Chapter 4

The Nest 3.0 Tool

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 reasonably 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.

TODO:
Screenshots!

4.1 Conceptual Improvements

Whilst algorithms have been developed 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.

4.1.1 Brick Geometry

It was noted when describing the original simulations[**?**] that structures built from hexagonal cells were aesthetically "more biological" than those consisting of cubic bricks. 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*.

With a hexagonal architecture, it is fairly simple to produce a relatively smooth circular shape using a small set of rules, whereas the same shape is

Feature	***Nest-2.11.1***	***Nest-3.0***
Implementation	C++, Unix, X11	Ruby, OpenGL, Fox
Extensibility	Any modification requires recompilation of code & detailed knowledge of the internal system structure	Based on very-high-level scripting language (Ruby), simple to extend
Architecture Types	Cubic, hexagonal; hard-coded into the system	Capable of supporting many architecture types
Agents	A single agent warps between random locations	Any number of agents can be present, and their movement behaviour controlled. Agents may also carry internal state if desired
Brick types	Limited to 4 colours	Virtually unlimited number of colours
Rules	3x3(x3 if 3D), 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 modify any aspect of the architecture building or removing bricks as determined by rules. Agents can also modify any region of their local space, thus building in front of them rather than where their body may be

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

much harder to produce within a cubic system, requiring many more rules to overcome the inherent "blockiness" of the cubic constituent components.

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.

4.1.2 Percieving 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 3x3x3 cube around the agents 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* abstract, disallowing many interesting sense-configurations.

In *Nest-2.11.1*, agents are represented as 0-dimensional points with no associated information: effectively they do not exist. 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 limit 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.

4.1.3 Rotation

When matching a rule against an agents 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. *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 rotation matches then the rule is applied.

Whether or not rotated rules are matched corresponds with an agents ability to distinguish between different global directions. *Nest-2.11.1* allowed rotations around the z-axis only in order to mimic the ability to distinguish compass directions, as social insects are often capable of doing. *Nest-3.0* extends this to allow rotations through the x- and y-axes, which can give the agents the ability to sense an equivalent of gravity, or any other directional field.

4.1.4 Agent Movement

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 *modifies* that environment (see below), and how the agent *moves* within it.

Nest-2.11.1 operates by selecting a single empty location, at random,

and matching rules against that neighbourhood during each simulation cycle. This single agent warping is adequate for investigating the behaviour of very simple stigmergic systems, and in fact sufficient for modelling a strictly sematectonic stigmergic system, but it does not allow designers to exploit the possibilities of the other multiagent systems which it claims to model.

In the new system the user has the option of using any number of agents, each with a specific position within the building volume, more accurately implementing the notion of a "swarm" [ref]. The agents now have the opportunity to interact, changing their behaviour in the presence of other agents, for example.

Furthermore, the movement of the agents can be constrained in a variety of ways. For example, agents may only be allowed to move 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.

Any positioning constraint my be realised within *Nest-3.0* simulations, and it is hoped that the ability to control this 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.

4.1.5 Architecture Modification

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 allow the investigation of excavated structures as well as those built in free space. Furthermore, it allows the *Nest-3.0* to implement a generalised, abstracted stigmergic structural modification model, in which rules simply define modifications of any type to local environments they match against.

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 *in front* of where it is currently located. 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 features both behaviours. This could be used, for example, in the construction and dismantling of temporary structures which guide the further construction of the architecture.

4.1.6 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

More important than this, however, is the ability to specify rules with cells which can match more than one type of brick in any given position. 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 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 has also been increase 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, and cells within rules. Each cell can be given a real number between 0 and 1. 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 greater than 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 below 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 which are "more important" during the rule matching process.

TODO:
the following two subsections might be better placed elsewhere

4.1.7 Architecture Evaluation

The criteria used to evaluate the architectures produced are purely structural (i.e. the existence of patterns within the architecture; modularity) rather

than functional criteria (how effective it would perform as a habitat for social insects; see [Krink 1997] for an example of functional evaluation of structures, using simulated spider-webs).

4.1.8 Algorithm Specification

Finally, and perhaps most importantly, the implications of the assertion regarding coordinated algorithms are left unexplored for instance, no methods other than hand-coding are suggested for the generation of these algorithms.

4.2 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.

4.2.1 Programming Languages and Libraries

Nest-3.0 is being developed using the high-level object-oriented scripting language Ruby[1], along with the FOX GUI Toolkit[2] and OpenGL[3]. A scripting language was chosen over C/C++[**?**] (used in the original *Nest-2.11.1* implementation) because it allows more rapid development, less time building programming scaffolding (developing and debugging complex datastructures), better object-oriented features (leading to better modularity and extendability) and easy modification of the program to test new features.

The sacrifice made for this flexibility is the speed at which the program runs, but to offset this factor much of the system, once prototyped in Ruby, has then been rewritten in low-level C++ and then made available to the high-level system as a loadable module.

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.

4.2.2 System Architecture

Internally the *Nest-3.0* simulation system is split into 3 software modules.

Architectures

The **Architecture** module contains the code specific to maintaining architectural data, including maintain cell relationships, extracting local neighbourhoods from the architecture and matching local environments to rules,

[1] http://www.ruby-lang.org
[2] http://www.fox-toolkit.org
[3] http://www.opengl.org

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.

Within this module, 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 experimented with 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 present here.

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 library. 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.

4.2.3 System Components

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 a `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.

4.3 Implementation Details

The structures in *Nest-3.0* exist within a discrete version of space, split into distinct cells. The design of the software makes no assumptions about the nature of this space however, allowing new geometric arrangements of cells to be used during experimentation. Currently two different cell geometries are available – cubic and hexagonal. Each geometry naturally has different characteristics, including 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.

Feature	**Cubic**	**Hexagonal**
2D Neighbourhood Size	9 cells	7 cells
3D Neighbourhood Size	27 cells	21 cells
Symmetries	x, y, z	z
Rotations	4 around x, 4 around y, 4 around z	6 around z

Table 4.2: Geometric details for cubic and hexagonal architectures.

TODO:
needs to be fleshed out more, or rethought

It is most important the significance of the geometry of individual cells and its effect on the overall architecture produced during a simulated build. For example, a relatively smooth cylinder can be produced with far less

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.

4.3.1 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		$= 10$

Table 4.3: Multi-state matching using Bit Arrays

A match is determined using the following boolean function:

$$match(cell_{rule}, cell_{target}) \iff (cell_{rule} \,|\, cell_{target}) = cell_{target} \qquad (4.1)$$

For example, a rule cell configured to match against `RED`, `BLUE` or `GREEN` target cells:

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

4.3.2 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

$$index(x, y, z) = x + (y \times XSIZE) + (z \times XSIZE \times YSIZE) \qquad (4.2)$$

where $XSIZE$ is the total size of the architecture along the x-axis, and $YSIZE$ is the length along the y-axis.

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 `Architecture` (and therefore also in a `Rule`, since it is a subclass of `Architecture`). As described above, an element in 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.

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 4.4.

Intuitively, some rotations might seem to be missing from this list, but in fact they are 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 4.1.

In order to match a rotated set of cells, rather than actually transform the architecture in place it is simpler and faster to *redirect* coordinate references

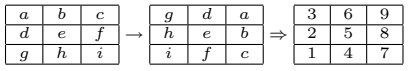
Cubic Rotations

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

for one cell to the cell which *would* be in that position if the architecture was actually rotated.

TODO:
A graphic illustrating this, something like:



An illustration of cell index rotation on a $3 \times 3 \times 1$ matrix

Thus, the `Cell` which was in location index 1 has moved to location index 3, `Cell` 6 has moved to location index 8, and so forth. The true value of a cell (x, y, z) is now determined by calculating the index of the cell at that coordinate location (see Equation 4.2), then transforming that index into the index where the cell would appear under rotation, and finally using this generated *rotated* index to determine the value within the actual array of cell data. The functions which provide these *index transforms* for cube architectures of any size s are given below:

TODO:
I suppose I have to explain where these come from. The truth is that I sat with 3x3x3 matrices and explored the index changes and figured them out from that.

$$RotateX(i, s) = \left(\left(\frac{i}{s} \bmod s\right) + 1\right) \times s^2 - \left(\left(\left(\frac{i}{s^2}\right) + 1\right) \times s\right) + (i \bmod s) \tag{4.3}$$

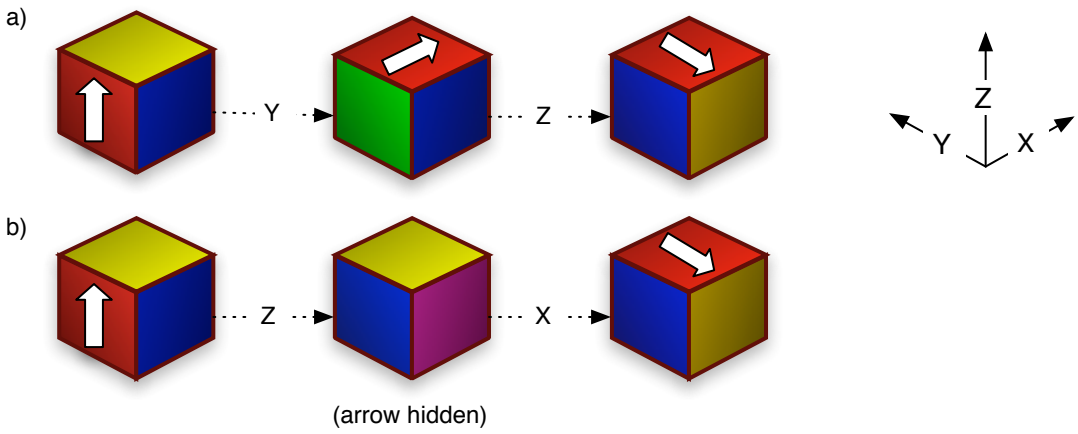


Figure 4.1: Rotational Equivalence. Cubes *a)* and *b)* are initially identical. Cube *a)* is rotated once through the y-axis, and then once through the z-axis. Cube *b)* is rotated once through the z-axis, and then once through the x-axis. Both result in identical orientations. It is important to note that the axes *do not rotate* with the cube.

TODO:
Provisionally: The first part of this equation offsets our index for the z coordinate component of the cube. The second element offsets the y component. not these equations only work on cubes. I have specified this in the text. however, i did make a song and dance about how rules could be any shape. It's important to note you can't rotate a rule around a face that isn't square (as I remember), but if this is true then it shouldn't be hard to extend these formulae to include cuboids with at least one face as a cube (the trick would be replacing s^2 with something like $XSIZE \times YSIZE$. worth the hassle?

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

TODO:
This one is more complex... I dreamt these up almost 2 years ago

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

TODO:
This one seems like the maddest of the lot, without discernible x, y or z components (but then it is rotation through the z axis so it's bound to be mad.

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 repetition of the above calculations.

Finally, 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 4.4. 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.

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.

4.3.3 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.

Rotation of Hexagonal Structures

Hexagonal neighbourhoods have only one axis of symmetry, namely the z-axis passing through the central cell. Through this axis, a hexagonal rule can be rotated 6 times

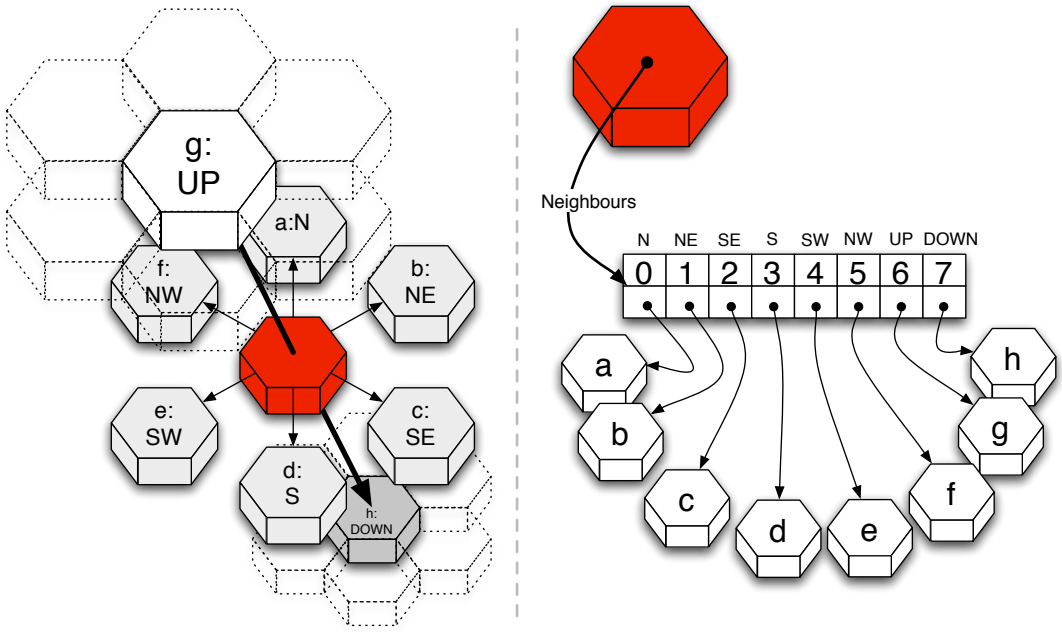


Figure 4.2: Inter-`Cell` linkage in hexagonal geometries. Each `Cell` is linke to the 8 face-adjacent cells – six within the same plane of cells, one above and one below.

4.4 *Nest-3.0*: An Abstract Rule System

Despite these apparent complications, Nest 3.0 remains at heart an abstract rule-based system which simply matches and modifies configurations of states to a subset of the states contained in a system. The constraints of the original system have been relaxed such that now neighbourhoods of any size can be matched, and the number of states available is effectively unlimited.

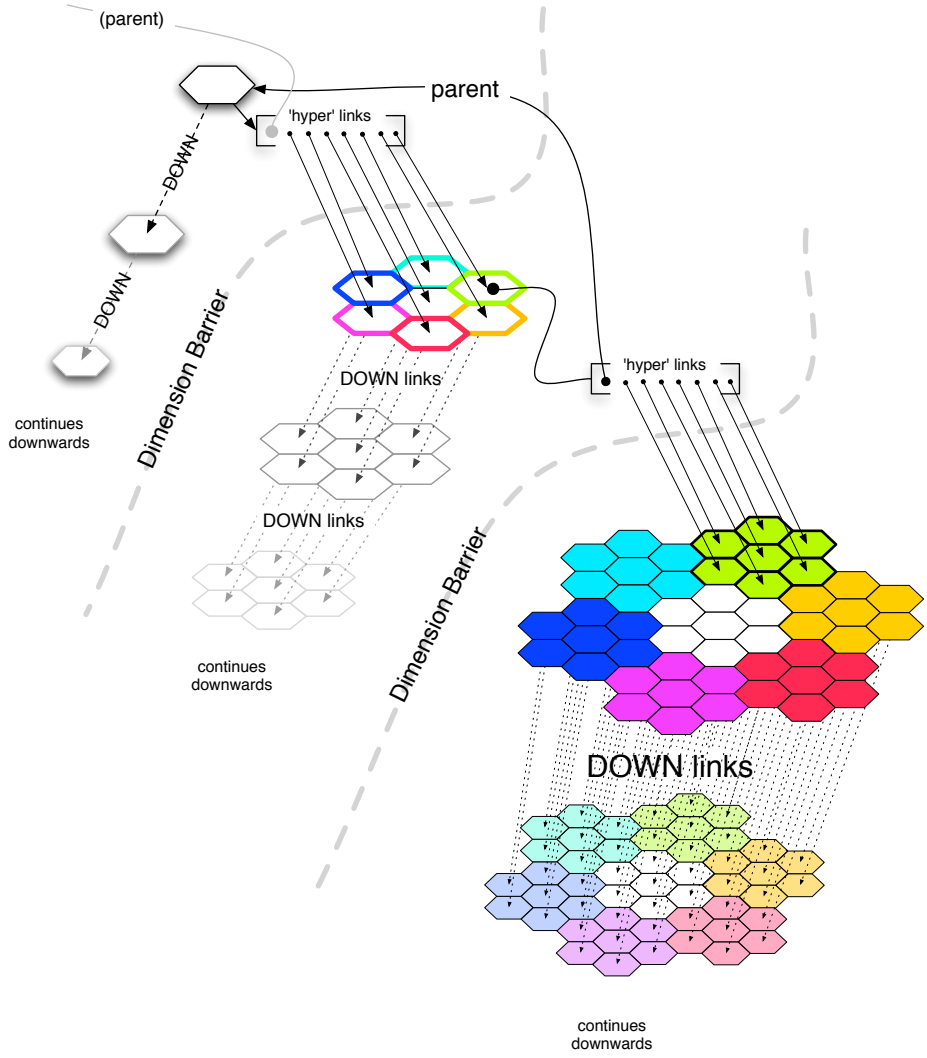


Figure 4.3: The HyperStructure used to construct the interlinked `Cell` network. The lattice space is *expanded* through dimensions, each dimension using neighbourhood information from the dimension above in order to ensure that inter-cell links are valid in the final "real space" dimension.