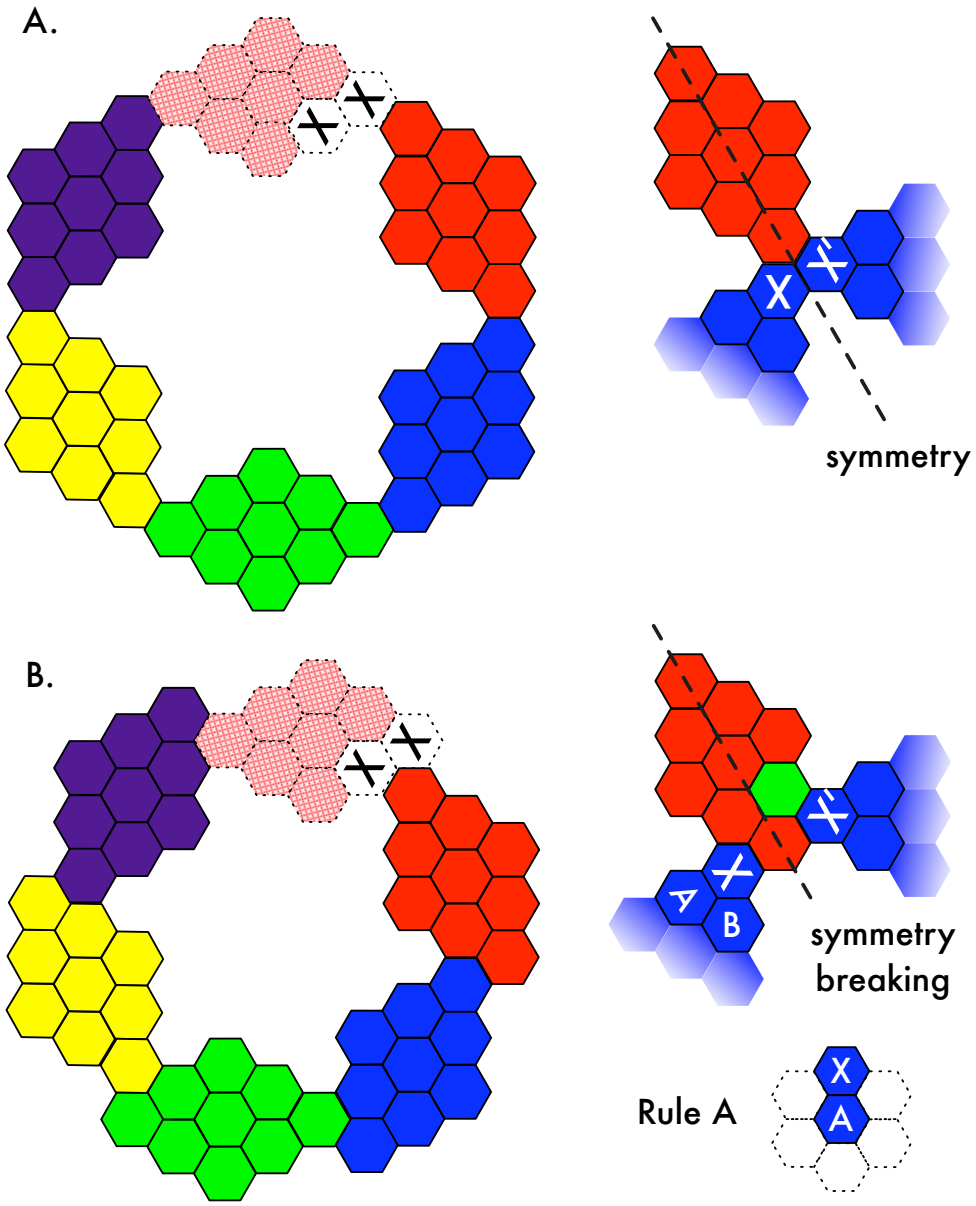


Figure 9.24: Symmetry problems exist when attempting to build a cycle of self-activating modules.

Self-activating patterns in particular present an unavoidable and fundamental problem. While such patterns can generate potentially unlimited amounts of structured architecture, if such systems are to be practically useful, this behaviour must be controlled.

Pattern extraction requires a global view of the architecture, and the construction behaviour of the algorithm. Considering individual parts of the architecture introduces further complexity to a process whose computational complexity is already combinatorial.

9.8.1 Guaranteeing Minimality

Even with pattern-based ordering, in all but the simplest cases[9] it is not possible to prove that the algorithm produced is minimal. In order to do so, each different combination of patterns which can successfully compose the whole architecture must be considered, in addition to the many different orderings of bricks within each of those patterns. This problem is once again combinatorial in nature, and while such an exhaustive search may be possible for small, simple architectures, it inevitably becomes intractable for larger structures.

It now seems certain that it is *not* feasible to extract truly minimal stigmergic algorithms from arbitrary structures, using the *Nest* model.

9.8.2 Modular Construction in Other Abstract Systems

Grammar-based structure generation systems such as L-systems[10] have successfully exploited repeated substructures in the of production large structures using compact algorithms (or grammars, in this case). The ease with which such a closely-related abstract system can accommodate modular design, while *Nest* systems cannot, serves to highlight the fundamental difference between lattice-based agent systems and grammar-based abstract systems.

Within *Nest* systems, building is based wholly at the lowest level of granularity (i.e. individual bricks). In contrast, the 'structure' in an L-system is refined, and the structure is generated using a top-down approach as the grammar continually replaces elements and refines the structural sentence.

[9] The structure shown in Figure 9.1 is a typical, simple example of a minimal algorithm.
[10] L-systems are discussed previously in Section 3.6.2.

This initial global definition of the structure is what ultimately enables grammar-based systems to generate modular architectures. It is trivial to define a grammar which translates the initial axiom into a number of appropriately arranged symbols, each of which can later be rewritten as a module. This gradual refinement of the entire architecture is clearly a powerful approach. However, it is entirely incompatible with the principles of stigmergic assembly.

9.8.3 Stigmergy, Local Information and the Limits of *Nest*

The necessarily local nature of stigmergy, and in particular the model of qualitative stigmergy presented in the *Nest* systems, has shown itself to be the source of extreme complication which considering the automatic extraction of repeating modules and stigmergic algorithms. As a stigmergic algorithm is executed, any implicit *modularity* is discarded: a stigmergic algorithm is a collection of rules, and only rules.

This notion was first explored in Section 5.1.1, where it was shown that 'building stages' can often only be derived by watching the construction as it proceeds. It is now clear, given the detailed consideration of modularity presented in this chapter, why this is often the case.

From the perspective of a designer, the constraints present in a modular stigmergic system are significant: if a rule can fire in more than one place, all interactions between that rule and and its neighbours – in each location – must be managed. This has been shown above to be a formidable, and possibly intractable undertaking.

Chapter 10

Discussion

Overview

While this investigation into the behavioural possibilities of stigmergic systems is essentially complete, several outstanding issues have yet be considered. In this chapter, the achievements of this thesis are noted, potential avenues for further research are suggested and some remaining questions are brought together and discussed.

10.1 Designing Emergence with Stigmergy

The goal of this research, when it was first conceived, was to examine the possibility of using the promising but elusive property of 'emergence' in real-world system design. While now-classic examples of emergent behaviour are universally acknowledged as elegant and impressive, it was unclear whether emergent phenomena could be made practical and useful in arbitrary contexts.

The manipulation of any complex system requires an understanding of each of its components, before significant conclusions can be drawn about the behaviour of the system as a whole. As with many other investigations into emergent behaviour[100, 166, 85], the *Nest* system was selected on the merits of its abstract nature. The simplicity of abstract systems ensures the absence of any external influences affecting their behaviour and thus promises a perfectly isolated experimental environment in which to explore the nature of emergent

systems. There is no doubt that despite the simplicity of the underlying rules which govern the behaviour of these systems, a large range of distinct behaviours can be generated.

The results presented in Chapters 6, 7 and 8 have shown that it is possible to design agent systems which use emergence, in the form of stigmergic building, and which achieve the specific and arbitrary goals of the system designer. In parallel, the range of possible system behaviours using the *Nest* model has also been explored, and the consequent limitations on the application of abstract stigmergic systems to arbitrary problems (i.e. problems far from any biological context) have been examined. In this manner, while it was not possible to derive a technique which produced *minimal* stigmergic algorithms, this research should be considered successful in achieving the above-stated goal.

10.2 *Nest* Systems and the Real World

Perhaps the most significant problem encountered during this study is ensuring that construction stops at a suitable point; the greater part of Chapter 9 considers the range of mechanisms for limiting building behaviour available within a *Nest* system. Despite this focus, the problem remains intact – within the constraints of the *Nest* model, it is extremely difficult to specify an algorithm which is compact yet does not exhibit unending construction activity.

The continued presence of this problem, despite the work presented here, is purely a result of the strict qualitative stigmergy which forms the foundation of the *Nest* model. The *only* mechanism available with which to measure architectures beyond the local neighbourhood is brick colour. It was shown in Chapter 6 that if no limit is placed on the number of brick colours, any architecture of *any* size can be reproduced using a stigmergic algorithm.

However, for large architectures this results in vast numbers of distinct brick colours (increasing at least linearly to the maximum dimension of the architecture[1]) which must be available in the environment, and distinguishable by agents.

[1]See Section 9.2.3 for discussion regarding the lower limit of the number of brick colours required to build an architecture of given dimensions.

10.2.1 Limiting Building Behaviour using Quantitative Stigmergy

Taking inspiration from biological swarm systems, the solution to this limitation seems obvious: long-ranged communication is achieved via the dispersal of chemical signals – pheromones – throughout the environment. The strength of any dispersing signal decreases as the distance from the source is increased. By allowing behaviour to be expressed only if the strength of the signal exceeds (or alternatively is below) a given threshold, building behaviour can be limited (or triggered) at distances far greater than the local sensory neighbourhood of an agent. Furthermore, the strength of the initial pheromone deposit can then be tuned to determine the distance at which this modification of behaviour will occur.

While such a mechanism is not compatible with the strictly qualitative *Nest* system, it is a far more elegant and efficient means of measuring distance than increasing the *complexity*[2] of the algorithm linearly as the dimensions of the architecture increase.

As highlighted in Section 4.8, there are no significant issues preventing the implementation of pheromone dispersal within the *Nest-3.0* system. From an experimental perspective, the addition of pheromone-based behaviour would enable the production of much simpler algorithms by *Nest* algorithm designers.

10.2.2 Quantitative Stigmergy and Global Construction Control

Other forms of quantitative stigmergy could be employed to guide the behaviour of the swarm. For instance, the local density of agents, when exceeding a certain threshold, could trigger group activity. This behavioural mechanism is seen in the social insect systems upon which *Nest* is based[12, 76].

Implementation of local agent density as a building trigger is again relatively simple within a *Nest-3.0* simulation[3]: if an agent measures the number of agents within its local environment to be above a certain threshold, then construction rules will be matched against the local brick configuration; otherwise, do nothing. The measurement of agent density could even be achieved by pheromone deposits by each agent within the environment.

[2]See Section 10.3 for a discussion of 'stigmergic complexity'.

[3]Measuring agent density is not possible within the original *Nest-2.11.1* system, since only a single agent 'exists' within the environment (see Section 4.2.4).

Simplicity through Quantitative Stigmergy

If *quantitative* stigmergy is employed, the mechanism which initiates and drives continued construction can be *the same mechanism* which limits the construction once it should be stopped. When using qualitative stigmergy, a specific and unique local configuration must be achieved in order to prevent future building, typically requiring the detailed specification of that configuration and how it is to be reached by the collective. Many rules and stimulating configurations must be managed in order to ensure this situation is achieved.

Quantitative stigmergic construction, on the other hand, ceases naturally once the intensity of the initiating factor – proximity to a pheromone source, a high density of agents, and so on – has decreased to an acceptable level, indicating that construction is no longer required. A contrived example of this technique in practice may be the construction of housing for agents: if the 'nest' structure has insufficient capacity, the local agent density at any point will be elevated. This may trigger those agents who detect increased density into building behaviours which extend and enlarge the nest structure. Once sufficient new housing has been built, the local agent density will naturally drop, and without the stimulus to continue building, agents can return to alternative activities.

Quantitative Stimuli and Abstract Simulation

Within the comfort of an abstract simulation, quantitative stimuli such as pheromone deposits can seem like an ideal mechanism for controlling the global behaviour of many agents. In reality, however, this mechanism is subject to a variety of external forces, and would very rarely provide the perfect, continuous gradient which might be assumed in computer simulation.

Chemical deposits, for example, will diffuse throughout the environment and produce the gradient discussed previously. However, this diffusion may never be perfect: chaotic flow and the presence of matter in the environment (including, in this context, the structure being built itself) will necessarily disturb this distribution, and therefore the chemical intensity distribution may never match that suggested by Figure 10.2 later in this chapter.

Furthermore, pheromones tend to decay *over time*. Since there is no real embodiment

of time within a *Nest* stigmergic algorithm (other than the quantised notion implicit in the brick ordering), it will be hard to account for pheromone decay in the building behaviour of a stigmergic system which uses anything other than highly-abstract 'chemicals' and physics during simulation.

10.2.3 Quantitative Stigmergic and Algorithm Extraction

The dual keystones enabling the effective extraction of stigmergic algorithms from existing structures are the absolutely discrete mechanism which determines whether a rule will fire or not, and the permanence of brick placement within the *Nest* system.

The removal of bricks by agents ('excavation'; see Sections 4.2.5 and 5.4.4) allows for a brick to be placed and removed any number of times before the 'final' architecture is submitted for analysis. It has been shown that the algorithms developed previously in this thesis can produce a viable stigmergic algorithm to build any architecture; consequently a stigmergic algorithm can be found to reproduce the architecture in question. However, the resulting stigmergic algorithm will only include 'building' rules. It is *impossible* to infer the existence of any rules which remove bricks, since they leave no bricks, and therefore no evidence, to indicate their activity during the simulation.

Similarly, the presence and decay of pheromones cannot be determined by examining a *snapshot* of construction at some given time. By introducing a stronger sense of 'time' to the simulation and stigmergic algorithm (through the existence and dependence on the continued decay of stimulating gradients, or the random movements of agents) the information contained within a single snapshot of the architecture (and the state of the environment in terms of and quantitative gradients) is now insufficient to *precisely* determine the triggers responsible for each piece of building activity.

In other words, if the behaviour of a stigmergic system is determined to some extent by factors which change as the simulation progresses, such as the addition, diffusion and decay of quantitative factors, or the *removal* of bricks, then it may not be possible to infer the presence of such factors without examining the *behaviour* of the system, instead of a static snapshot of the system state in the form of an architecture.

10.3 A Measure Of Stigmergic Algorithm Quality

It was shown in Chapter 6 that a stigmergic algorithm can be extracted from any architecture. From that point, the focus of this research has been to improve the fundamental techniques and produce 'better' algorithms. However, little has been presented to qualify one algorithm as 'better' than any other. In this section, the notions of algorithm quality and complexity are discussed.

10.3.1 Stigmergic Algorithm Complexity

Algorithmic complexity typically refers to the computational tractability of a given process. However, in this context it is more useful to define the complexity of a stigmergic algorithm as the amount of information required to specify a particular algorithm. More complex stigmergic algorithms will feature a greater number of rules or colours, and therefore require a more sophisticated agent to operate. The complexity of a stigmergic algorithm a might be defined as:

$$\text{complexity}(a) = f(\text{num\_rules}(a), \text{num\_brick\_colours}(a)) \tag{10.1}$$

Where the value of $f()$ is stronly monotonic in the number of rules, and also strongly monotonic in the number of brick colours. In other words, as the number of rules or bricks increases, the complexity of the algorithm should increase proportionally.

What should be taken from equation is that the two factors which determine the 'complexity' of a stigmergic algorithm are the number of rules, and the number of distinct brick colours within the algorithm. A lower complexity is more desirable, since simpler agents (simulated, robotic or otherwise) can be created which will 'run' the stigmergic algorithm. Stigmergic algorithm complexity can therefore also be considered as stigmergic algorithm *size*.

This notion of complexity is very similar to that of minimum description length ([55]; see [39] for a discussion of this within the context of SUBDUE and sub-structure discovery). The minimum description length principle [138] states that the best theory to describe a set of data is that which minimises the description length of the entire data set. Within the

Nest context, the best algorithm is that which accurately reproduces the desired architecture whilst minimising the size of the description of the algorithm.

10.3.2 Stigmergic Algorithm Quality

Since the quality of a stigmergic algorithm is now an important measure of the success of any derived extraction process, it is important to more clearly state how such quality might be determined. The quality of a stigmergic algorithm can be tied directly to the following three factors:

Architecture Size – Algorithms which can produce larger architectures are, all other factors aside, more desirable, since the algorithm responsible more efficiently describes the construction process. In other words, if two algorithms A and B, of equal 'complexity' (see above), can produce desirable architectures of sizes S_A and S_B, where $S_A > S_B$, then algorithm A is 'better' than B. Stigmergic algorithm quality *increases* with produced architecture size.

Architecture Desirability – While large architectures are generally considered 'better' than smaller ones, if the architecture does not fit the criteria of its users, the worth of the algorithm is lessened. With regard to the body of work surrounding *Nest* systems up to and including this thesis, the desirability of any architecture is determined ultimately by external observers[4]. As a result, the determination of the desirability of an architecture is subjective, or at the very least determined to a large extent by the context in which the stigmergic system exists, rather than through any objective or universal means.

This reflects a discussion presented in Section 5.2, where the measure of the 'coherency' of a structure is examined, and it is argued that any non-functional method of determining the desirability of an architecture depends on the interpretation of the structure by the observer.

[4]While [19, 22] featured work to automatically evaluate 'more natural' structures, the success of this evaluation can only be measured against the original judgements of the external observers as to what constitutes a 'natural' structure.

Whatever means are selected for measuring the desirability of an architecture, the quality of a stigmergic algorithm *increases* with architecture desirability.

Algorithm Complexity – As was described above, the complexity of an algorithm is a factor of both the number of rules, and the number of distinct brick colours within it. An algorithm with lower complexity can be 'performed' by simpler agents, and so a lower algorithm complexity is clearly desirable. Therefore, stigmergic algorithm quality *decreases* as algorithm complexity increases.

10.3.3 Algorithm Quality and The Motivation for Emergence

The discussion presented above attempts to loosely formalise the assertion that a 'good' stigmergic algorithm is capable of producing more output (structure) for less input (algorithm size or complexity). This embodies one of the original motivations (see Section 1.2.2) for using stigmergic and emergent systems in general: complex, interesting and/or useful collective behaviour emerging from the actions and interactions of relatively simple individual agents. The simplicity of the agents, when considered in an abstract framework such as *Nest*, is directly analogous to the simplicity of the algorithm which specifies their behaviour.

10.4 Stigmergic Complexity

An interesting by-product of the consideration of stigmergic algorithm complexity is a possible means of objectively measuring the complexity of 'physical' structure using these techniques. If it is possible to extract a stigmergic algorithm from a structure, and then measure the *complexity* of that algorithm, then it is possible to label the original architecture with the complexity of the algorithm which is required to produce it. In other words, the relative complexity of physical structures may be measurable by determining the complexity of the stigmergic algorithm which is capable of reconstructing that architecture.

Such a measure would hold the most meaning if it could be shown that the extracted algorithm is always *minimal*. The present lack of minimal algorithm-producing techniques denies such strong measurement. However, despite this the progress towards this goal presen-

ted in this thesis allows some less strict assertions to be made. For instance, the *Increasing States* algorithm has been shown to provide the minimal brick colour assignment for a given set of rules. The minimal number of brick colours required to ensure an architecture will be accurately reconstructed might therefore be used as a weaker measure of relative structural complexity.

10.5 Beyond Modules: The Construction of Features

It could be concluded from the results presented here that it is simply not possible to automatically produce biologically plausible, elegant and *minimal* stigmergic algorithms given only the target architecture. However, the inability to extract minimal algorithms is not necessarily a failing in our understanding of stigmergic systems, or in our ability to formulate algorithmic procedures for dealing with repeating module constraints. It may not be possible to process certain structures using *any* algorithm; their algorithms may only be arrived at experimentally through examination of system behaviour.

For instance, the maze-construction algorithm shown in Section 5.2.4 and presented again as Figure 10.1 produce regular features but lacks structural patterns. This algorithm would be impossible to extract using the techniques developed and explored herein. Instead, it can only be 'discovered' through trial an error – or *evolution* with some suitable measurement of fitness. The *best* algorithms, maximising the ratio between smaller algorithm size and larger useful or desirable structure, may be those which defy analysis or extraction.

The elegance of the maze-building algorithm exists in part because of the application of rotated versions of the two fundamental rules, but more importantly in the fact that the quality of the produced structure is not dependent on large, intricate, repeated substructures. Instead, the 'useful' aspects of the structure *emerge* from the empty space produced as the bricks are placed. The subjectivity of structural quality was discussed in Section 5.2.2, along with the argument that while some structures might *appear* modular and 'coherent', using these as criteria for measuring the success of a stigmergic algorithm is always subject to challenge.

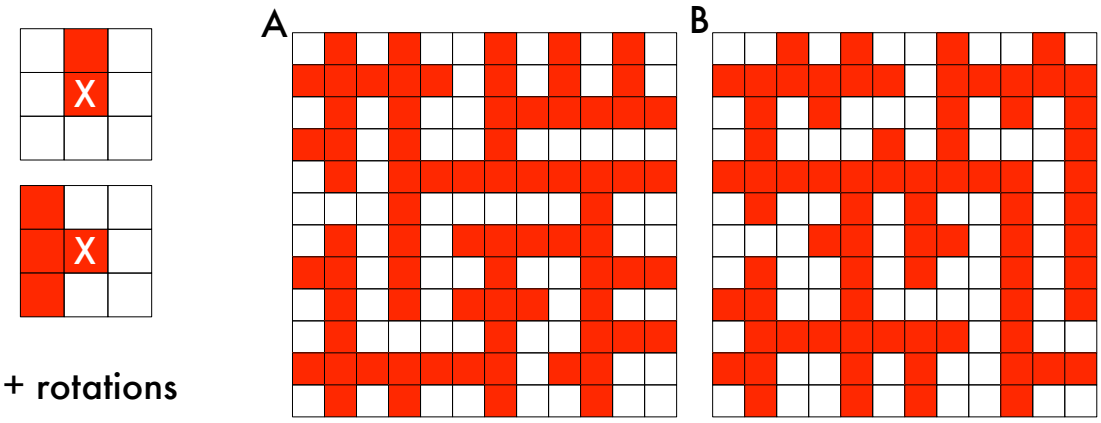


Figure 10.1: With rotation enabled, seemingly-complex architectures with clearly defined *features* (in this case corridors and chambers of definite sizes) can be produced with extremely simple rule sets. The results of two different simulations are shown in the right, both using the same two rules on the left, plus all rotations through the Z-axis (effectively 8 rules if fully specified). This Figure also appears in Section 5.2.4 as Figure 5.4.

10.6 Future Work

The work presented in this thesis suggests several avenues for further investigation. While the work here has assisted in clarifying some of the limitations of the *Nest* system, both as a model of biological stigmergy and as a tool of the design of emergent multiagent systems, much of the vast range of capabilities remains to be explored. As a result of its modular design and simple extensibility, the new *Nest* implementation presented in the *Nest-3.0* tool (Chapter 4) makes the further exploration of a wide variety of abstract stigmergic systems a feasible prospect.

Several areas which deserve consideration beyond that presented in this thesis are discussed below.

10.6.1 Beyond The *Nest* Model

In this investigation, we have tested the potential of the strictly sematectonic[5] *Nest* model of stigmergic behaviour (in its abstract form) to its limits. Despite the apparent flexibility and success demonstrated in [158, 156, 22, 19, 17], many of the implicit limitations present

[5]See Section 2.3.4 for a discussion of different forms of stigmergy.

in purely-qualitative stigmergic systems have now been made clear, most significantly the complexity of managing modular construction and the assembly of rotated structures.

As discussed earlier in this chapter, the addition of forms of quantitative stigmergy would enable a far wider range of stigmergic algorithms to be developed. In particular, quantitative stigmergy offers a solution to the problem of halting construction. The development of algorithms which use this technique should be investigated in detail, so that the viability of including this mechanism in non-biological systems can be ascertained.

Rotation and Local Compasses

While the capability to apply rotated instances of stigmergic algorithm rules is available in both *Nest* implementations (Section 4.2.3), the development of algorithms which exploit rule rotation has not been considered in depth, as a result of the conflict between rule rotation and brick colour assignment (see Section 7.6).

However, the addition of gradient fields to the environment (e.g. through pheromone diffusion) allows the introduction of a new type of rule rotation – **local compass** rotation. Rather than relying on the internal compass to determine a global 'north', a 'local north' could be established along the line of increasing intensity of a particular pheromone gradient. This is illustrated in Figure 10.2. While allowing this form of rotated rule matching does not remove the problems highlighted in Section 7.6, it *does* allow stigmergic systems designers to specify construction symmetrical around a central axis. It seems certain that this would enable even simpler specification of stigmergic algorithms which build biological nest-like structures.

The Relationship Between Geometry and Algorithm

If stigmergic systems based on the techniques developed in this thesis are physically implemented, agents must work without the presence of a lattice, or by maintaining a representation of the environment as a lattice internally. As noted in Section 4.8, the exploration of alternative lattice geometries may give far clearer insights into the effects artificial geometries impose on stigmergically-constructed architectures. For example, by comparing hexagonal

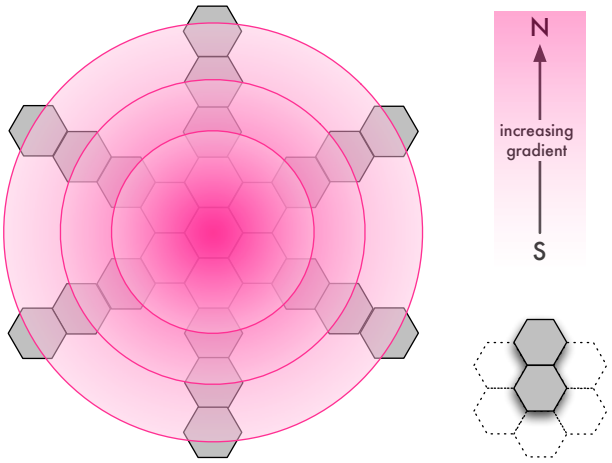


Figure 10.2: The application of a rule rotated to match 'local' north. If north is defined by each agent as the direction along the line of increasing gradient, then rules can be applied to build structure growing outward from the source of the gradient.

stigmergic algorithms with those based in a cubic lattice, it is clear that the hexagonal cells enable smoother curves to be build using dramatically fewer rules than would be required within a cubic system.

Better understanding how the geometry of the lattice impacts upon the types of shapes that can be easily built, and of how the symmetries within that lattice affect the rotation of rules, will contribute significantly to a more complete understanding of the potential of stigmergy in real-world applications.

10.6.2 Stigmergic Architecture Repair

As is typical in science, understanding of how a particular system works is often most aided by observation of that system as it malfunctions. Much of the original research into natural stigmergy was performed by observing the behaviour of insects performing construction behaviours, and in particular observing how those behaviours adapted in the presence of external modification, or damage, to the architecture under construction [145, 76, 29] (see Section 2.3.1). While some of the most valuable information is obtained by the structural pathologies introduced by the insect's attempt to continue construction, the ability to perform repairs on an existing structures would certainly by extremely valuable for almost any application of stigmergic systems.

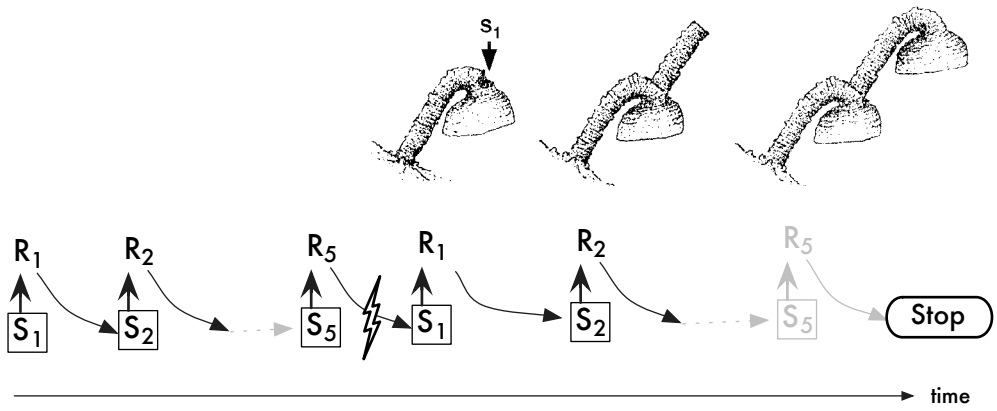


Figure 10.3: When a stimulating environmental configuration is created out of sequence, the pathological building behaviour of the stigmergic agent is revealed. Taken from [157], originally presented in Section 2.3.1 as Figure 2.4.

It is currently unclear how stigmergic algorithms within the *Nest* system might incorporate repair behaviours. An excellent validation of any form of abstract stigmergic model however would be the demonstration of similar pathologies when the virtual architecture is similarly damaged by experimenters during simulation.

Furthermore, it is currently unknown as to how stigmergic systems based on the *Nest* model could perform *successful* repairs to a structure during simulation, or how an existing stigmergic algorithm must be modified to incorporate such behaviour. If damage to the structure occurs during building, it is almost certain that the stimulating configurations within that region will *not* match those which trigger the rules which originally built bricks there. Therefore it is likely that to perform repair, the architecture in that local area may have to be modified, by the removal of bricks[6], in order to bring it to a state where some repair routines can operate.

It is also crucial that defects in structure can be *detected* as errors, rather than as accurate and timely configurations of bricks (that is: occuring in the correct region of space at the appropriate moment during construction). As shown in Figure 2.4, if an erroneous configuration is not recognised as an error, structural deviations can easily be produced.

This line of investigation will be of great worth if stigmergic construction is to successfully

[6]Excavation and the issues surrounding brick removal in *Nest* and stigmergy in general are considered in Section 4.2.5

transfer from the simulated domain to the physical world.

10.6.3 An Interactive Approach to Modular Deconstruction

The discussion of automatic pattern detection in structures (Chapter 9) concluded that the methodology considered was now currently tractable, even for architectural deconstruction problems which are easily performed by the observer.

A promising approach to performing pattern-based algorithm extraction may leverage the aptitude of human observers to identify likely patterns, via a software interface, and allow the software to identify repetitions of this pattern, coverage of the architecture, and possible undesirable interactions (such as those caused by a self-activating module). In this manner, a part of the combinatorial problem of pattern identification can be spread between the human brain, a natural pattern-finding machine, and the meticulous analysis of the computer.

Chapter 11

Conclusions

In this final chapter, the achievements and advances presented in previous chapters are summarised. Key points are highlighted in **bold** typeface.

Nest-2.11.1

In the work presented here, the field of stigmergic construction, and in particular **the *Nest* model of stigmergy developed originally by Bonabeau, Theraulaz et al. has been considered in detail**[158, 156, 17, 19, 22, 29]. The *Nest-2.11.1* software, which was used to explore and latterly evolve stigmergic algorithms, is also briefly described. Several assertions were presented by Bonabeau et al. as a result of their initial experimentation:

1. "All biological-like architectures ... are generated by coordinated building algorithms, in which the shape to be build is naturally decomposed into modular subshapes ... defining corresponding building stages."[156]

2. "No 'interesting' pattern can be generated using a single type of brick."[19]

3. "The coordinated subspace [is] relatively small and compact ... This property is confirmed by random exploration of the rule space."[156]

Criticism

The original investigations into simulated stigmergic construction were critically discussed, and flaws in their use of a genetic algorithm to search for 'coordinated' algorithms were highlighted, including:

- Using a fixed-length genome, and thus fixing the size of 'coordinated' algorithms, and penalising shorter algorithms
- Using a subjective fitness function which penalises the existence of interesting architectural substructures smaller or larger than a $5 \times 5 \times 5$ cube

It was finally asserted that the closeness between algorithms and architectures actually operates at the level of building stages, rather than individual rules, within an algorithm. Instead, it is shown that ***two algorithms which contain identical or similar building stages will produce identical or similar architectures***.

The relationship between the *Nest* model and other abstract systems are also explored. Strong similarities exist between lattice swarm systems and cellular automata, while the model fundamentally differs from L-systems, another prominent 'constructive' abstract system.

Nest-3.0

The original abstract stigmergic software developed (*Nest-2.11.1*) suffers from the shortcomings of being implemented only on a single computing platform (Unix and X-Windows) using a terse implementation language (`C`), and being too closely tied to the original *Nest* model description. **In order to remove any future experimental constraints, a new *Nest* simulation implementation was developed, named *Nest-3.0***.

From the outset, this software was written to run on a wide range of computing platforms (Section 4.3). By selecting the `C++` language for internal data representation, the speed of operations on fundamental *Nest* entities (cells, agents, rules, architectures and so on; Section 4.4.1) is maximised. The user interface is written using the widely available `OpenGL` graphics

language, enabling the real-time 3D display of generated architectures. The high-level `Ruby` language acts as a glue between the former and the latter, and enables fast prototyping of new simulation features, and additions to the *Nest* model. Novel features within *Nest-3.0* include:

- **Creation of arbitrarily-sized sensory neighbourhoods** (Section 4.2.2)
- **True multiagent system, using multiple agents with distinct locations and arbitrary internal state** (Section 4.2.4)
- **Brick excavation in addition to building; in general, stigmergic rules which specify arbitrary neighbourhood modification** (Section 4.2.5)
- **Flexible brick state matching against subsets of all brick values** (Section 4.2.6)
- **Full rotation of rules, with novel speed optimisations for rotating cubic neighbourhoods** (Section 4.5.2)
- **Consistent hexagonal graph lattice production via 'hypercells'** (Section 4.6)

Further extensions, along with suggested implementation methods, are also noted.

Automatic Generation of Coordinated Algorithms

In this section the original approaches adopted by Bonabeau et al. to search for 'interesting' structures were considered, along with the assertions they made regarding what characteristics a stigmergic algorithm must have to produce such structures. The notions of 'coordinated algorithm', 'building stages' and 'coherent structures' have been examined and their various weaknesses exposed. Despite closely examining the description presented in [156], an unambiguous and testable understanding of what 'coherence' means with regards to an architecture, and what 'coordination' means with regards to a sequence of stigmergic rules, remains elusive.

Temporarily accepting the informal *notions* of coherent structure and coordinated algorithms, it is then shown that **structures which do *not* fit with these definitions**

may still exhibit interesting properties despite variations in their precise appearance over a number of simulation runs. The strongest example presented as a challenge to the notions of 'coherence' produces regular 'maze' structures using only two rules. Despite structural differences, this algorithm reliably produces specific architectural features.

Post-Rules

The concept of **post-rules** (along with pre-rules and meta-rules) was introduced as a means to avoid some of the ambiguity present outlined above, and as a better means for automatically generated stigmergic algorithms for the systematic exploration of the space of possible rule sets. However, despite the promise post-rules would appear to hold for predicting algorithm behaviour, they can only indicate which rules *could* fire next, instead of which rules *will* fire next during simulation.

Numerical calculations indicate that while the size of of the problem space has been hugely reduced, it is still computationally intractable to systematically explore the entire space even for simple systems. **The approach adopted by Bonabeau et al. is therefore impractical as a means for thoroughly examining the potential behaviour of stigmergic systems**.

Algorithm Extraction

As indicated by the title of this thesis, the ultimate goal is to investigate means of designing systems which use emergent behaviours to achieve arbitrary goals. Within the context of stigmergic construction, these arbitrary goals are any structure or valid arrangement of bricks within the model. It is therefore the aim of this work to devise means of producing an emergent, stigmergic system which produces these architectures.

A simple method of extracting a stigmergic script from any architecture has been developed. The two keys steps in this process are

1. Assign a random *valid* order to the bricks within the architecture

2. Assign a unique colour to each brick within the architecture to avoid any potential rule conflicts during simulation.

If the ordering of bricks placed is known, then the neighbourhood around a brick at the time that brick was built can be determined by removing all bricks from the neighbourhood which are ordered *after* the brick to be placed. This modified neighbourhood is therefore identical to the rule which placed the brick in question. The two steps of ordering and state assignment are sufficient to produce a set of rules which *reliably* reproduce the architecture supplied to the process.

Most importantly, this process shows that **any architecture can be decomposed into a set of stigmergic rules**. In other words, *no* architecture exists, representable within the limits of the *Nest-3.0* system, that cannot be decomposed using this technique. However, algorithms produced by this process are far from optimally small. Instead, the algorithms always contain the same number of rules, and distinct colours of bricks, as there are bricks in the architecture.

The *Increasing States* Algorithm

To reduce the number of distinct brick colours in an extracted stigmergic rule set, a novel algorithm was invented. **The *Increasing States* algorithm produces minimal brick colours for any given rule set**. A new brick colour is added to the algorithm *only when it is required to prevent a rule firing out of sequence.* Such conflicts are identified by analysing the rules within the algorithm and using post-rules to determine which rules might fire sequentially.

The purpose of distinct brick colours – necessary in qualitative stigmergy – is clarified in a discussion of the limitations of using material qualities as a mechanism for managing rule activation. It is shown that if rotated rules are allowed to fire within a simulation (in other words, if a stigmergic agent has a limited internal compass), then it is *impossible* to fully constrain the space of stimulating configurations which will cause that rule to fire.

Ordering using a Genetic Algorithm

It is shown that different brick orderings can affect the minimal number of brick colours required for a stigmergic algorithm. Similarities between the ordering of bricks within a structure, and other node ordering problems such as the classic Travelling Salesman Problem (TSP) were also noted, along with distinctions. A genetic algorithm (GA) approach was selected to perform the ordering, and the importance and suitability of genome representations are discussed in relation to *valid brick orderings*. **Direct** representations of node order require sophisticated crossover and mutation operators to ensure that the resulting offspring solutions remain valid. **Indirect** representations require more effort to determine the brick order from the representation, but crossover and mutation can be applied more simply.

The results of GA experiments have been presented, and the approach has been shown as viable in reducing the number of brick colours required to reproduce arbitrary architectures, although initial populations of random solutions tend to contain solutions which are near the best found.

Patterns, Structure and *Nest* Stigmergy

Despite the optimisation of this algorithm extraction approach, the number of rules within an algorithm remains equal to the number of bricks in the input structure. It is shown that in order to produce smaller algorithms, rules within the algorithm must fire repeatedly, and this repetition must be carefully controlled such that construction behaviour does not stray from that which is desired.

Two types of fundamental repetition are identified within single rules – *independent* and *self-activating*. This same distinction exists within *groups* of rules, or modules. Independently repeating modules are also termed *leaf* modules. **The discussion of repetition clarifies the original discussion of building states in [156], by demonstrating the difference using the properties of rules as explored in post-rule investigations**.

Accurate Construction

The issue of ensuring accurate construction with the presence of repetition is considered in detail. The necessity of brick colours as a device for global structure measurement is shown. It is also shown that **the minimum number of brick colours required to build a structure of maximum dimension n bricks is $n/2$**. Furthermore, it is demonstrated that **in the absence of external structures, self-activating modules can never be used within a stigmergic algorithm**.

Automatic Module Division and Tractability

The issue of automatic extraction of repeated substructures, modules or building stages is considered in detail, and it is shown that **this problem holds the same combinatorial limitations as automatic algorithm generation**. Furthermore, an additional battery of combinatorial problems is layered atop, including:

- Detection of self-activating modules
- Prevention of endless construction when using self-activating modules
- Brick colour assignment within module overlap areas

Ultimately, automatic building stage extraction requires a global view of the building behaviour and rule interactions of the algorithm as it is generated, and the derivation of any algorithmic 'shortcuts' has not been possible at this time.

A measure of the power or quality of stigmergic algorithms has been proposed. Through further discussion of the issues raised in this work, it has also become clear that stigmergic algorithms are most powerful when they are not employed to produce exact structures. In contrast, compact stigmergic algorithms can produce large architectures with architectural *features*, whose value may only be exposed via some form of *functional evaluation*.

Avenues for further research which may show practical merit, or improve our understanding of various aspects of alternative stigmergic implementations, form the conclusion of the original work presented here.

Designing Emergence and Swarm Construction, Redux

In the introduction the questions which would guide this investigation were presented and made explicit. At this point, it is useful to reconsider those questions, and where possible, provide answers.

Is it possible to use sematectonic stigmergy to build arbitrary structures?

It was demonstrated in Chapter 6 that a stigmergic algorithm can be produced for **any** structure representable within the *Nest* model of space. The quality of this algorithm, in relation to those which are 'hand-made', is often poor, but this does not detract from the proof that there exists no structure which cannot be decomposed into *some* set of sematectonic stigmergic rules.

If it *is* possible to use stigmergy for arbitrary tasks, how can it be used?

This has also been demonstrated in Chapter 6. By ordering an architecture, it is possible to extract the stigmergic rules which will build each brick. Unique colours can then be applied to each brick to ensure that the algorithm executes as intended.

Can the application of stigmergy to arbitrary construction problems be automated using some algorithm process, and if so, what are the algorithms required?

The mechanisms outlined in Chapter 7 and Chapter 8 describe two possible mechanisms for the automatic extraction of stigmergic algorithms. This process must consist of two steps: ordering, and state-assignment.

A novel process – the *Increasing States Algorithm* – has been devised to determine the minimal number of brick colours which will allow the structure to be reproduced without possibility of error. This algorithm also assigns brick colours to the stigmergic rules. This algorithm was described in Chapter 7

An approximate method ordering the bricks within a structure, based on genetic algorithms, has been presented and shown to assist in minimising the number of brick colours

required to produce an arbitrary structure. This process was described, and experimental results have been presented in Chapter 8.

What are the limitations, if any, of the *Nest* model?

It was seen in Chapter 8 that in order to produce a *compact* stigmergic algorithm, the number of bricks built must be greater than the number of rules within the algorithm. Several sections of an architecture must be built using the same sets of rules, and therefore it must be decomposed into repeating structural units for this to be achieved.

It has been shown in Chapters 9 and 10 that introducing the repetition of rules within a strictly sematectonic model such as *Nest* also brings significant constraint issues which must be considered by any process wishing to extract a stigmergic algorithm. To produce an artefact of any size larger than the local perceptual neighbourhood of an agent using sematectonic stigmergy, a certain minimum number of brick colours (increasing with the size of the artefact) are required. If external implementation constraints place a limit on the number of distinct usable types of building material, purely-sematectontic stigmergy can only be used to build structures of a limited size, determined by the size of the agent's perceptual neighbourhood.

Furthermore, the repetition of certain types of structure introduce the possibility of construction which continues indefinitely. There are no mechanisms within the *Nest* model which can constrain this form of construction. Finally, if agents are able to match rotated versions of rules, production of the desired architecture, without deviations, cannot be guaranteed. In summary, sematectonic stigmergy has been shown to be insufficient to develop compact algorithms for the construction of arbitrary structures.

If any limitations exist, can they be overcome and if so, how?

The strong limitations encountered by this investigation serve to highlight the importance of quantitative mechanisms within stigmergic systems, an example of which is the diffusion of pheromone within the environment. These mechanisms can provide important information which is capable of transmission beyond the local perceptual neighbourhood of the

participating agents.

Very importantly, such mechanisms can provide means of roughly measuring the distance between two points in the environment (i.e. the current distance from the source of the stimuli). This can therefore be used to measure the size of architectural features which are larger than the bricks within the direct perceptual neighbourhood of an agent. Quantitative stigmergy may also provide means to limit the construction of 'self-activating' modules, enabling the use of this type of repetition within stigmergy systems.

Final Words

> "We believe that our study constitutes a first step towards a deeper understanding of the origins of natural shapes in terms of the logical constraints that may have affected the evolutionary path."[156]

It is hoped that this work presents a more thorough and detailed investigation into the nature of abstract stigmergic systems of the type introduced by Bonabeau, Theraulaz et al. than available previous to its inception. It has been shown that while the *Nest* model affords appealing examples of emergent behaviour, the strict qualitative behaviour combined with the vast simplification of space within of the lattice model makes specification of elegant, biologically plausible algorithms difficult. The aspects of this problem which render the automatic design of minimal stigmergic algorithms impossible at this time have now been highlighted and explored in detail.

While it has not been possible to fully realise the goal of this research on the outset – to produce minimal stigmergic algorithms for building arbitrary structures – the advances in our understanding of stigmergic systems should remain invaluable for anyone considering the use of stigmergic techniques in any situation.

.

Bibliography

[1] J. Adam. Emergence, abstract multiagent systems and multi-state networks. Technical Report CSM-373, Department of Computer Science, University of Essex, 2002.

[2] Andrew Adamatzky and Owen Holland. Edges and computation in excitable media. In *Proceedings of the Sixth International Conference on Artificial Life*, 1998.

[3] Andrew Adamatzky and Owen Holland. Phenomenology of excitation in 2d cellular automata and swarm systems. *Chaos, Solitons and Fractals*, 9:1233–1265, 1998.

[4] B. K. Ambati, J. Ambati, and M. M. Mokhtar. Heuristic combinatorial optimization by simulated darwinian evolution: a polynomial time algorithm for the traveling salesman problem. *Biological Cybernetics*, 65:31–35, 1991.

[5] Carl Anderson. Self-organization in relation to several similar concepts: Are the boundaries to self-organization indistinct? *Biological Bulletin*, 202:247–255, 2002.

[6] E. Anderson and M. Ferris. Genetic algorithms for combinatorial optimization: The assembly line balancing problem. *ORSA Journal on Computing*, 6:161–173, 1994.

[7] S. Aron, J.-L. Deneubourg, S. Goss, and J. M. Pasteels. Functional self-organisation illustrated by inter-nest traffic in the argentine ant iridomyrex humilis. In W. Alt and G. Hoffman, editors, *Biological Motion*, pages 533–547. Springer-Verlag, 1990.

[8] Nils A. Baas and Claus Emmeche. On emergence and explanation. *Intellectica*, 202(25):67–83, 1997.

[9] W. Banzhaf. The "molecular" traveling salesman. *Biological Cybernetics*, 64:7–14, 1990.

[10] Michel Baranger. Chaos, complexity, and entropy: A physics talk for non-physicists. available online at http://necsi.org/projects/baranger/cce.pdf (2001).

[11] R. Beckers, J.-L. Deneubourg, and S. Goss. Trails and u-turns in the selection of the shortest path by the ant lasius niger. *J. Theor. Biol.*, 159:397–415, 1992.

[12] R. Beckers, O. E. Holland, and J. L. Deneubourg. *From Local Actions to Global Tasks: Stigmergy and Collective Robotics*. MIT Press, 1994.

[13] Mark A. Bedau. Weak emergence. In J. Tomberlin, editor, *Philosophical Perspectives: Mind, Causation, and World*, pages 375–399. Blackwell, 1997.

[14] G. Beni. From swarm intelligence to swarm robotics. In *Proc. of Workshop on Swarm Robotics, 8th Intl. Conf. on Simulation of Adaptive Behaviour (SAB'04)*. MIT Press, 2004.

[15] Jean Paul Benzecri. *L'analyse des donnèes. II. L'analyse des correspondances.* Dunod, Paris, 1973.

[16] E. Bonabeau, M. Dorigo, and G. Theraulaz. Inspiration for optimization from social insect behaviour. *Nature*, 406:39–42, 2000.

[17] E. Bonabeau, G. Theraulaz, and M. Dorigo. *Swarm Intelligence - From Natural to Artificial Systems.* Oxford University Press, 1999.

[18] Eric Bonabeau. From classical models of morphogenesis to agent-based models of pattern formation. *Artificial Life*, 3(3):191–211, 1997.

[19] Eric Bonabeau, Sylvain Guerin, Dominique Snyers, Pascale Kuntz, and Guy Theraulaz. Three-dimensional architectures grown by simple 'stigmergic' agents. *BioSystems*, 56:13–20, 2000.

[20] Eric Bonabeau, Florian Henaux, Sylvain Guérin, Dominique Snyers, Pascale Kuntz, and Guy Theraulaz. Routing in telecommunications networks with "smart" ant-like agents. 98-01-003, Santa Fe Institute, 1998.

[21] Eric Bonabeau, Andrej Sobkowski, Guy Theraulaz, and Jean-Luis Deneubourg. Adaptive task allocation inspired by a model of division of labor in social insects. In Dan Lundh, Bjorn Olsson, and Ajit Narayanan, editors, *Biocomputing and Emergent Computation*, pages 36–45. World Scientific, 1997.

[22] Eric Bonabeau, Guy Theraulaz, and François Cogne. The design of complex architectures by simple agents. Technical Report 98-01-005, Santa Fe Institute, 1998.

[23] Adrian Bowyer. Automated construction using cooperating biomimetic robots. Technical Report Technical Report 11/00, University of Bath, 2000.

[24] R. M. Brady. Optimization strategies glean from biological evolution. *Nature*, 317:804–806, 1985.

[25] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, 1984.

[26] Rodney A. Brooks. A robust layered control system for mobile robots. *Journal of Robotics and Automation*, RA-2(1), 1985.

[27] Rodney A. Brooks. Intelligence without reason. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann.

[28] Jerome Buhl, Jean-Louis Deneubourg, and Guy Theraulaz. Self-organized network of galleries in the ant messor sancta. In M. Dorigo, G. Di Caro, and M. Sampels, editors, *Ant Algorithms:*, pages 163–175. Springer-Verlag, 2002.

[29] Scott Camazine, Jean-Louis Deneubourg, Nigel Franks, Eric Bonabeau, Guy Theraulaz, and James Sneyd. *Self-Organization in Biological Systems.* Princeton, 1999.

[30] Peter Cariani. *Artificial Life II, SFI Studies in the Sciences of Complexity*, chapter Emergence and Artificial Life, pages 775–797. Addison-Wesley, 1991.

[31] Pablo Miranda Carranza and Paul Coats. Swam modelling – the use of swarm intelligence to generate architectural form. In *Proceedings of Generative Art 2000*, 2000.

[32] Cristiano Castelfranchi. The theory of social functions: Challenges for computational social science and multi-agent learning. *Cognitive Systems*, 2001.

[33] A. D. Channon and R. I. Damper. Perpetuating evolutionary emergence. In *Proceedings of SAB '98*, 1998.

[34] F. Chantemargue and B. Hirsbrunner. A collective robotics application based on emergence and self-organization. In *Proceedings of ICYCS'99, Nanjing, China*, 1999.

[35] Maurice Clerc. When ant colony optimization does not need swarm intelligence (available online at http://clerc.maurice.free.fr/pso/aco/aco_swarm_intelligence.zip), 2000.

[36] P. S. Coates, N. Healy, C. Lamb, and W. L. Voon. The use of cellular automata to explore bottom up architectonic rules, 1996.

[37] Joel E. Cohen. Human population: the next half century. *Science*, 302:1172–1175, November 2003.

[38] Rosaria Conte and Nigel Gilbert. *Artificial Societies: The Computer Simulation of Social Life*, chapter 1, pages 1–15. UCL Press, London, 1995.

[39] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[40] M. D. Cox and G. B. Blanchard. Gaseous templates in ant nests. *J. Theor. Biol.*, 204(2):223–238, 2000.

[41] James P. Crutchfield. The calculi of emergence: Computation, dynamics and induction. *Physica D Special Issue on the Proceedings of the OJI International Seminar*, 1994.

[42] James P. Crutchfield. *Integrative Themes: Santa Fe Institute Studies in the Sciences of Complexity XIX*, chapter Is Anything Ever New? Considering Emergence. Addison-Wesley, 1994.

[43] James P. Crutchfield, J. D. Farmer, N. H. Packard, and R. S. Shaw. Chaos. *Sci. Am.*, 255:46–57, 1986.

[44] Vince Darley. Emergent phenomena and complexity. In R. Brooks and P. Maes, editors, *Artificial Life IV, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 411–416. MIT Press, 1994.

[45] J.-L. Deneubourg and S. Goss. Collective patterns and decision making. *Ethology, Ecology and Evolution*, 1:295–311, 1989.

[46] J. L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chretien. The dynamics of collective sorting: Robot-like ants and ant-like robots. In *From Animals to Animats: Proceedings of the First International Converence on Simulation of Adaptive Behavior*. MIT Press, 1991.

[47] R. Diestel. *Graph Theory*. Springer, 2000.

[48] M. Dorigo and L. M. Gambardella. Ant colonies for the travelling salesman problem. *BioSystems*, 43:73–81, 1997.

[49] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(189):53–66, 1997.

[50] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 26(1):29–41, 1996.

[51] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[52] H. A. Downing and R. L. Jeanne. Nest construction and the paper wast, polistes: A test of stigmergy theory. *Animal Behaviour*, 36:1729–1739, 1988.

[53] K. Eric Drexler. *Engines of Creation: The Coming Era of Nanotechnology*. Anchor, 1986.

[54] Alexis Drogoul and Jacques Ferber. From tom-thumb to the dockers: Some experiements with foraging robots. In *From Animals to Animats II: Proceedings of the Second International Converence on Simulation of Adaptive Behavior*, pages 451–459. MIT Press, 1993.

[55] B. Edmonds. What is complexity? – the philosophy of complexity *per se* with application to some examples in evolution. In F. Heylighen and D. Aerts, editors, *The Evolution of Complexity*. Kluwer, Dordrecht, 1999.

[56] A. Espinas. *Des Sociétés animales – Essais de Psychologie Comparée*. F. Alcan., 1877.

[57] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising genetic algorithm approach to job-shop scheduling, re-scheduling, and open-shop scheduling problems. In Stephanie Forrest, editor, *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pages 375–382, San Mateo, CA, 1993. Morgan Kaufmann.

[58] R. P. Fletcher, C. Cannings, and P. G. Blackwell. Modelling foraging behavior of ant colonies. In *Lecture Notes in Artificial Intelligence 929*, pages 772–783. 1995.

[59] D.B. Fogel. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics*, 60:139–144, 1988.

[60] S. Forrest. Emergent behavior in classifier systems. *Physica D*, 42:213–227, 1990.

[61] Stephanie Forrest. Emergent computation: Self-organising, collective and cooperative phenomena in natural and artificial computing networks. *Physica D*, 42:1–11, 1990.

[62] Mark S. Fox. *Constraint-directed Search: A Case Study of Job-Shop Scheduling*. Morgan Kaufmann Publishers, 1987.

[63] N. R. Franks. Teams in social insects: group retrieval of prey by army ants. *Behav. Ecol. Sociobiol.*, 18:425–429, 1986.

[64] N. R. Franks. Army ants: a collective intelligence. *Am. Sci.*, 77:138–145, 1989.

[65] T. Fukuda, S. Nakaggawa, Y. Kawauchi, and M. Buss. Structure decision method for self-organizing robots based on cell structure – cebot. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 695–700. IEEE Computer Society Press, 1989.

[66] P. Funes and J. Pollack. Computer evolution of buildable objects. In P. Husbands and I. Harvey, editors, *Fourth European Conference on Artificial Life*, pages 358–367. MIT Press, 1997.

[67] P. Funes and J. Pollack. Computer evolution of buildable objects. In P. Bentley, editor, *Evolutionary Design by Computers*, pages 387–403. Morgan Kaufmann, 1999.

[68] Di Caro G. and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research (JAIR)*, 9:317–365, 1998.

[69] Di Caro G. and M. Dorigo. Mobile agents for adaptive routing. In *Proceedings of the 31st Hawaii International Conference on System*, pages 74–83. IEEE Computer Society Press, 1998.

[70] Niloy Ganguly, Biplab K. Sikdar, Andreas Deutsch, Geoffrey Canright, and P. Pal Chaudhuri. A survey on cellular automata. Technical report, Centre for High Performance Computing, Dresden University of Technology, December 2003.

[71] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[72] Nigel Gilbert. *Artificial Societies: The Computer Simulation of Social Life*, chapter Emergence in Social Simulation, pages 144–156. UCL Press, London, 1995.

[73] M. Giurfa and E. Capaldi. Vectors, routes and maps: new discoveries about navigation in insects. *Trends Neurosci*, (22):237–242, 1999.

[74] D.E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Longman, 1989.

[75] D.E. Goldberg and Jr. R. Lingle. Alleles, loci and the tsp. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 154–159. Lawrence Erlbaum, 1985.

[76] P.-P. Grassé. La reconstruction du nid et les coordination inter-individuelles chez bellicositermes natalensis et cubitermes sp. la thérie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insect. Soc.*, 6:41–80, 1959.

[77] David G. Green and David Newth. Towards a theory of everything? – grand challenges in complexity and informatics. *Complexity International*, 8, 2001.

[78] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the tsp. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 160–165. Lawrence Erlbaum, 1985.

[79] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. CRC Press, 2003.

[80] Mika Hirvensalo. Copying quantum computer makes np-complete problems tractable (tucs technical report no. 161). Technical report, Turku Centre for Computer Science, 1998.

[81] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proceedings of the Workshop on Knowledge Discovery in Databases*, pages 169–180, 1994.

[82] L. B. Holder. Empirical substructure discovery. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 133–136, 1989.

[83] Lawrence B. Holder, Diane J. Cook, and Horst Bunke. Fuzzy substructure discovery. In *Proceedings of the Ninth International Machine Learning Conference*, pages 218–223, 1992.

[84] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[85] John Holland. *Emergence: From Chaos to Order*. Addison-Wesley, 1997.

[86] O. Holland and C. Melhuish. Stigmergy, self-organization and sorting in collective robotics. *Artificial Life*, (5), 1999.

[87] Bert Hölldobler and Edward O. Wilson. *Journey to the Ants*. Harvard University Press, 1995.

[88] Uwe Homberg. In search of the sky compass in the insect brain. *Naturwissenschaften*, (91):199–208, 2004.

[89] K. Hosokawa, T. Tsujimori, T. Fujii, H. Kaetsu, H. Asama, and Y. Kuroda. Mechanisms for self-organizing robots which reconfigure in a vertical plane. In T. Lueth, R. Dillman, P. Dario, and H. Worn, editors, *Proceedings Distributed Autonomous Robotic Systems 3, DARS '98*, pages 111–118. Springer-Verlag, 1998.

[90] T.L. Huntsberger, P. Pirjanian, and P.S. Schenker. Robotic outposts as precursors to a manned mars habitat. In *Proc. Space Technology and Applications International Forum (STAIF-2001)*, pages 46–51, 2001.

[91] A. S. Jain and S. Meeran. A state-of-the-art review of job-shop scheduling techniques. Technical report, Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, Dundee, Scotland, 1998.

[92] Chris V. Jones and Maja J. Mataric. From local to global behavior in intelligent self-assembly. In *IEEE International Conference on Robotics and Automation*, pages 721–726, September 2003.

[93] M. Jüunger, G. Reinelt, and G. Rinaldi. *Annotated Bibliographies in Combinatorial Optimization*, chapter The Travelling Salesman Problem. J. Wiley & Sons, 1997.

[94] I. Karsai and Z. Pénzes. Comb building in social wasps: self-organization and stigmergic scripts. *J. Theor. Biol.*, 161:505–525, 1993.

[95] Istvan Karsai. Decentralized control of construction behavior in paperwasps: an overview of the stigmergy approach. *Artificial Life*, 5:117–136, 1999.

[96] István Karsai and Zsolt Pénzes. Nest shapes in paper wasps: can the variability of forms be deduced from the same construction algorithm? *Proc. R. Soc. Lond.*, 265(B):1261–1268, 1998.

[97] István Karsai and Zsolt Pénzes. Optimality of cell arrangement and rules of thumb of cell initiation in *Polistes dominulus*:a modelling approach. *Behav. Ecol.*, 11(4):387–395, 1999.

[98] István Karsai and Guy Theraulaz. Nest building in social wasps. *Sociobiology*, 265:83–114, 1995.

[99] Istvan Karsai and John W. Wenzel. Organization and regulation of nest construction behavior in *Metapolybia* wasps. *Journal of Insect Behavior*, 13(1), 2000.

[100] Stuart Kauffman. *At Home in the Universe: The Search for the Laws of Self-Organization and Complexity*. Oxford University Press, 1995.

[101] S. Kirkpatrick, C. D. Gelati, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[102] J. Kolen and J. Pollack. The observers' paradox: Apparent computational complexity in physical systems. *Journal of Exp. and Theoret. Artificial Intelligence*, 7(3), 1995.

[103] T. Krink and F. Volrath. Analysing spider web-building behaviour with rule-based simulations and genetic algorithms. *J. Theor. Biol.*, 185:321–331, 1997.

[104] C. R. Kube and H. Zhang. Collective robotics: From social insects to robots. *Adaptive Behaviour*, 2:189–218, 1994.

[105] C. R. Kube and H. Zhang. Task modelling in collective robotics. *Auton. Robots*, 4:53–72, 1997.

[106] C. Ronald Kube and Eric Bonabeau. Cooperative transport by ants and robots. *Robotics and Autonomous Systems*, 30:85–101, 2000.

[107] C. Ronald Kube and Hong Zhang. Collective robotic intelligence. In *Proceedings of the 2nd International Workshop on the Simulation of Adaptive Behaviour*, pages 460–468. MIT Press, 1992.

[108] P. Kuntz, P. Layzell, and D. Snyers. A colony of ant-like agents for partitioning in vlsi technology. In P. Husbands and I. Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life*, pages 417–424. MIT Press, 1997.

[109] C. G. Langton. Studying artificial life with cellular automata. *Physica D*, 22:120–149, 1986.

[110] C. G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D*, 42:12–37, 1990.

[111] P. Larrañaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13:129–170, 1999.

[112] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, 1985.

[113] E. Lumer and B. Faieta. Diversity and adaptation in populations of clustering ants. In *Proceedings of the Third International Conference on the Simulation of Adaptive Behaviour: From Animals to Animats 3*, pages 499–508. MIT Press, 1994.

[114] M. Lüscher. Air-conditioned termite nests. *Sci. Am.*, 205:138–145, 1961.

[115] Zachary Mason. Programming with stigmergy: Using swarms for construction. In Standish, Abbass, and Bedau, editors, *Artificial Life VIII*, pages 371–374. MIT Press, 2002.

[116] Maja J. Mataric. Designing emergent behaviors: From local interactions to collective intelligence. In *Animals to Animats 2: Proceedings of the second international conference on simulation of adaptive behaviour*, pages 432–441. MIT Press, 1993.

[117] Maja J. Mataric. Designing and understanding adaptive group behavior. *Adaptive Behaviour*, 4(1):51–80, 1995.

[118] Maja J. Mataric and Matthew J. Marjanovic. Synthesizing complex behaviors by composing simple primitives. In *Proceedings, Self Organization and Life, From Simple Rules to Global Complexity, European Conference on Artificial Life (ECAL-93)*, pages 698–707, 1993.

[119] H. Meinhardt. *The algorithmic beauty of sea shells.* Springer-Verlag, 1998.

[120] C. Melhuish, J. Welsby, and C. Edwards. Using templates for defensive wall building with autonomous mobile ant-like robots. In *Towards Intelligent Mobile Robots (TIMR)*, 1999.

[121] A. S. Mikheyev and W. R. Tschinkel. Nest architecture of the ant formica pallidefulva: structure, costs and rules of excavation. *Insect. Soc.*, 51:30–36, 2004.

[122] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machines. In *Proceedings 1994 IEEE Interational Conference on Robotics and Automation*, pages 441–448. IEEE Computer Society Press, 1994.

[123] Christopher L. Nehaniv. Evolution in asynchronous cellular automata. In *Proceedings of the eighth international conference on Artificial life*, pages 65–73, 2002.

[124] G. Nicolis. *Dynamics of Hierachical Systems.* Springer, 1986.

[125] G. Nicolis and I. Prigogine. *Self-Organization in Non-equilibrium Systems.* Wiley, 1977.

[126] T. O'Conner. Emergent properties. *American Philosophical Quarterly*, 31:91–104, 1994.

[127] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permuation crossover operators on the tsp. In J. J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference*, pages 224–230, 1987.

[128] A. Pamecha, C.-J. Chiang, D. Stein, and G. S. Chirikjian. Design and implementation of metamorphic robots. In *Proceedings 1996 ASME Design Enineering Technical Conference and Computers and Engineering Conference*, pages 1–10. AMSE Press, 1996.

[129] H. Van Dyke Paranuk. Making swarming happen. In *Proc. Conference on Swarming and Network Enabled Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C4ISR)*, 2003.

[130] H. Van Dyke Paranuk and Raymond S. VanderBok. Managing emergent behavior in distributed control systems. 1997.

[131] Chris A. Parker, H. Zang, and C. Ronald Kube. Blind bulldozing: Multiple robot nest construction. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2010–2015, 2003.

[132] J. M. Pasteels, J.-L. Deneubourg, and S. Goss. Self organisation mechanisms in ant societies (i) trail recruitment to newly discovered food sources. In J. M. Pasteels and J.-L. Deneubourg, editors, *From individual to collective behaviour in social insects (Experimentia Supplementum)*, volume 54, pages 155–175. Springer-Verlag, 1987.

[133] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, 1990.

[134] E. Rabaud. *Phénoméne social et société animale*. Paris: F. Alcan., 1937.

[135] T. S. Ray. Evolution, evology and optimization of digital organisms. Technical Report 92-08-042, Santa Fe Institute, 1992.

[136] G. Reinelt. Tsplib - a traveling salesman library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[137] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21:25–34, 1987.

[138] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific, 1989.

[139] Edmund M. A. Ronald, Moshe Sipper, and Mathieu S. Capcarrere. Design, observation, surprise! a test of emergence. *Artificial Life*, 5:225–239, 1999.

[140] R. Rucker. *Artificial Life Lab*. The Waite Group Press, Corte Madera, CA., 1993.

[141] R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behaviour*, 5(2):169–207, 1997.

[142] B. Scönfisch and A. de Roos. Synchronous and asynchonrous updating in cellular automata. *BioSystems*, 51:123–143, 1999.

[143] S. K. Scott. *Oscillations, Waves and Chaos in Chemical Kinetics*. Oxford University Press, 1994.

[144] J. Searle. Minds, brains and programs. *The Behavioral and Brain Sciences*, 36:417–424, 1980.

[145] A. P. Smith. An investigation of the mechanisms underlying nest construction in the mud wasp paralstor sp. (hymenoptera: Eumendinae). *Animal Behaviour*, 1978.

[146] R. V. Solé, B. Luque, and S. Kauffman. Phase transitions in random networks with multiple states. Technical Report 00-02-011, Santa Fe Institute, 2000.

[147] Ricard V. Solé, Ramon Ferrer-Cancho, Jose M. Montoya, and Segi Valverde. Selection, tinkering and emergence in complex networks. *Complexity*, 8(1):20–33, 2003.

[148] H. Spencer. *The principles of sociology*. New York, 1882.

[149] Luc Steels. *Cooperation between distributed agents throough self-organization*, pages 175–196. North-Holland, 1990.

[150] Luc Steels. The artificial life roots of artificial intelligence. *Artificial Life*, 1:75–110, 1994.

[151] A. M. Stuart. Alarm, defense, and construction behavior relationships in termites (isoptera). *Science*, 156:1123–1125, 1967.

[152] Tomoaki Suzudo. Searching for pattern-forming asynchronous cellular automata - an evolutionary approach. In *ACRI 2004*, pages 151–160, 2004.

[153] H. L. Swinney and J. P. Gollub. *Hydrodynamic Instabilities and the Transition to Turbulence*. Springer-Verlag, 1981.

[154] Jr. T. A. Witten and Paul Meakin. Diffusion-limited aggregation at multiple growth sites. *Phys. Rev. B*, 28:5632–5642, 1983.

[155] O. Terán, B. Edmonds, and S. Wallis. Mapping the envelope of social simulation trajectories. *Multi Agent Based Simulation 2000 (MABS200) - LNAI*, 1979:229–243, 2001.

[156] G. Theraulaz and E. Bonabeau. Modelling the collective building of complex architectures in social insects with lattice swarms. *J. Theor. Biol.*, 171:381–400, 1995.

[157] G. Theraulaz and E. Bonabeau. A brief history of stigmergy. *Artificial Life*, 5:97–116, 1999.

[158] Guy Theraulaz and Eric Bonabeau. Coordination in distributed building. *Science*, 269:686–688, 1995.

[159] A. Turing. On computable numbers, with an application to the entscheidungs problem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936.

[160] Takeaki Uno and Mutsunori Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.

[161] Peter Wavish. Exploiting emergent behaviour in multi-agent systems. In *Decentralized A.I. 3 – Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agents World (MAAMAW-91)*, pages 297–310, 1991.

[162] R. Wehner. Desert ant navigation: how minature brains solve complex tasks. *J Comp Physiol A*, (189):579–588, 2003.

[163] W. M. Wheeler. The ant colony as an organism. *Journal of Morphology*, 22:307–325, 1911.

[164] G. M. Whitesides, J. P. Mathias, and C. T. Seto. Molecular self-assembly and nanochemistry - a chemical strategy for the synthesis of nanostructures. *Science*, 254:1312–1319, 1991.

[165] Stephen Wolfram. Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55:601–644, 1983.

[166] Stephen Wolfram. *A New Kind Of Science*. Wolfram Media, 2002.

[167] W. A. Wright, R. E. Smith, M. Danek, and P. Greenway. A measure of emergence in an adapting, multi-agent context. In *SAB 2000 Proceedings Supplement*, pages 20–27, 2000.

[168] Andrew Wuensche and Michael J. Lesser. *The Global Dynamics of Cellular Automata*, chapter The Transition Function and Global Dynamics, pages 15–49. Santa Fe Studies, 1992.

[169] T. Yamada and R. Nakano. Genetic algorithms for job-shop scheduling problems. In *Proceedings of Modern Heuristic for Decision Support*, pages 67–81, 1997.

[170] E. Yoshida, S. Murata, K. Tomita, and H. Kurokawa. Distributed formation control for a modular mechanical system. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS '97*. IEEE Computer Society Press, 1997.