A COMPREHENSIVE FRAMEWORK FOR THE SNAKE-IN-THE-BOX PROBLEM

by

Christopher A. Taylor

(Under the direction of Don Potter)

Abstract

The snake-in-the-box problem was posed by Kautz [5] in the context of coding theory in 1958 and is essentially the problem of finding the *longest* path in an n-dimensional hypercube subject to a simple set of constraints. Finding the longest path for dimensions greater than 7 is an open problem and cannot be solved with exhaustive search techniques due to explosiveness of the search space. Many interesting and creative techniques have been used to try to solve this problem, and this thesis continues that tradition by developing a novel computational and visualization framework aimed to speedily facilitate the evaluation of computational heuristic ideas.

Index words: Hypercube, Heuristic Search, Graph Theory,
Visualization, Constraint Satisfaction

A COMPREHENSIVE FRAMEWORK FOR THE SNAKE-IN-THE-BOX PROBLEM

by

Christopher A. Taylor

B.S., Western Carolina University, 1998

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2007

A COMPREHENSIVE FRAMEWORK FOR THE SNAKE-IN-THE-BOX PROBLEM

by

Christopher A. Taylor

Approved:

Major Professor: Don Potter

Committee: Khaled Rasheed
Robert Robinson

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2007

Dedication

To my wife Christa and daughters Sarah and Sage, without them in my life I would be uninspired.

Acknowledgments

This thesis would not have been possible without the help of many people. I would especially like to thank Dr. Potter for steering me in a practical direction. Thanks to Dr. Robinson for taking time to speak with me about some of the mathematical aspects of this problem. Dr. Rasheed also offered motivational words at a time in which my project was staggering.

I have to thank Dr. Lee Pratt and his wife Dr. Marie-Michele Pratt. They have been so supportive of my effort to complete this thesis that it is impossible for me to adequately characterize their support with words. My current employer, Dr. Steven J. Knapp was very understanding and gave me the time I needed to finish. In the very early stages of the project, a very nice lady named Jeannie McElhannon allowed me to work in a nice executive office where I scratched out the initial ideas for the visualization environment.

Finally, my wife has been so patient throughout this entire graduate experience. Her efforts have kept me balanced during times when balance was incredibly difficult to maintain.

Table of Contents

Page

Acknowledgments v

Chapter

1 INTRODUCTION 1

2 OVERVIEW OF THE FRAMEWORK 5

2.1 RELEVANT APPROACHES IN THE LITERATURE 9

2.2 INTERACTIVE VISUALIZATION ENVIRONMENT BASICS 11

2.3 'INTELLIGENT' FEATURES OF THE FRAMEWORK . . . 14

2.4 PARTIAL PATH BRANCH AND BOUND SEARCH 19

3 BENEFIT OF THE PARTIAL PATH APPROACH 26

4 A CASE STUDY 36

4.1 GAINING INTUITION 36

4.2 THE K-CUBE HEURISTIC 39

4.3 THE ITERATIVE IMPROVEMENT HEURISTIC 41

4.4 THE CASE FOR THE GENETIC ALGORITHM AND SIMULATED ANNEALING 49

5 CONCLUSION AND FUTURE DIRECTION 52

Bibliography 55

Appendix

A USER GUIDE 57

A.1 PPBBSM USER GUIDE 57

A.2 IVE USER GUIDE 58

B BASIC MODEL OF THE FRAMEWORK 61

C EXAMPLE INTERFACE IMPLEMENTATIONS 64

C.1 FORWARD NEIGHBOR SELECTOR 64

C.2 UPPER BOUND ESTIMATE 65

D PPBBSM SOURCE CODE 68

E VALIDATION JUNIT TEST CASE EXAMPLE 70

Chapter 1

INTRODUCTION

The purpose of this research is to develop a comprehensive framework to help future researchers solve the snake-in-the-box problem. The term "comprehensive" is used because the framework strives to bridge the gap between theoretical mathematical approaches and purely heuristic approaches. The snake-in-the-box problem is that of finding the longest open-path in a multi-dimensional hypercube. In order to understand the contributions of the framework, it is important to first understand snakes and hypercubes. In [3], Harary introduces standard hypercube terminology. Q_n represents an *n-dimensional* hypercube. The vertices of the hypercube can be labeled with binary digits. For dimension n, the Q_n graph has 2^n vertices. All edge connected vertices have a hamming distance of 1. Hamming distance refers to the number of bits in which two binary digits differ. For example, the binary digits 0001 and 0000 have a hamming distance of 1 while 0001 and 1110 have a hamming distance of 4. The hypercube is defined recursively.

$$Q_1 = K_2; \;\; Q_n = K_2 \times Q_{n-1} \tag{1.1}$$

Equation 1.1 can be understood by imagining the complete graph on 2 vertices (a line segment along the x-axis). To get from Q_1 to Q_2, draw lines from the vertices of Q_1 in the direction of the y-axis. Thus, Q_2 is a square with a vertex at each corner. To get from Q_2 to Q_3 (a 3 dimensional cube), extend the square towards the z-axis. In general, going from dimension $n-1$ to n involves projecting a copy of Q_{n-1} onto some new axis and connecting all the vertices between the Q_{n-1} graph and this projected copy.

Table 1.1: Moves for an optimal dimension four snake

Snake Move (edge)	**Skin Set**	**Possible Move**
0000-0001	0010,0100,1000	0011,0101,1001
0001-0011	0010,0100,1000,0101,1001	0111, 1011
0011-0111	0010,0100,1000,0101,1001,1011	0110, 1111
0111-0110	0010,0100,1000,0101,1001,1011,1111	1110
0110-1110	0010,0100,1000,0101,1001,1011,1111	1010,1100
1110-1100	0010,0100,1000,0101,1001,1011,1111,1010	1101
1100-1101	0010,0100,1000,0101,1001,1011,1111,1010	

The snake-in-the-box problem has been described quite well in [8] and [9], but an example that shows how hypercube vertices become inaccessible as a result of "moves" of the snake will serve as a useful reference. Figure 1.1 shows a four-dimensional hypercube in two panels that are side by side. The image on the left panel shows binary labeled vertices with the right-most bit being the least significant. The snake starts from vertex labeled '0000' and proceeds to '0001'. The basic rule of the snake-in-the-box problem is that any vertex adjacent to a vertex on the snake path is inaccessible. Inaccessible vertices are known as the "skin". At times, it will be useful to refer to the collection of inaccessible vertices as the skin set. Table 1.1 demonstrates a list of moves showing a growing skin set when finding the optimal dimension 4 snake.

Students learning about the snake-in-the-box problem commonly ask if the choice for the start vertex makes a difference. The answer is no because of the symmetric nature of the hypercube. This has led to the definition of a *canonical* snake. A canonical snake always starts from the 0 vertex. Another defining characteristic of the canonical snake limits the number of early choices a snake has. For example, table 1.1 shows that the first move is from 0000 to 0001. Actually, the only choice for a canonical snake is 0001 because the rule states that if the snake aims to climb to a higher dimension (i.e., flip a bit to 1), and there is a choice

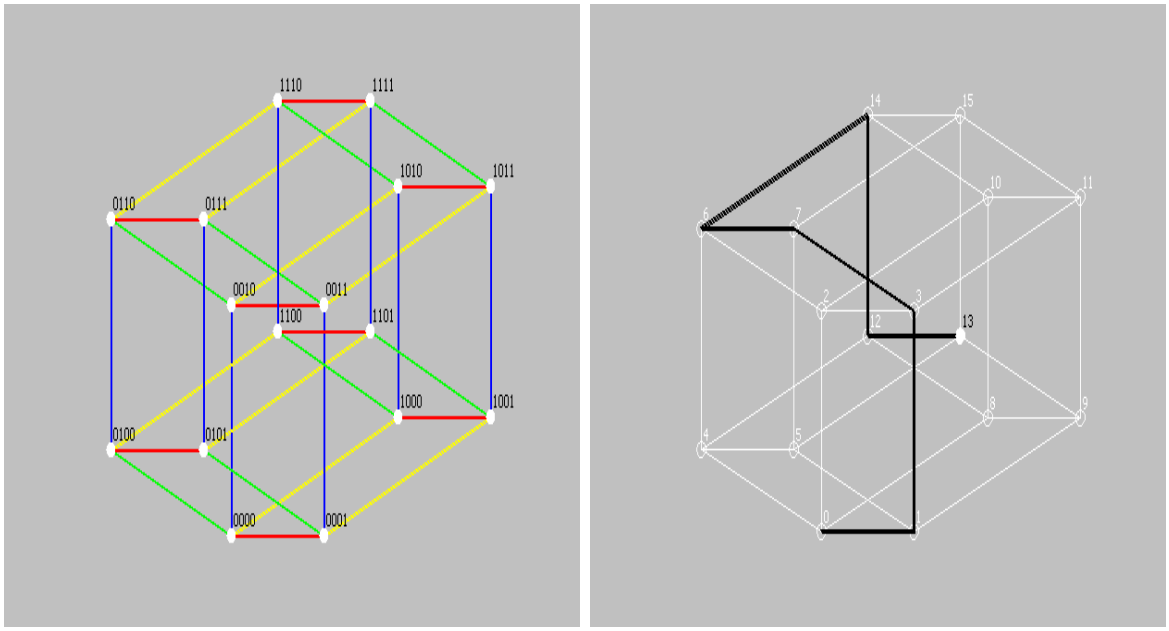

Figure 1.1: An example of a 4-dimensional hypercube with binary (left) and integer labeled vertices (right). All canonical snakes start from 0000. The edges are color coded so that each color corresponds to a change in a particular bit/dimension. For example, all yellow edges represent jumps along the 4th dimension. The image to the right shows (in black) the edges that compose the snake built from Table 1.1. While the path is forming, any vertex that is adjacent to a vertex on the snake path but is not on the snake path becomes inaccessible. Inaccessible vertices are referred to as skin, and the white edges connected to skin vertices are also inaccessible.

among higher dimensions that have not been visited (in this case the choices are dimensions 1 (0001), 2 (0010), 3 (0100) and 4 (1000) because no bits have been flipped before the first move, then the snake must climb to the lowest of the unvisited dimensions (dimension 1). In [6], Kochut discusses the reduction in search space due to the $n!$ symmetries of the n-cube.

Chapter 2

OVERVIEW OF THE FRAMEWORK

The framework can be thought of as a heuristic development environment. Its goal is to inspire new heuristic ideas that improve our understanding of the snake-in-the-box problem. It offers tools and APIs that help researchers gain intuition for snake paths in multi-dimensional hypercube space. It also offers a branch and bound search module and an interactive visualization environment (IVE). Each module can be used independently or can be made to work together. This thesis will describe how this interaction can take place, but for now consider only the branch and bound search module as a standalone piece of functionality. Wikipedia [11] describes branch and bound perfectly as itp ertains to the framework. "Branch and bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded en masse, by using upper and lower estimated bounds of the quantity being optimized" (Wikipedia, 2007). The BB implementation in the framework attempts to optimize the length of the longest snake and supports the ability to pre-define a set of partial paths in the hypercube through which the snake path is constrained. Henceforth, the search module will be referred to as PPBBSM to mean the partial path branch and bound search module. The PPBBSM is effective because the environment around the partial path ends improves the ability to reason about the upper bound snake length for a given partial path configuration.

The PPBBSM works by doing a depth first search from a pre-defined start (forward) path from the 0 vertex and automatically connects and merges with partial paths when

necessary. A conservatively defined start path is (0-1-3-7); the initial path shared by all canonical snakes. After each automated move, many constraints are checked and the search continues if none are violated. The PPBBSM also works with what it calls a backward path. A backward path can be induced as a result of the systematic forward path moves enumerated by the PPBBSM. When a partial path is first defined in the initial configuration, one of its partial path ends may have several accessible neighbors. However, as the PPBBSM grows the forward path via the depth first search, this partial path end's neighbors may all become inaccessible (skin). When this happens, the PPBBSM can recognize that the partial path end would be a dead end for the snake when the forward path is forced to merge with it. Thus, what once was a partial path can end up becoming the backward path. Once this happens, there can be no other dead ends for the other partial paths because the final snake cannot have two terminal endpoints. It can only have one starting vertex (0 for canonical snakes) and one terminal vertex. If this were to happen, the PPBBSM could backtrack immediately.

Throughout the discussion thus far, there has been constant mention of an initial partial path configuration. A fair question concerns the source of the initial partial path configuration. What are the rules that govern a good initial set of partial paths? It seems irrelevant that the PPBBSM improves the time it takes to find an optimal snake through an initial configuration if the initial configuration is itself flawed. The same questions also pertain to the notion of "seeding" found in the literature. In [1] for example, good snakes from lower dimensions are used as seeds in higher dimension searches. In these cases, researchers are hypothesizing about what makes a good initial seed. It would be nice to be able to validate such hypotheses with intuition gained from visual evidence. The k-cube heuristic discussed in chapter 4 provides an example of how the IVE supports this kind of validation. This approach offers a generalization of an initial seed which is shown to produce interesting results.

The issue of partial path placement was another motivating factor for the development of the interactive visualization environment (IVE). The features of the IVE are discussed in

detail in section 2.2, but the main idea is that it is an environment that provides information about partial path configurations. The IVE visually displays properties that can be of direct use in the development and validation of heuristics. Users can arbitrarily create any legal partial path configuration and all defined paths are interactively editable. The IVE does not allow illegal snake-in-the-box moves and at times provides an informative error message when the user tries to make such moves.

The framework's comprehensive label comes from the fact that its main components are highly extendable in a cooperative fashion. The PPBBSM enables a speedy answer to the question of the longest possible snake through a given set of partial paths (chapter 3). An uninformed approach would be to perform a basic depth-first search using nothing more than an adjacency matrix. Appendix B shows that the PPBBSM relies on a sophisticated object-oriented model of a hypercube and its paths to give information about the promise of a particular line of search. The uml class model in appendix B.1 shows that this model has the vocabulary to check for the satisfaction of many types of constraints. Section 2.4 discusses the main constraints made possible by the object-oriented model. These constraints are checked after each forward path move. As such, the PPBBSM can be used as an evaluation measure for other heuristic search strategies because it significantly improves the time it takes to find the longest snake through the initially defined paths. This functionality supports viewing the problem from the perspective of how to evolve the ideal placement of a small number of initial partial paths. The IVE allows for the meaningful visualization and validation of a partial path placement strategy. Chapter 4 introduces an iterative improvement heuristic approach that elegantly solves Q_7 and performs respectably in Q_8. When solving dimension 7, the iterative improvement technique evolved partial path configurations containing 3 partial paths with only 3 vertices specified on each path. This approach was inspired by intuition gained from working with the IVE. The extension points have been carefully considered so that the framework can also be of relevant use to future researchers. The PPBBSM can be extended in two main ways. As previously described, the PPBBSM uses a depth-first search

from the forward path through the hypercube. When necessary, it merges with partial paths along the way. By default, the PPBBSM exhaustively tries every accessible neighbor of every vertex on the forward path end. The existence of partial paths reduces the branching factor, but this reduction is still insignificant in higher dimension hypercubes. The framework offers utility methods that supply valuable information that can be used to limit the number of choices to consider at each step. For example, figure B.2 shows that the SnakeUtils java class can return the mean degree in terms of accessible vertices for a given list of vertices. This could be used as part of a heuristic approach that ranks alternative move choices. The SubCubeGenerator allows one to track properties of any subcube in the hypercube. This was useful when implementing the k-cube heuristic, which constrains the snake to follow an initial path defined by a chain of sub cubes. The framework defines a Java interface called the ForwardNeighborSelector. Any implementation of the ForwardNeighborSelector has access to these utilities. The choices can be limited by writing a ForwardNeighborSelector implementation that returns only a subset of all neighbors from the forward path end. The PPBBSM visits only the forward neighbors provided by an implementation of this interface. The k-cube heuristic approach is also an example of how intuition from the IVE can lead to an informed implementation of the ForwardNeighborSelector interface. This approach allows for the specification of a high level search plan that defines the initial "shape" of the snake's start path. It is the IVE that gives us the ability to meaningfully discuss notions such as the shape and rationale (see Section 4.1) of successful snakes. This offers an unprecedented vocabulary to the heuristic development effort. It is in this sense that we refer to the framework as a heuristic development environment.

The second extension point is the UpperBoundEstimate interface. The PPBBSM backtracks when constraints are violated. A thorough treatment of these constraints can be found in section 2.4, but the upper bound constraint is one such constraint that is checked after every forward move of the PPBBSM. The PPBBSM is initialized with a configuration file (see Appendix A.1) that allows for the specification of a desired lower bound. For example,

if the lower bound property had the value of 25 for a search in dimension 6, the PPBBSM would not return a snake with a length less than 25. The idea is that a researcher's growing mathematical understanding of the problem can be encapsulated in the implementation of the UpperBoundEstimate Java interface. Section 2.4.1 shows that the UpperBoundEstimate implementation can work with properties of a partial path configuration to produce an estimate for an upper bound. When the upper bound estimate falls below the desired lower bound, the PPBBSM backtracks. A more sophisticated upper bound implementation means more fruitless regions of the search space can be discarded by the PPBBSM when the estimate falls below the desired lower bound. This is the way in which the framework bridges the gap between a theoretical mathematical approach and traditional heuristic approach. The reasoning in section 2.4.1 is an example of the type of theoretical mathematical reasoning that the UpperBoundEstimate extension point allows.

2.1 RELEVANT APPROACHES IN THE LITERATURE

The idea for the extensible framework has been inspired by some truly creative work from previous researchers. In [1], Casella and Potter use a population-based stochastic hill-climber (PBSHC) as an evolutionary approach to finding snakes. The PPBBSM and IVE have been motivated by the desire to improve fitness evaluation measures for creative approaches such as the PBSHC.

In [6], Kochut is actually searching for longest *coils*, which are like snakes but must return in the end to the start node (closed-path). He discusses pruning snakes that cannot possibly be closed to form a coil. After each move, he checks the environment to see that it is still possible to form a closed path. The notion of continuous constraint checking has inspired many features found in the framework.

In [8], Dayanand Rajan introduces the notion of a reversible snake. He discusses that in even dimensions $d \leq 6$, there is a unique longest d-snake. This longest snake happens to be a reversal of itself as witnessed by a suitable permutation of d. In order to understand this, it is

important to understand the difference between the transition sequence and node sequence ways to represent the snake. In Table 1.1, the snake that is constructed has a decimal node sequence of (0, 1, 3, 7, 6, 14, 12, 13). However, if you align the binary labels of vertices of the snake path from top to bottom, you see that each binary digit differs by one bit from the digit(s) it is next to. Observe the binary node sequence below, the bit that is flipped represents the snake move (or edge) in the transition sequence representation. The integers in parentheses represent the growing snake as a transition sequence.

```
0000
0001   ... bit 1 is flipped ... (1)
0011   ... bit 2 is flipped ... (1, 2)
0111   ... bit 3 is flipped ... (1, 2, 3)
0110   ... bit 1 is flipped ... (1, 2, 3, 1)
1110   ... bit 4 is flipped ... (1, 2, 3, 1, 4)
1100   ... bit 2 is flipped ... (1, 2, 3, 1, 4, 2)
1101   ... bit 1 is flipped ... (1, 2, 3, 1, 4, 2, 1)
```

This yields a *transition* sequence of (1, 2, 3, 1, 4, 2, 1). The unique canonical Q_6 optimal snake has the transition sequence found below ...

```
1,2,3,4,2,1,5,4,1,6,5,1,2,    4,5,1,3,5,2,1,5,4,2,6,4,5
```

The gap in the middle is the halfway mark for the snake. Rajan observed that for even dimensions $d \leq 6$, the last half of the snake from right to left is a permutation of the first half of the snake from left to right. To understand this, reverse the list of transition values found in the right half and place them under the transition values on the left to get the following...

```
1,2,3,4,2,1,5,4,1,6,5,1,2
5,4,6,2,4,5,1,2,5,3,1,5,4
```

Everywhere there is a one on the top row; there is a 5 beneath it. Everywhere there is a 2 on the top row, there is 4 beneath it. Everywhere there is 3 on the top row, there is a 6 beneath it. Everywhere there is a 4 on the top row, there is a 2 beneath it. Everywhere

there is 5 on the top row, there is 1 beneath it. Everywhere there is a 6 on the top row, there is a 3 beneath. This shows that the last half of the snake is a permutation of the first half. Rajan used this idea to develop an algorithm that finds snakes that are "reversible". The advantage of his approach is that it narrows the search to a relatively small class of snakes that can be more easily enumerated. The validation of this approach comes from the fact that it led to finding a length 97 snake in Q_8. After more than five years, this is still the Q_8 lower bound.

Table 2.1 shows that the forward and backward snakes are also reversible in the sense of hamming distance from each end. As the snake grows from the start vertex 0, the 3^{rd} column in the table below displays the hamming distance of each vertex from 000000. In the 4^{th} column starting from the bottom, the numbers correspond to the hamming distance of each snake vertex from the terminal vertex 10 (001010). This could lead to a simpler algorithm for reversible snakes than the one described in [8].

Finally, in [9], Tuohy et al. take the approach of introducing constraints based on heuristics such as "tightness" that enable considerable pruning of the search space. The PPBBSM and IVE can help to validate such heuristic ideas.

2.2 INTERACTIVE VISUALIZATION ENVIRONMENT BASICS

The IVE is a modular GUI application written with Java Swing. Appendix A.2 provides a user guide for the IVE. Along the top and bottom are extensible control panels that respond to user events. This section contains a brief overview of some of the features and controls of the IVE, but a detailed account is not necessary in order to understand the following main contributions the IVE makes to the overall framework.

1. The IVE displays useful information about properties of snakes and hypercubes that can help a user design and evaluate heuristics for their own research.

Table 2.1: This table shows that the optimal snake in Q_6 has complete forward and backward hamming distance agreement. If you read the 3^{rd} column from top to bottom, it is the same as the numbers in the 4^{th} column from bottom to top.

Decimal	Binary	Hamming Distance from 0	Hamming Distance from 10
0	000000	0	2
1	000001	1	3
3	000011	2	2
7	000111	3	3
15	001111	4	2
13	001101	3	3
12	001100	2	2
28	011100	3	3
20	010100	2	4
21	010101	3	5
53	110101	4	6
37	100101	3	5
36	100100	2	4
38	100110	3	3
46	101110	4	2
62	111110	5	3
63	111111	6	4
59	111011	5	3
43	101011	4	2
41	101001	3	3
40	101000	2	2
56	111000	3	3
48	110000	2	4
50	110010	3	3
18	010010	2	2
26	011010	3	1
10	001010	2	0

2. The IVE provides novel interactive path editing controls that can actually facilitate an intuition for multi-dimensional hypercube space.

3. This intuition is of practical use when coming up with rules that constitute favorable initial partial path configurations which can then be submitted to the PPBBSM to yield the longest snake through such an initial configuration.

Figure 2.1 shows a screen shot of the interactive visualization environment. The controls along the top and bottom are useful for defining a snake and understanding the effects of its moves. The top panels include the ability to define a snake from the zero vertex (start path), introduce and track arbitrary partial paths, and optionally define a "Backward" path from a hypothetical terminal vertex. The benefits of offering this functionality will become clear in Section 2.3 discussing the specifics of the upper bound estimate. There are other features that affect the display, but the final button along the top is the "Clone" button. This button is useful when a user would like to fork a copy of a given situation in order to study the effects of taking alternative paths. Pressing the clone button also writes the current snake in text form to the hard drive. This text file can then be used as input into the PPBBSM.

Along the bottom are controls that aid in visualization and display information that may be used in heuristic calculations. Hypercubes above $d = 3$ are somewhat confusing to view on a two-dimensional surface (computer screen). It is considerably less confusing if some of the edges can be removed. The first two fields on the lower control panel address this issue. The "PosEdges" box takes a comma separated list of integer values which correspond to the absolute value of the difference between the decimal label of the vertices the edge connects. The "NegEdges" field allows control over which inaccessible edges to display. For example, in figure 2.1 one would ordinarilly expect there to be a negative edge from vertex 1 to vertex 17. This is because the edge from 1 to 17 was made inaccessible by the start path moves from 0-1-3. If you take the absolute value of the difference between the vertices that make up the edge $|1 - 17| = 16$, you will only see that edge if the 16 is among the comma-separated values in the NegEdges box. The same is true for the PosEdges text box.

The “Binary” check box shows all vertices with their binary labels. The “Schema” field is a very useful way to focus on sub-areas or hyperplanes in the hypercube. The user can specify a comma separated list of schemata with syntax consistent with that found in [10]. The “Filter” check box toggles the hyperplane filter functionality. Playing with this feature inspired the k-cube heuristic discussed in section 4.2. “Show Negative” determines whether the inaccessible edges are white, or if they appear colored like the accessible edges. The “ShowDist” check box appends a bracketed number to the display of all vertex labels. This number corresponds to the distance each vertex is from the end of the start path. It was this distance display that led to the discovery of the forward-backward distance agreement discussed in section 2.1. “ShowDegree” appends a parenthetical integer to the vertex labels referring to the number of accessible vertices connected to each vertex.

2.3 ’INTELLIGENT’ FEATURES OF THE FRAMEWORK

When a user edits a path in the IVE, the resulting display offers a window into the complexity of the PPBBSM. For example, it was previously stated that a backward path can be induced as part of the enumeration steps of the PPBBSM. When a user edits a path in the IVE that leaves one of the partial paths with no remaining accessible neighbors, the IVE will actually delete the partial path from the list of partial paths and show it in the backward path text field. Both the IVE and the PPBBSM get information from objects that model and abstract the complexities of the hypercube and its set of partially defined paths (see appendix B. The impetus to build a snake with three section types (start path, internal path(s), and backward path) comes from the recognition that global information about the upper bound for a given partial snake can more easily be derived from analyzing properties local to all partial path ends. After each move, the local environment is reassessed and both the current snake length and upper bound estimate are displayed in the upper left corner of the visualization panel. Feedback from the IVE was very useful when designing and debugging the standard upper bound estimate implementation used by the PPBBSM.

The presence of a backward path offers significant pruning possibilities of the overall search space. The backward path starts from what is intended to be the terminal vertex of the final snake, so right away any other accessible vertex that is not on a partial path with only one accessible neighbor can be pruned. This is because a snake path could only reach such a vertex through its only accessible neighbor, but will have no outlet once it arrives. Thus, a vertex with only one accessible neighbor would have to be a dead end. Thus, all vertices of degree one (i.e., one accessible neighbor) can be pruned given the presence of a defined backward path. One may think this to be trivial because a backtracking search could backtrack out of all dead ends immediately. The value comes from the fact that the algorithm for estimating the upper bound is a function of the number of vertices and properties of those vertices that remain accessible. The fewer vertices there are to inspect, the quicker the upper bound estimate can decide to terminate the search because the estimate falls below the desired lower bound threshold. The beauty of this approach is that there is often a recursive effect that allows the framework to prune significant regions of the search space.

Figures 2.1 and 2.2 demonstrate advanced pruning functionality that would take place after the first depth first search expansion of the PPBBSM from this initial configuration. Q_5 is easily exhaustively searched and its longest path is actually represented by a set of eight canonical length 13 snakes. Figure 2.1 shows the result of defining an initial start path of (0,1,3,7) and an internal partial path containing vertices (10,26,18). This could constitute an initial configuration supplied to the PPBBSM. The first thing the PPBBSM would do is ask its implementation of the ForwardNeighborSelector for a list of forward neighbors to put on the depth-first search stack. The default implementation used by the PPBBSM selects and returns both canonical forward neighbors of 7 (6 and 15). The PPBBSM would then grow the forward path to (0,1,3,7,6) by following the first of the two alternatives. Incidentally, the default implementation of the ForwardNeighborSelector returns the alternatives sorted in ascending order.

Figure 2.1 shows an initial configuration of the standard canonical start path (0,1,3,7) and a partial path of (10,26,18), which appears in the "Select Partial Path" drop down. The upper bound estimate is 17. A more detailed discussion of the mathematical basis of this estimate will be offered in Section 2.4.1. Figure 2.2 shows an incredible difference from the previous figure. All that remains is one vertex that joins the two paths; the rest of the accessible vertices have been completely pruned. It seems surprising that as soon as the user types "6" and hits "enter" in the start path text box, this result is instantly displayed. The key to why the PartialPathHyperCubeWrapper was able to prune so much lies in the fact that it recognized that the partial path had to be converted into a backward path. In fact, the "Select Partial Path" drop down is empty in figure 2.2 and its vertices are now shown in the "Backward" path text box. The chain reaction started with the removal of vertex 22. Vertex 22 is inaccessible from vertex 6 because the canonical rule states that the snake must climb to the fourth dimension before climbing to the fifth. This would leave a snake that reaches vertex 18 with no place to go, thus forcing it to become the first vertex of the backward path. Look at figure 2.1 and imagine that vertex 22 is inaccessible. This leaves vertices 20 and 21 connected to each other and to the vertices 28 and 29 respectively. Since vertices 28 and 29 are also connected to each other, there would be no way for a snake to reach either vertex 20 or 21 and be able to proceed to other accessible vertices. Given that reaching 20 or 21 would necessitate the end of the snake path, these vertices can be pruned in the presence of a defined backward path. The move from 7 to 6 caused vertex 15 to be inaccessible which makes vertex 31 a dead end so it can be pruned. Vertex 25 is already a dead end so it can be pruned. This pruning step reveals that vertices 28 and 29 are now dead ends, thus the same reasoning applied to vertices 20 and 21 can be applied to them. Once they are pruned, 12 and 13 become dead ends as well. Vertex 15 was made inaccessible because of the move to 6, so all that remains is vertex 14. This is shown in Figure 2.2.

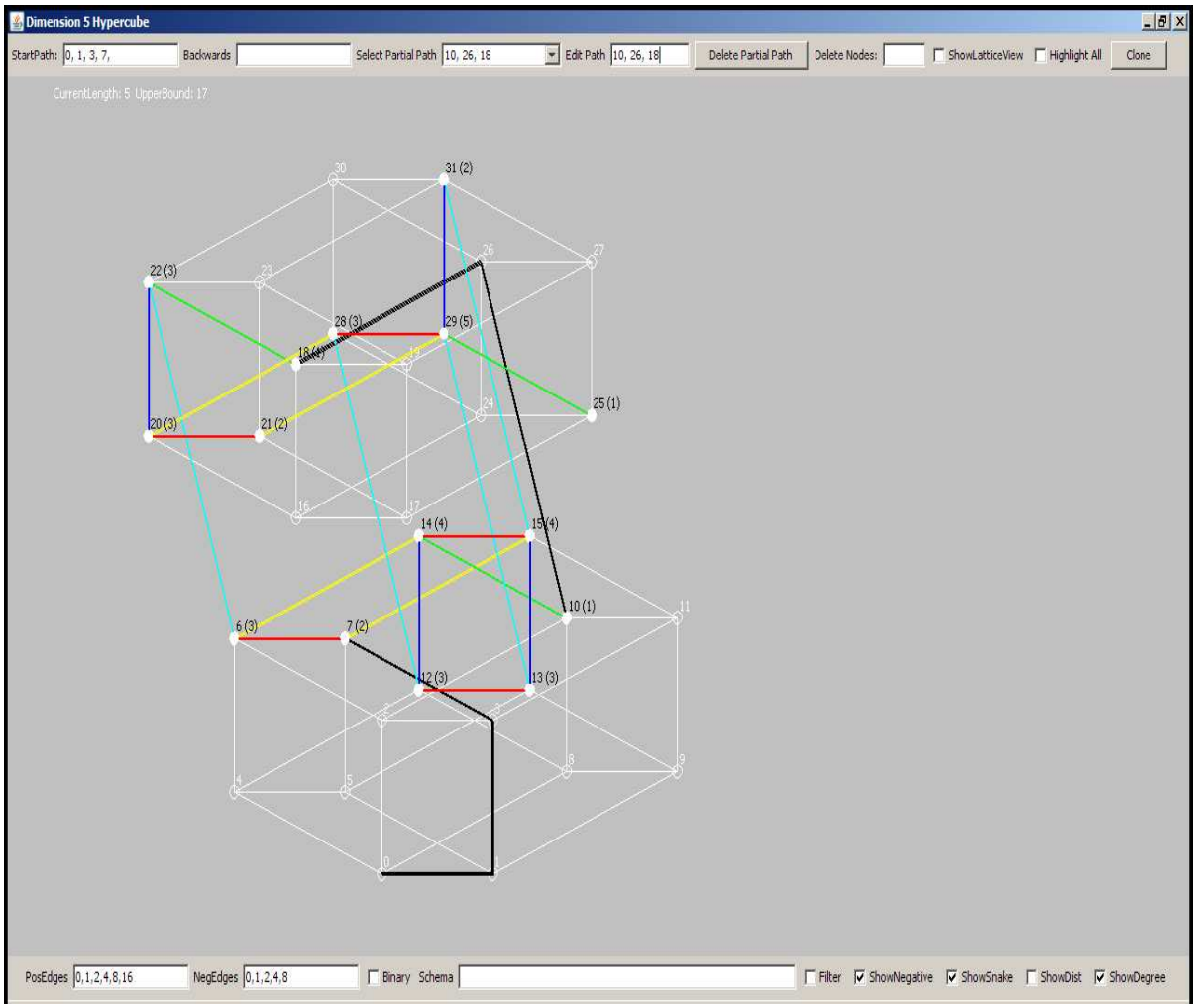

Figure 2.1: This figure visually represents an initial partial path configuration that could be supplied to the PPBBSM. The start path is (0, 1, 3, 7) and the only defined partial path is (18, 26, 10). The PPBBSM starts from vertex 7 and proceeds with a depth-first search throughout the hypercube. Partial paths are appended to the forward path when the depth-first search takes the forward path to a vertex adjacent to any of the partial path ends.

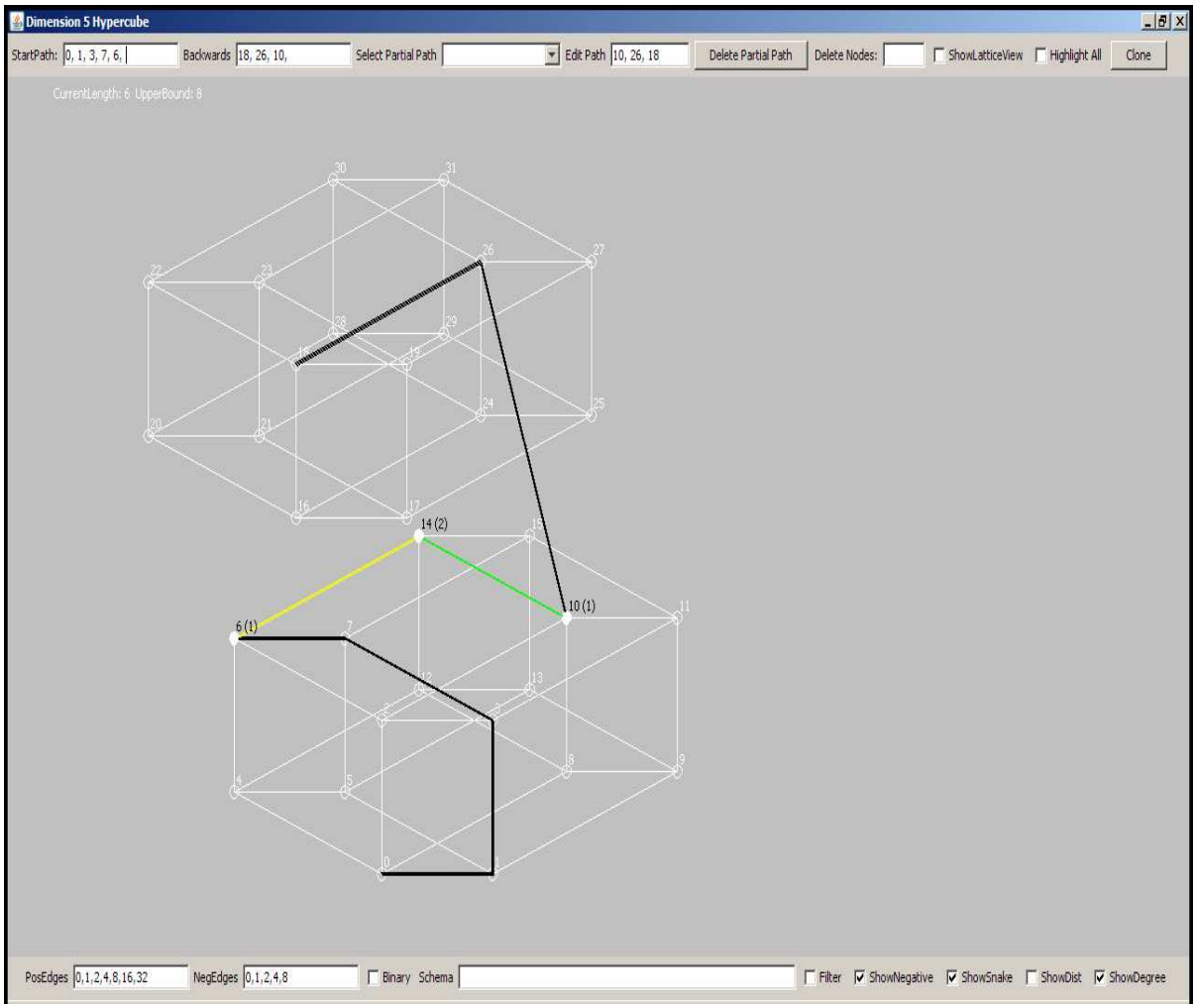

Figure 2.2: After the move from 7 to 6, the upper bound is again adjusted and the partial path has been converted to the backward path. Amazingly, all accessible nodes except for the one between the start and end path have been pruned. The PartialPathHyperCube wrapper is the object that models the properties of the hypercube and keeps track of all partial paths. It has told the IVE that the only accessible vertex not currently on a path is vertex 14. The PPBBSM is also informed by the PartialPathHyperCubeWrapper. The appendices more clearly explain how these framework components interact.

2.4 PARTIAL PATH BRANCH AND BOUND SEARCH

The partial path branch and bound search algorithm starts with an initially defined partial path snake as input, and outputs a snake that satisfies defined lower bound criteria if one can be found through all initial partial paths. All paths except the actual start path are optional. There is syntax for defining the initial partial path snake. For instance, an initial dimension 7 configuration could be defined as "0, 1, 3, 7;;86, 84, 68;b:75, 73, 72". This snake starts at 0,1,3,7. It has an internal path with vertices (86,84,68). The snake must end at vertex 75 and get there via (72,73). Multiple colon-separated internal (partial) paths may be specified. The "b:" denotes the presence of a backward path. The search algorithm starts from the end of the start path and continues in depth-first fashion. When the start path moves to a vertex adjacent to either end of an existing partial path, the partial path is appended to the forward path. When any of its constraints are violated, the algorithm backtracks to the most recent branch point and continues the depth-first search. There are three main types of constraints that make this search algorithm much more effective than uninformed depth-first search. The next three sub-sections detail the way the search algorithm continuously checks for violations against the constraints and how they are represented and updated after each move.

2.4.1 UPPER BOUND CONSTRAINT

The upper bound constraint and calculation can best be understood by example. Figure 2.3 shows a situation in Q_6 with an initial start path and 3 additional partial path elbows. The particular snake can be characterized by the string "0, 1, 3, 7;;12, 13, 29;62, 58, 50;37, 53, 49;". This means that it has a start path of "0,1,3,7" and internal partial path elbows of "12,13,29", "62,58,50", and "37,53,49". There is no initially defined backward path.

The upper-left corner indicates that the current snake length is 9 and no snake from this initial configuration can be longer than $L = 27$. It is important to note that the longest possible snake in Q_6 is 26. The method the framework uses to arrive at the upper bound

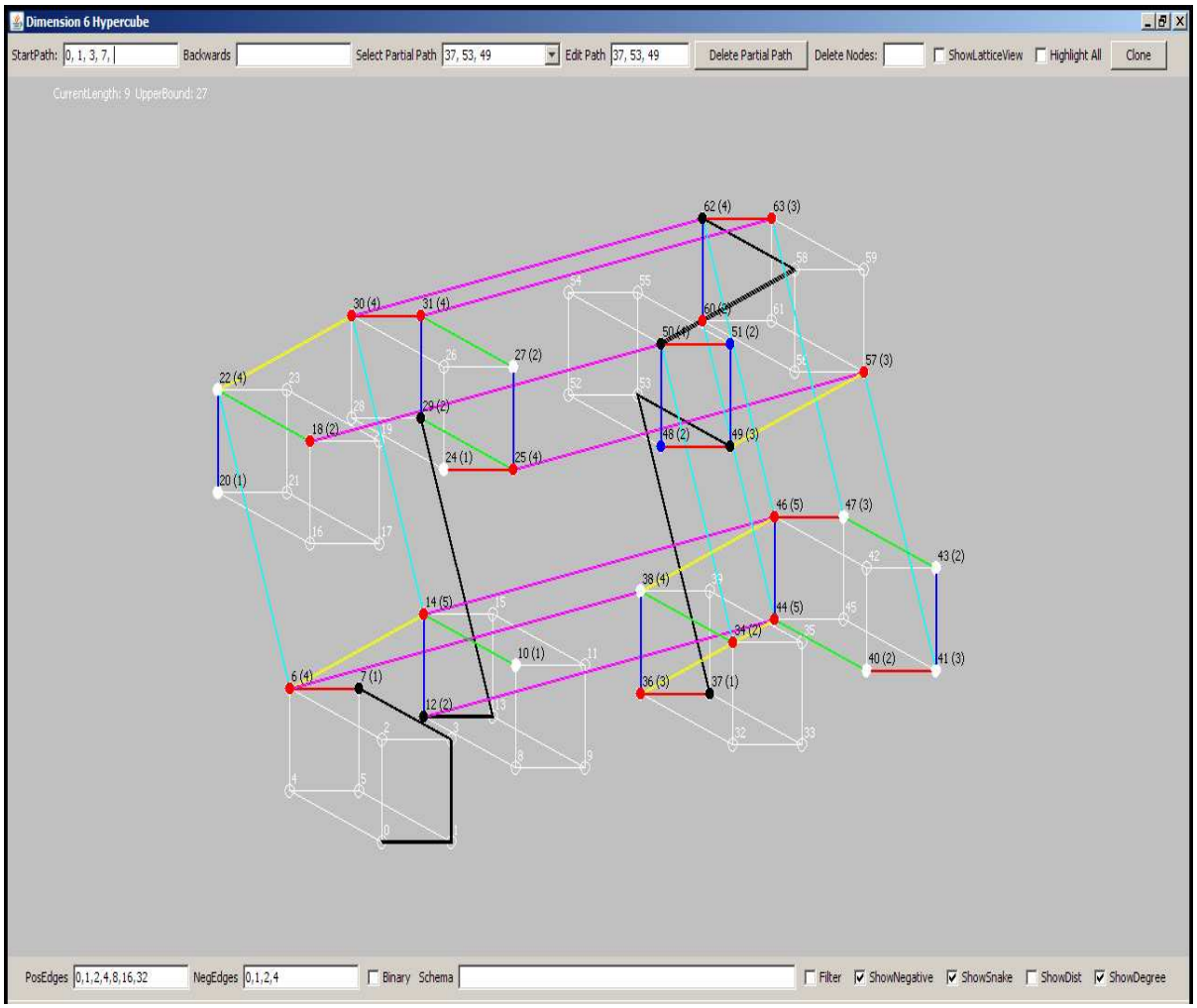

Figure 2.3: A set of paths (start path with three small partial paths) used to demonstrate the upper bound estimate employed by the PPBBSM. The vertices highlighted red are neighbors of various path ends. The vertices highlighted blue are neighbors also, but they are shared neighbors of two partial path ends (49 and 50). The integers in parentheses beside the integer vertex label represent the number of accessible neighbors from the given vertex. See the text for a discussion of the significance of shared neighbors.

estimate of 27 will soon be described, but first it is important to establish a vocabulary for what is shown in the image. In this graph of the hypercube, there are vertices and edges. Among the edges, there are colored edges (PosEdges), black edges, and white edges (NegEdges). The black edges are those that have been designated by the user to be included in the final snake. So those can be called the path edges. The colored edges are edges that are still accessible and could possibly be included in the final snake. The white edges are attached to inaccessible vertices and cannot be included in the final snake. To reduce confusion, not all negative edges are shown. The only white edges are those specified in the NegEdges text box (0,1,2,4).

Among the different types of vertices, there are inaccessible (skin) vertices that are "hollow" (outlined in white). For example, vertices 4 and 5 are inaccessible. Vertices 0 and 1 are also inaccessible, but these are referred to as path vertices because they are connected by a black path edge. There are two types of accessible vertices; those that are white-filled (10,20,22,24,27,38,47,40,41,43) and those that are colored. The colored vertices are mostly red (6,14,18,30,31,25,36,34,44,46,60,63), but there are two blue vertices (48,51). There are also black-filled vertices (7,12,28,37,49,50,62). These are the partial path ends that can be extended to further define the snake path. Incidentally, the 0 vertex is a partial path end but since the framework only deals with canonical snakes that start at 0, a constraint that all neighbors of 0 are inaccessible is imposed. More will be said about this as a pruning optimization in 2.4.3. In this particular diagram, the red and blue colored vertices are neighbors of the path ends. The red-filled vertices are neighbors from only a single path end, whereas the two blue vertices (48,51) are common neighbors to two different path ends (namely 50 and 49). With this vocabulary established, it is now simpler to understand the calculation for the upper bound estimate.

Phase one of the calculation starts from a completely unrealistic assumption that all the remaining accessible vertices can be included on the snake path. The only vertices that cannot possibly be included are the hollow, white outlined skin vertices. Once these are

removed from consideration, the vertices that remain are those that are already part of the snake path, and those that are colored (either white filled, red, or blue).

Let P denote the set of all vertices that are currently defined as snake path vertices. Let A denote the set of all remaining accessible vertices that are not on any of the defined paths. Also, let S represent the set of vertices that will compose the final snake.

$P = \{0, 1, 3, 7, 12, 13, 28, 37, 53, 49, 62, 58, 50\}$

$A = \{6, 14, 10, 18, 22, 20, 30, 31, 27, 25, 24, 48, 41, 57, 60, 63, 36, 38, 34, 46, 44, 47, 40, 41, 43\}$

Note that $|P| = 13$ and $|A| = 25$. In an ideal (but unrealistic) scenario, all of the vertices in A could be combined with the vertices in P to assemble one continuous open path. If this could happen, these 38 vertices would form a snake of length 37 (It is always the case the snake path length L is equal to the number of vertices that make up its path minus one). The upper bound calculation starts with this naive upper bound value. Phase two in the calculation is to come up with a conservative estimate for the number of vertices that cannot possibly be included in the final snake S. This is the point at which the red and blue highlighted vertices are considered.

Let P_e denote the set of partial path ends (black-filled vertices) except the 0 vertex. For some $p_e \epsilon P_e$, let $\Gamma(p_e)$ refer to the set consisting of the neighbors of p_e. Also, let G_e denote the set of all neighbors of vertices in P_e. Note that $G_e \subseteq A$. But when $|\Gamma(p_e)| > 1$ (for a single arbitrary $p_e \epsilon P_e$), it will obviously be the case that $|\Gamma(p_e)| - 1$ members of A could not possibly be included in S.

In this example,

$G_e = \{6, 14, 31, 25, 30, 18, 26, 44, 46, 57, 60, 63, 48, 51\}$

and the question that this step of the calculation asks (conservatively) is how many of these vertices of G_e will necessarily have to be left out of the set of vertices in the final snake S. Table 2.2 shows how these vertices of P_e and all their neighbors G_e are considered.

Phase three adjusts the raw total to take into account common neighbors of partial path ends. Notice in the table that the neighbors of path ends 49 and 50 have two vertices in

common (48 and 51). The proper, but conservative approach is to realize that it could be the case that the snake will choose to go from 49 to 57. It could also choose to go from 50 to (18 or 34) and if so, these common neighbors (48 and 51) would be double counted in this scheme. So to adjust for this, the intersecting neighbors are subtracted from the raw total. Doing so for this example reduces the total amount subtracted from $G_e \subseteq A$ by 2 (that is, subtract 8 instead of 10). Next take the naive original number of possible snake path vertices (38) and subtract 8 to get a new value of 30 possible vertices that can make up the snake path. This yields an upper bound on $L \leq 29$. So why does the visualization application report that the upper bound estimate is 27? How can it subtract an additional 2 vertices?

Phase four is the phase that holds the most promise for mathematical thinkers trying to solve this problem. The default implementation of the UpperBoundEstimate interface arrives at 27 by making the following analysis.

Let $M \subset [A \setminus G_e]$ be defined as the set of monovalent (vertices of degree = 1) among vertices of A that have not already been considered for subtraction. Thus ...

$M = \{10, 24, 20\}$

If any one of these vertices is included in S, the snake will terminate. The snake can only terminate once; otherwise it wouldn't be a single continuous path. So we can confidently conclude that two additional vertices can be taken out of consideration as candidates for inclusion in S. Thus $|S| \leq 28$ which yields a maximum possible snake length $L \leq 27$.

There is a subtle but quite useful feature of the framework when a backward path is present. If a partial path end has an accessible degree of one when a backward path is present, the partial path can safely be extended to its only accessible neighbor. Once this happens, the upper bound estimate is able to reduce the upper bound estimate by inspecting the properties around this new partial path end.

Table 2.2: This table shows how to reduce the initial naive estimate of the upper bound given in phase one of the upper bound calculation. This is phase two of the calculation for the situational upper bound. It calculates a raw score of what to subtract from the $|A|$ to reduce the upper bound. The actual amount subtracted from $|A|$ is reduced in a subsequent phase to reflect the intersection of path end neighbors.

$p_e \epsilon P_e$	$\Gamma(p_e)$	Subtract from $\|A\|$ (raw)
7	$\Gamma(7) = \{6\}$	$\|\Gamma(7)\| - 1 = 0$
12	$\Gamma(12) = \{14, 44\}$	$\|\Gamma(12)\| - 1 = 1$
29	$\Gamma(29) = \{25, 31\}$	$\|\Gamma(29)\| - 1 = 1$
37	$\Gamma(37) = \{36\}$	$\|\Gamma(37)\| - 1 = 0$
62	$\Gamma(62) = \{30, 63, 60, 46\}$	$\|\Gamma(62)\| - 1 = 3$
49	$\Gamma(49) = \{57, \mathbf{48}, \mathbf{51}\}$	$\|\Gamma(49)\| - 1 = 2$
50	$\Gamma(50) = \{18, 34, \mathbf{48}, \mathbf{51}\}$	$\|\Gamma(49)\| - 1 = 3$
	Raw Total to Subtract	10

2.4.2 CAN IT REMAIN CONNECTED?

The discussion around figure 2.2 demonstrated a remarkable chain reaction that led to the pruning of many initially accessible vertices. Occasionally, this chain reaction can prune all the neighbors of more than one partially defined path. If this situation occurs, the snake composed of the defined partial paths would have more than one dead end. This would create a situation that would make it impossible for all the defined paths to come together to form a single continuous snake path. After each move, the framework checks whether this illegal condition has taken place, and the search backtracks.

2.4.3 CAN IT STILL BE MAXIMAL AND CANONICAL?

It was previously stated that limiting the search space to truly canonical and maximal snakes yields a significant reduction of the overall search space. Rajan [8] states that the maximal snake cannot be extended from either its start or its end vertices. A canonical maximal snake is a maximal snake that starts (and cannot be extended) from the 0 vertex.

To gain an appreciation for the actual reduction, consider the number of Q_6 snakes found with an exhaustive, uninformed depth-first search. There are only 110381 maximal canonical Q_6 snakes compared to 246651 Q_6 snakes that could be extended from 0 after the search routine was finished. The idea for doing this came from Kochut's *coil-searching* optimization described in [6]. For coils, the snake must be able to return to the start vertex. Thus, he abandons a search when there are no more neighbors accessible from the start. For snakes however, the rules of the problem prohibit the path from returning to the start. Thus, the PPBBSM backtracks when it is impossible to prevent a snake from extending from the start.

The idea is pretty simple and is also applied to the backward snake path thus increasing the opportunity to violate constraints. For any given hypercube Q_d, the 0 vertex has d neighbors. Each of these d neighbors have $d-1$ neighbors other than 0. For example, in Q_4 the neighbors of vertex 4 (other than vertex 0) are (5,6,12). In Q_4, the snake defined as (0,1,3,7,15,14,10) makes (5,6,12) inaccessible. Since all neighbors of vertex 4 are inaccessible to the snake, the snake cannot move in such a way as to make vertex 4 inaccessible. Thus, the snake will always be extendable from the start (vertex 0) to vertex 4. In higher dimensions, snakes can create this situation very early in their construction. In such a situation, the backtracking of the branch and bound module prunes a significant number of possible snakes.

Chapter 3

BENEFIT OF THE PARTIAL PATH APPROACH

This chapter shows the results of an experiment designed to demonstrate how run times of the PPBBSM differ given various types of initial configurations. The hypothesis is that an initial configuration with several small initial partial paths compares favorably in algorithm run time to an initial configuration with only the start path specified. The first figure shows the final optimal Q_7 snake found by the PPBBSM from the various initial configurations given in later figures. The figures are organized into groups. The first group (figures 3.2, 3.3, and 3.4) demonstrates the runtimes for initial configurations with only 13 of the 51 longest path vertices. The second group of figures (3.5 and 3.6) shows how the PPBBSM is affected by two different initial configurations containing 16 vertices. The final group (figures 3.7 and 3.8) shows an example of how the algorithm runs for an initial configuration in Q_8. It is important to note that all experiments were run on a laptop with 1.5 GHz processor and 512MB of RAM.

These groups of figures validate the utility of the partial path approach. Ideally, there would exist an approach that facilitates exhaustively searching higher dimension hypercubes (Q_7 and above) in a reasonable amount of computing time. However, the computational complexity in these dimensions is prohibitive. Heuristic approaches have given us current lower bound snakes in Q_8 and above, but it seems reasonable to expect that heuristic approaches would give better results if they searched over reduced representations of the search space. These figures suggest that redefining the problem as the search for a small number of compatibly placed initial partial paths is a promising way to combine the benefits of exhaustive

search with heuristic search. Furthermore, the UpperBoundEstimate interface offers a carefully designed extension point allowing for sophisticated pruning of the exhaustive search portion. The current implementation offered by the framework has produced the interesting run time result found in the comparison of figure 3.4 to figure 3.5. In figure 3.4, the run time of the PPBBSM is faster for only 13 total initially specified vertices than the initial configuration found in figure 3.5 with 16 vertices. This happened because the 13 vertices were distributed as small but well-spaced partial paths whereas figure 3.5 has all the vertices on the start path. We have proposed that the PPBBSM could be used as a tool to evaluate heuristic approaches that look for highly fit partial path placements. In the next chapter, we report the experimental results that put this idea to the test.

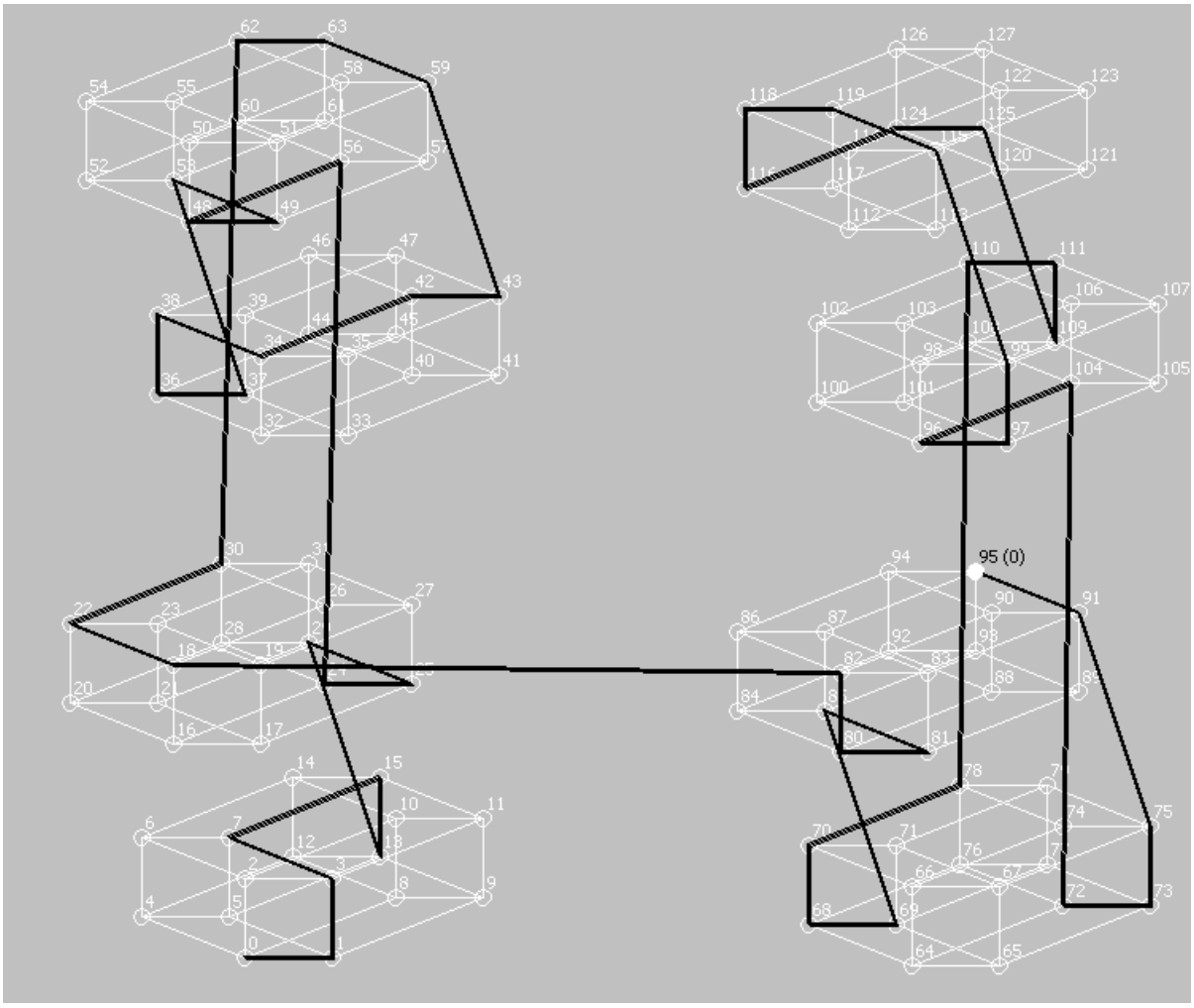

Figure 3.1: One of the 12 optimal canonical Q_7 snakes. Evidence from the subsequent figures suggest that quickest runs of the PPBBSM seem to be most highly correlated with initial configurations of several short but well spaced partial paths.

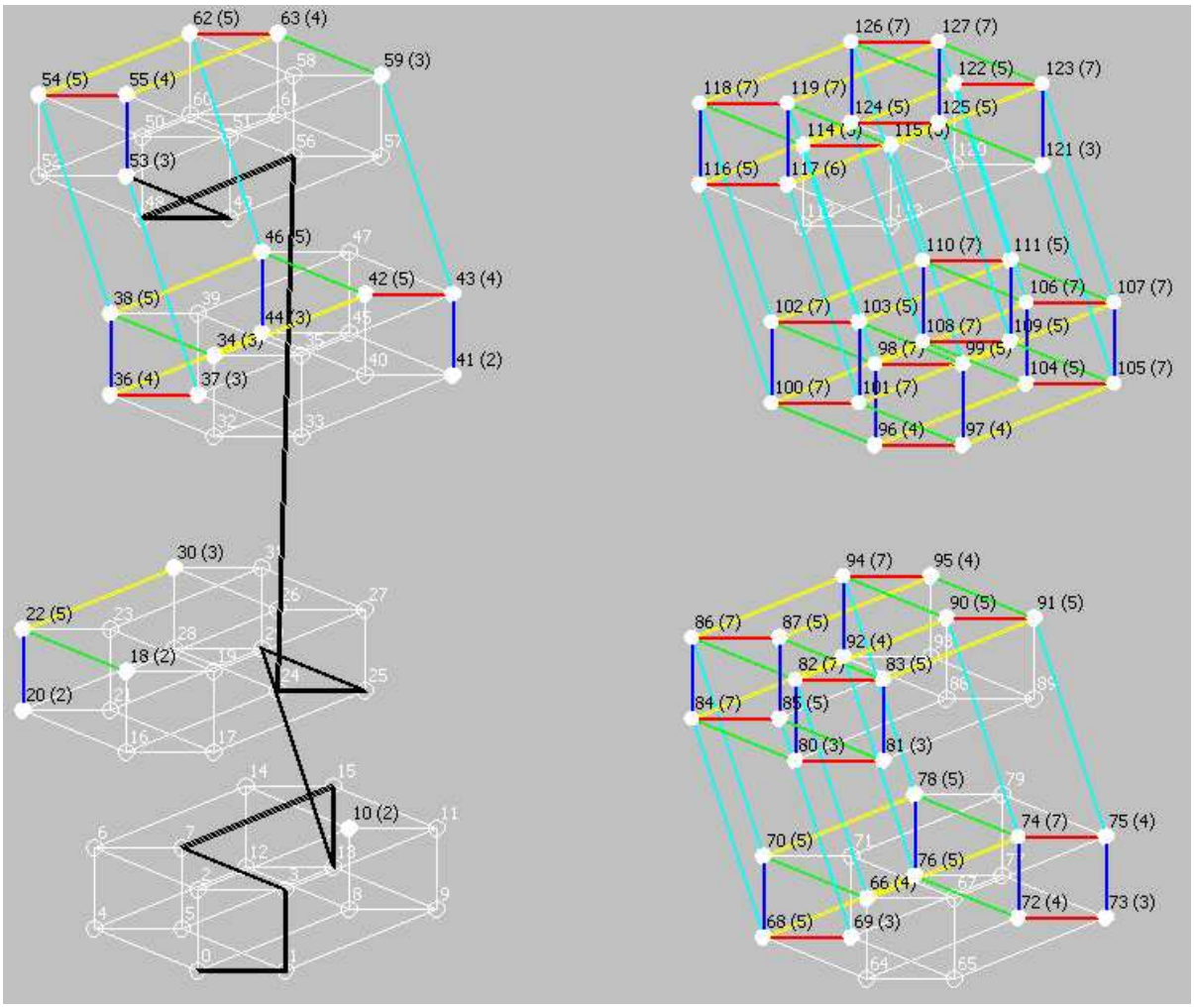

Figure 3.2: It took 74 minutes to find the snake in figure 3.1 from this initial configuration. Only 13 vertices are defined in the initial configuration and all are on the start path. This is essentially an uninformed depth-first search from a length 12 start path. The PPBBSM takes a long time because the upper bound estimate has limited information and thus doesn't backtrack as quickly as an initial configuration with multiple partial paths. The next two figures illustrate this.

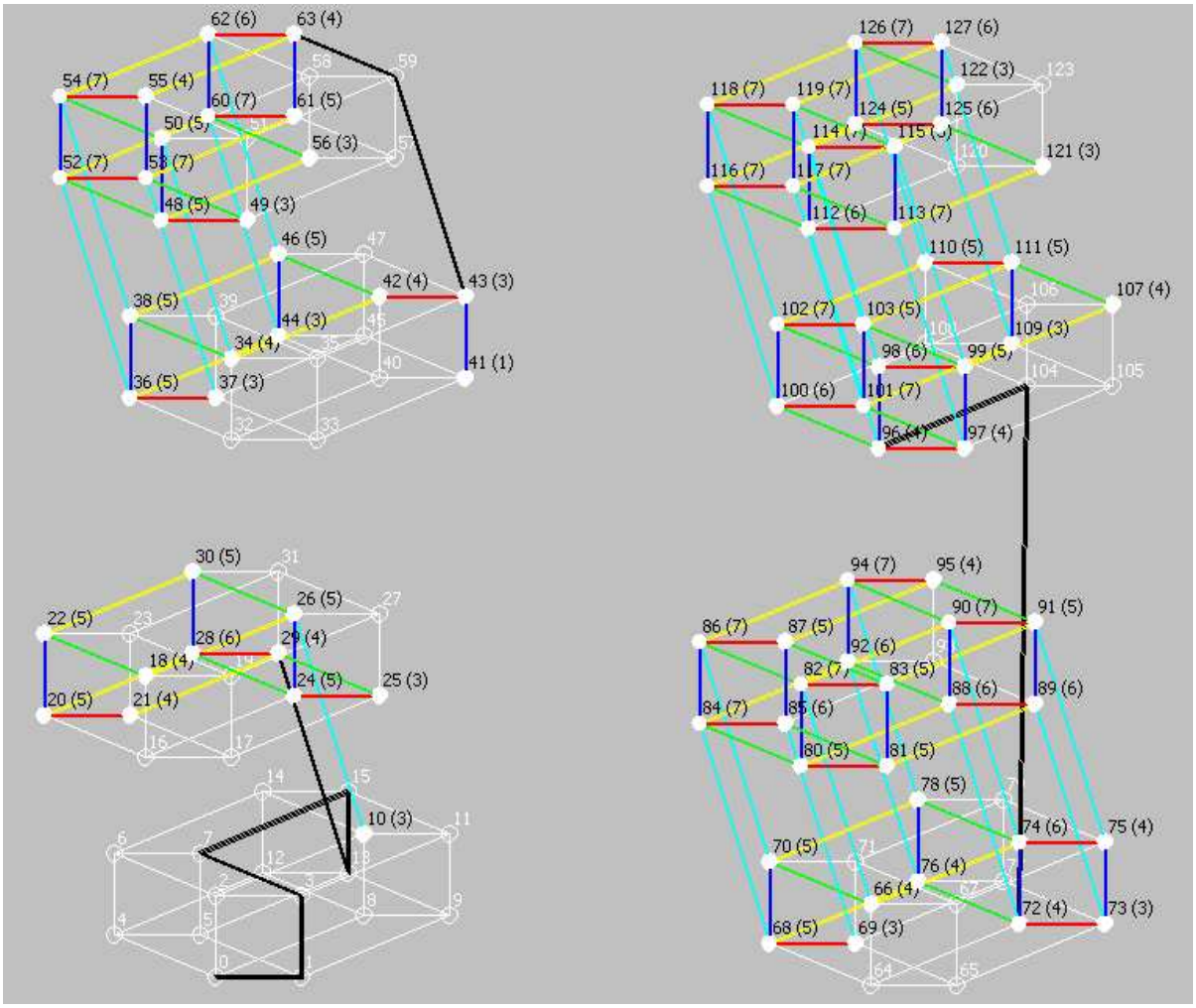

Figure 3.3: It took approximately 19.37 minutes to find the snake in figure 3.1 from this initial configuration of 13 defined vertices. Notice that the length of the start path was shortened in order to make enough vertices available to define the two partial paths.

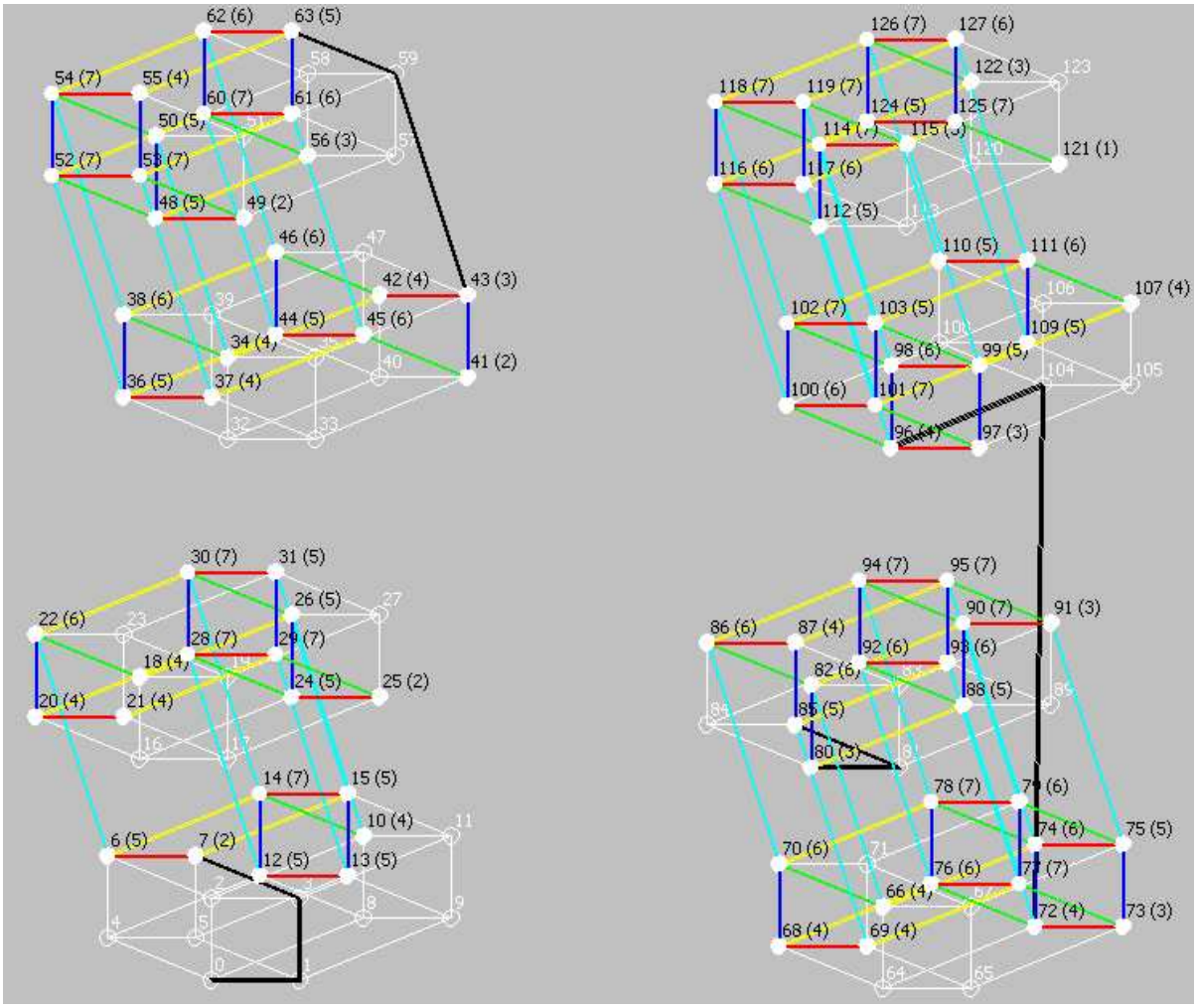

Figure 3.4: The only difference between this figure and figure 3.3 is that more of the start path was trimmed in favor of creating an additional internal partial path. However, there is a significant difference in PPBBSM run time for this configuration. It only took 153 seconds (2.55 minutes) to find the snake in figure 3.1 from this initial configuration of 13 initially defined vertices.

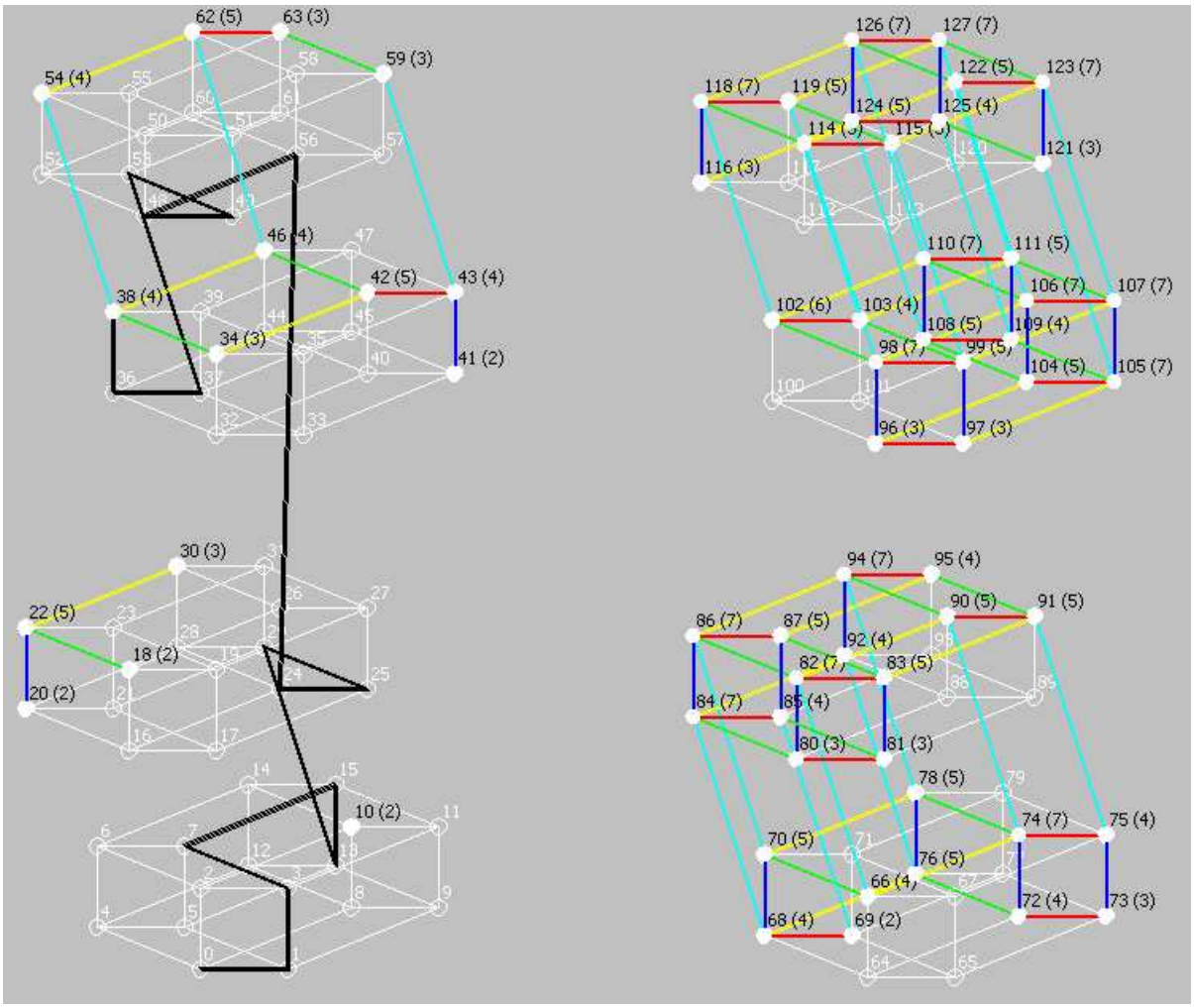

Figure 3.5: These next two figures show two initial configurations of 16 vertices. This figure shows how the PPBBSM run time improves after adding 3 more vertices to the start path of the poorly performing initial configuration in figure 3.2. It took 200 seconds to find the longest snake from this configuration. But note that an initial configuration of 16 vertices on the start path is still outperformed by the multiple partial path 13 vertex initial configuration found in the previous figure 3.4.

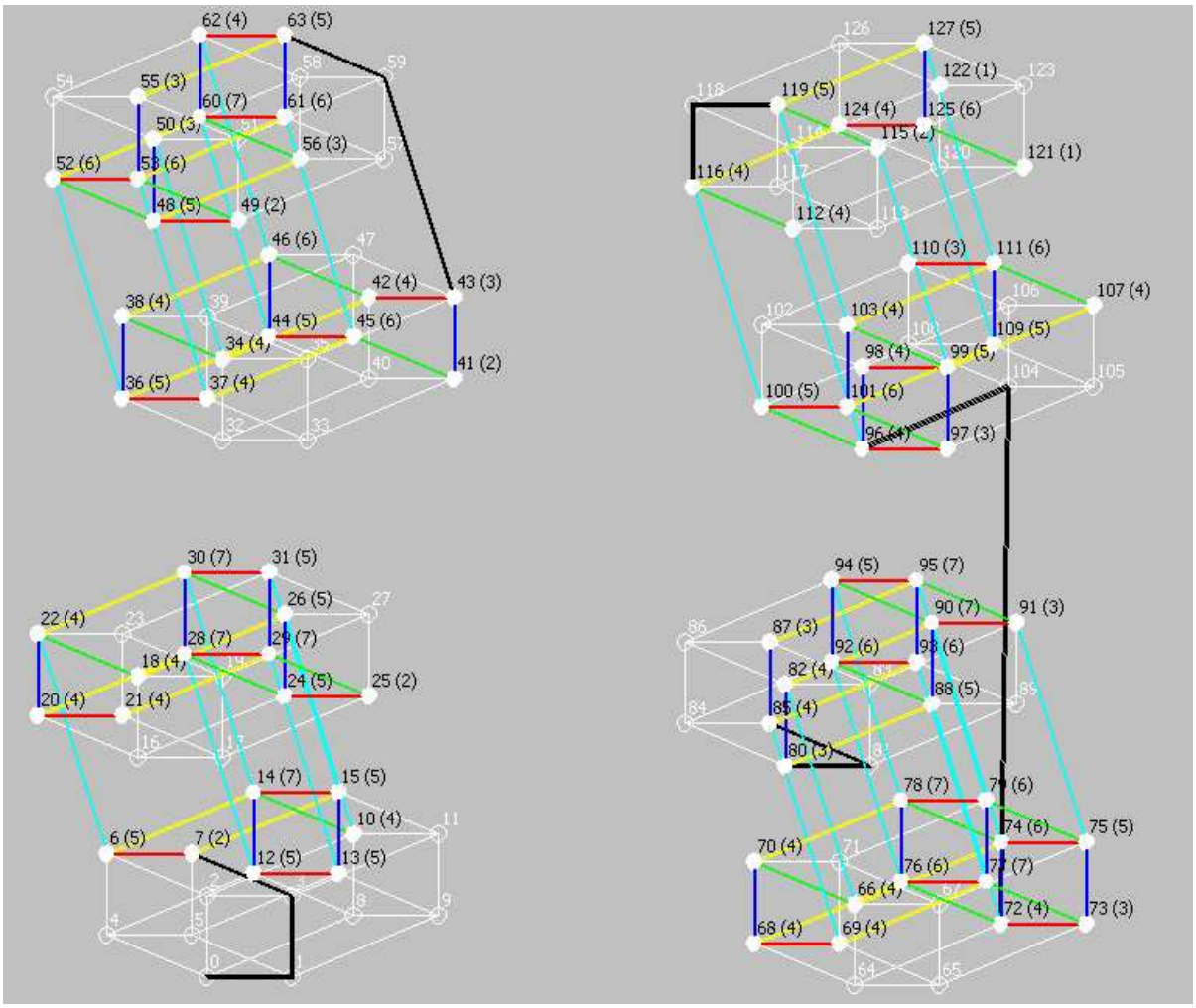

Figure 3.6: This figure shows how the PPBBSM differs from the previous in that many vertices were taken off from the start path in favor of defining 4 initial internal partial paths. It only took 6.81 seconds to find the longest snake from this configuration of 16 initially defined vertices.

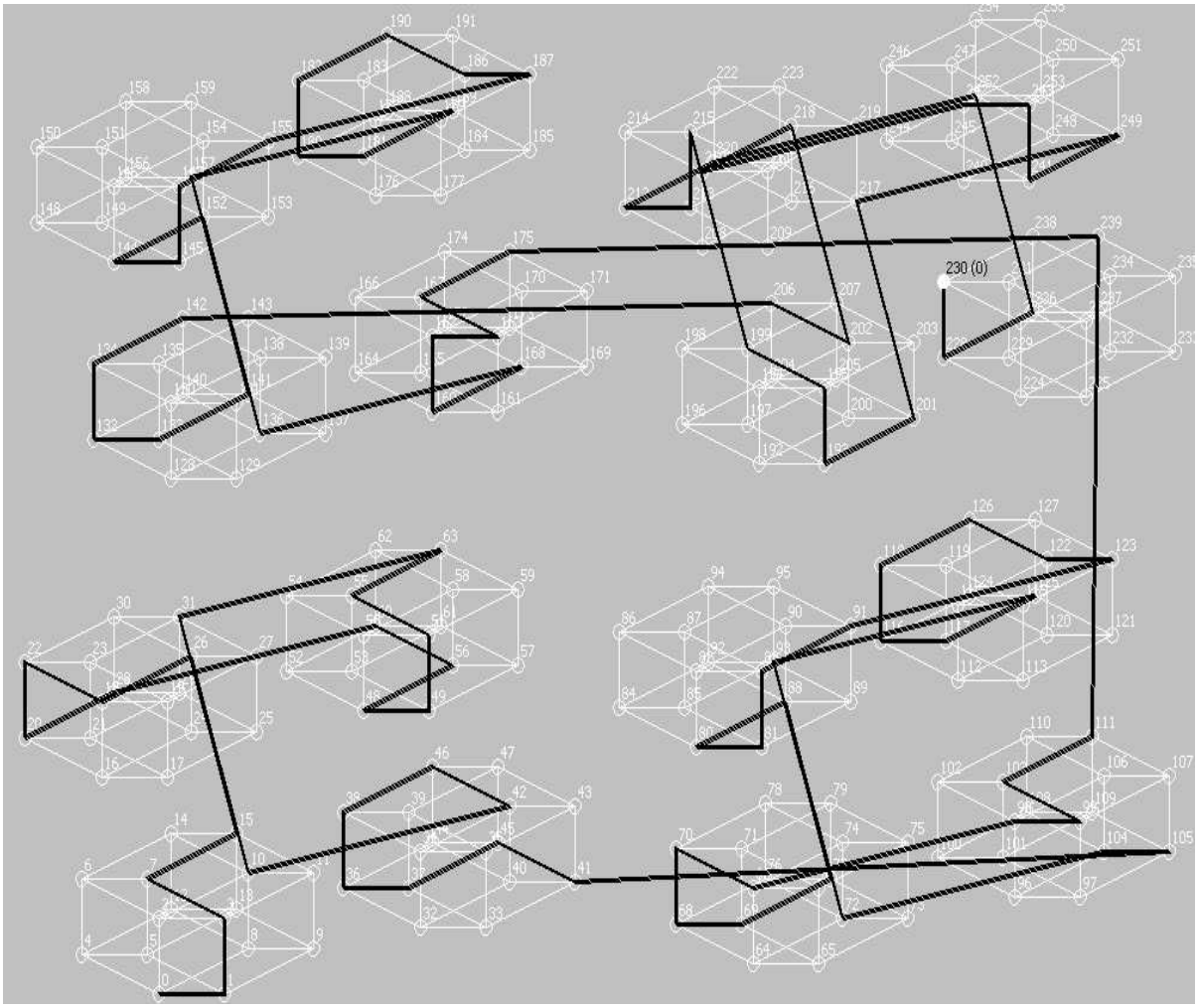

Figure 3.7: Q_8 currently has a lower bound of 97. This is an example of one of the length 97 snakes.

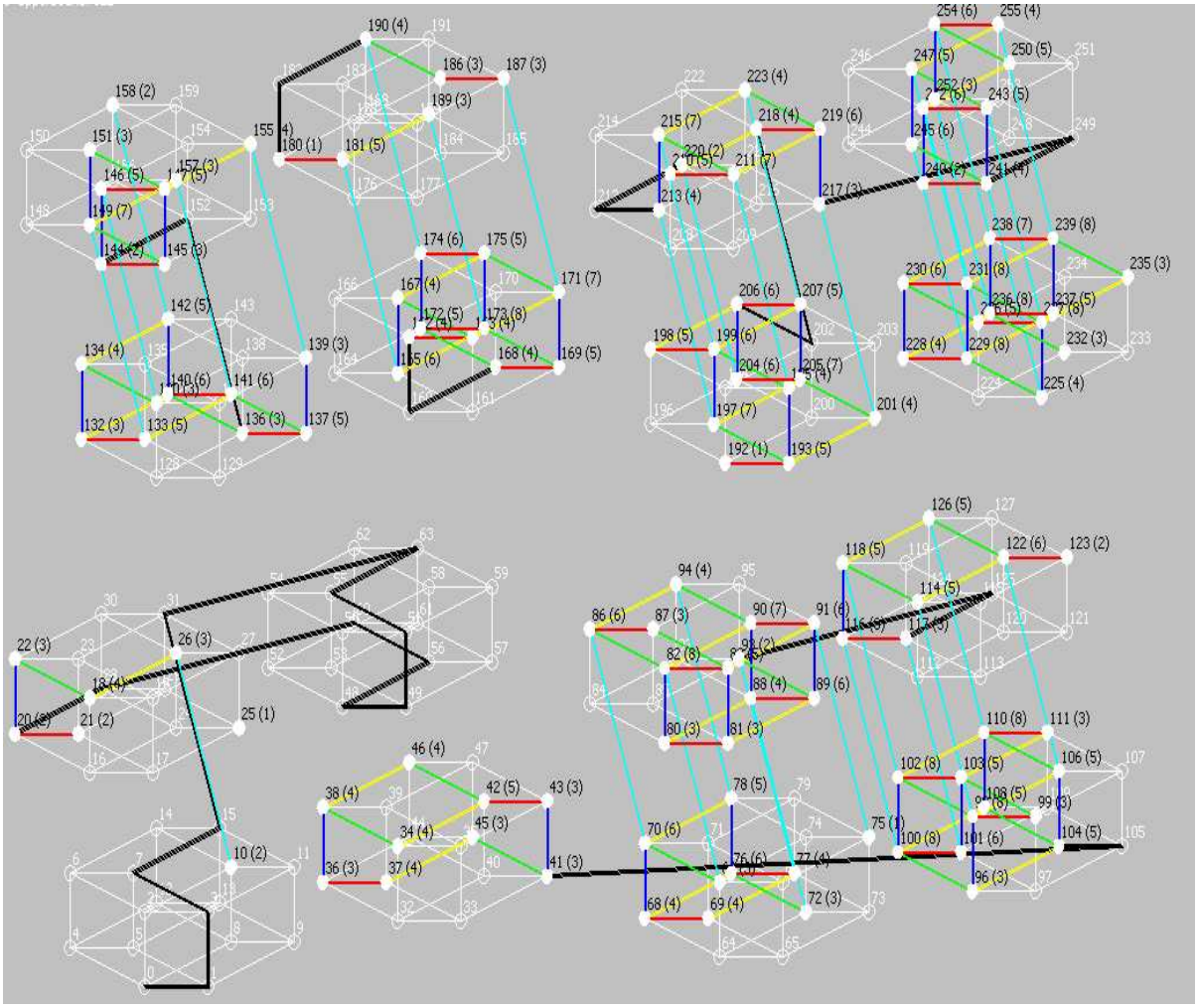

Figure 3.8: It took about 2.88 minutes to find the snake in figure 3.7 from this initial configuration of 39 vertices and 8 partial paths. This example shows that it should be possible to use the PPBBSM as an evaluation function during an evolutionary search for well positioned initial partial path configurations.

Chapter 4

A CASE STUDY

The main value of the framework is that it offers an environment that inspires the design of new heuristic approaches for the snake-in-the-box problem. It does this by providing a wealth of tools to build an intuition for multi-dimensional hypercube space. This chapter details a specific interaction with the framework that has led to the development of two new heuristics. The k-cube heuristic allows one to use intuition gained from the IVE to define a high level plan to which the snake must adhere. We also developed an iterative improvement approach that starts from random maximal snakes and randomly breaks them into partial path fragments. These fragments are handed over to the PPBBSM to find better snakes. This approach quickly found one of the optimal Q_7 length 50 snakes. It also managed to improve a length 87 snake in Q_8 to a length of 89. The original length 87 snake was found by the k-cube heuristic technique. After discussing these experiments, we use insight gained from our analysis of the results and detail how the Genetic Algorithm (GA) and Simulated Annealing (SA) can offer improvement.

4.1 GAINING INTUITION

Q_6 is a dimension in which the enumeration of all canonical snakes is quite practical. Consequently, it is possible to learn properties that distinguish between optimal and sub optimal path moves. The source code for all the tools of the framework are in a source code repository. Included in the repository is a script that takes as input a partially defined snake and returns the length of the longest snake in Q_6 through the partial snake's vertices. The script works by doing a regular expression search through a file containing the exhaustive

enumeration of all maximal and canonical Q_6 snake node sequences. It uses regular expressions to abstract the nodes between the specified paths. If the script is called with multiple arguments (corresponding to multiple partial paths), it tries all the different orders of those paths and looks for the reversed versions of the paths as well.

This script is flexible in what it allows as input. For instance, a user can see the longest possible length of a snake that contains "0,1,3,7,6" as path vertices by supplying just that string as a command line argument. The script is written in a java scripting language known as groovy [7]. If the "longestsnake.groovy" script is called with the command argument "0,1,3,7,6", the script reports that $L = 25$ is the length of the longest snake containing those connected vertices. This is interesting because there is only one longest canonical snake in Q_6, and it has length $L = 26$. The fact that there is a unique $L = 26$ snake in Q_6 is very helpful when trying to derive properties that characterize mistake moves. The script has told us that the move from 7 to 6 is a mistake because no $L = 26$ path can be built with the rest of the accessible vertices. Armed with this information, the user can bring up the IVE to visually inspect properties of the hypercube that differ between the two start paths "0,1,3,7,6" and "0,1,3,7,15". An ongoing effort should be made to quantify these properties that only currently provide an abstract intuition, but this case study will show that intuition alone has led to the design of a heuristic yielding respectable results. The user can supply multiple arguments corresponding to different partial paths a final snake must include. For example, when the script is called with "0,1,3,7,6" and "43,41,40" as the arguments, the return value is $L = 25$. When called with "0,1,3,7,6" and "48,50,54", the longest possible snake in Q_6 containing those two connected sets of vertices is $L = 24$. Thus, the user can also gain an intuition for properties that make good partial path configurations by inspecting both of these in the IVE.

Looking at mistakes in Q_6 helps a researcher understand important differences in the properties that distinguish optimal from sub optimal snakes. Over time, one can see general tendencies that successful snakes share in various dimensions. Successful snakes seem to move

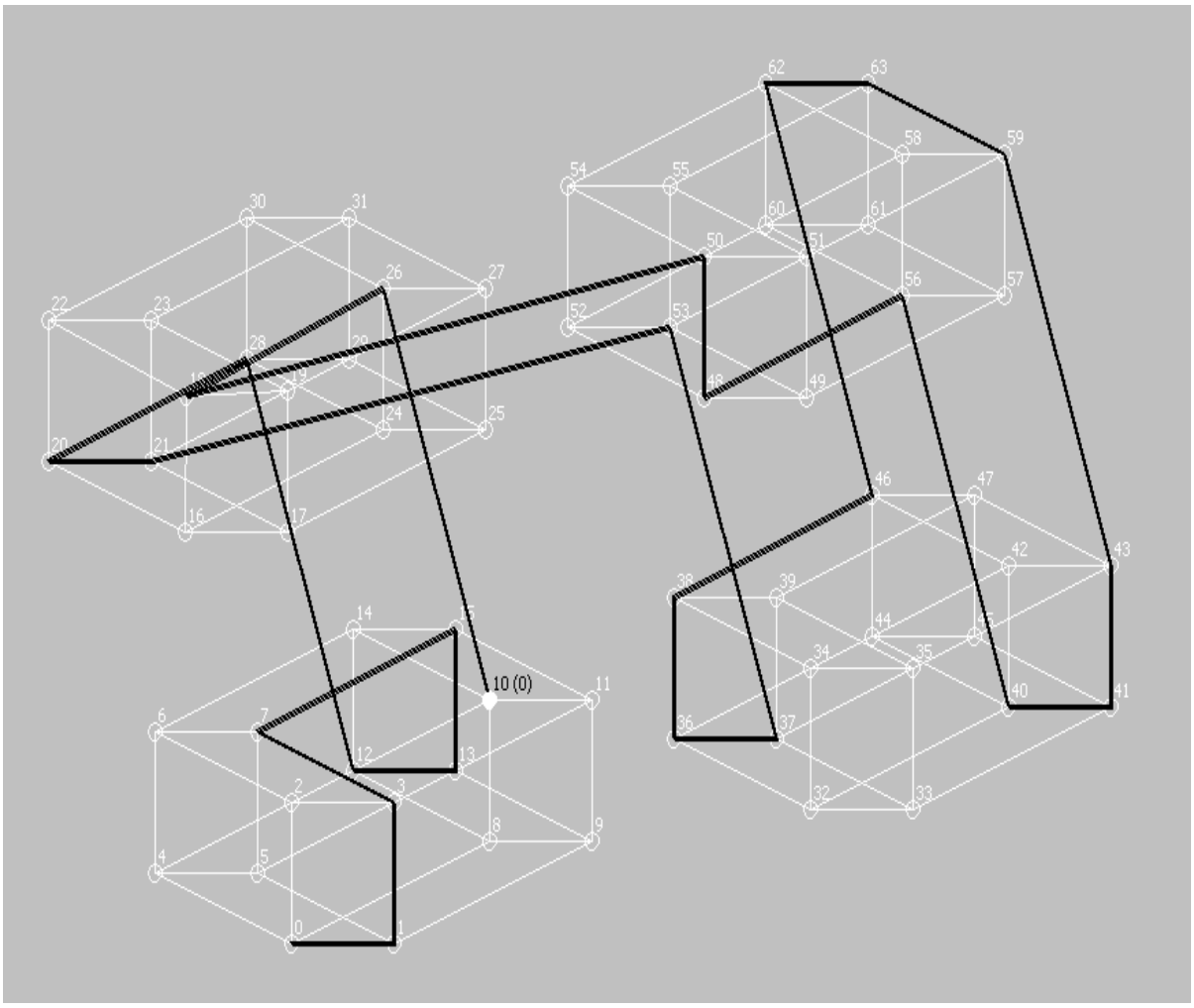

Figure 4.1: This is the unique longest 26 length snake in Q_6. Its vertices are (0, 1, 3, 7, 15, 13, 12, 28, 20, 21, 53, 37, 36, 38, 46, 62, 63, 59, 43, 41, 40, 56, 48, 50, 18, 26, 10). Notice that its final vertex is 10 which is on the same 4-cube (vertices 0-15) as its starting vertex 0. The first time the snake crosses over dimension 6 is when it moves from vertex 21 to vertex 53. The first 10 vertices of this snake are (0,1,3,7,15,13,12,28,20,21). This is actually a maximal (but very sub optimal) length 9 snake in Q_5. The length of the longest snake in Q_5 is 13, so this initial path that is sub optimal in Q_5 is preparing for the snake's later return across the sixth dimension when it leaves vertex 50 to get to vertex 18. At that point, the snake proceeds to vertex 26 and finally to vertex 10.

purposefully through the hypercube. They tend to strategically leave behind vertices that can be used later. Figure 4.1 shows the optimal snake in Q_6. The snake leaves the 4-cube identified by vertices (16-31) after 9 total initial transitions (0,1,3,7,15,13,12,28,20,21). But when it does so, it leaves behind vertices (18,26) that are eventually used towards the end of the snake. When watching the moves of this snake step by step in the IVE, one can almost glean a rationale for why the snake avoids certain vertices. If this can be formalized with the help of intuition from the IVE, it will significantly help with the effort to filter or rank possible arbitrary candidate moves.

4.2 THE K-CUBE HEURISTIC

The k-cube heuristic was designed to put intuition to the test. It allows a researcher to define the high level shape of a path by restricting the initial transitions to conform to a proposed plan. The shape is defined by specifying an ordered list of sub cubes (or k-cubes) the snake must initially traverse. The k-cubes are uniquely defined by the schema syntax discussed in [10]. An example plan in Q_5 will show how this works. An example schema list of (00***,*11**,10***) defines a comma separated plan that represents a chain of three initial 3-cubes the snake must traverse in the order they appear in the list. Figures 4.2 - 4.4 visually show the 3-cubes that constrain the snake path. Imagine a snake that starts with the vertices (0,1,3,7,6). Figure 4.2 shows that all of these vertices are contained completely by the first 3-cube of the plan. At vertex 6, the canonical snake must proceed to vertex 14. Upon arrival at vertex 14, the snake has reached the second 3-cube in the plan. The k-cube heuristic works by allowing the snake to make any transition within the current 3-cube, but if the snake leaves the current 3-cube, it must move to a vertex contained by the next 3-cube in the plan. In this example, the snake cannot move to vertex 10 from vertex 14 because that would constitute leaving its current 3-cube. If it leaves the current 3-cube, it must jump to the third 3-cube in the plan. The vertices of the third 3-cube are shown in figure 4.4. The

snake cannot actually get to that 3-cube from vertex 14, so it must move within the second 3-cube.

We use the term k-cube heuristic instead of 3-cube because the k-cube heuristic is implemented in such a way that the user can define a chain of any type of sub cubes. It could be a chain a 2-cubes or 4-cubes in Q_5 for example. The ForwardNeighborSelector Interface extension point made it incredibly simple to implement this heuristic. The PPBBSM gets a list of neighbors to visit from the implementation of the ForwardNeighborSelector it is configured to use. The KCubeHeuristic implementation effectively limits the branching factor in higher dimensions in a significant way.

We experimented with this heuristic by defining a plan for Q_8 consistent with the Q_6 motif discussed in Section 4.1. The following schema list (00000***, 000*11**, 0001*0**, 00110***, 0*1111**, 0101*1**, 01*100**, 01*000**, 010*01**, 0*1101**, 0010*1**) establishes a plan for only eleven initial 3-cubes in a Q_8 search. Figure 4.5 highlights the 3-cubes in the plan. It is difficult to see the 3-cube order in this single figure, but one can see the general idea behind the plan. It strives to create a snake like the Q_6 optimal snake by simply allowing patches the snake could return to in vertices (0-127). Vertices (0-127) are the first Q_7 half of the Q_8 hypercube. This is similar to how a plan for the Q_6 optimal snake would have left vertices for the snake to return to within its first Q_5 half (vertices 0-31) . In just over 2 hours and with an initial start path seed of (0,1,3,7,15,14,12,28,29), the PPBBSM returned a length 87 snake. It is quite interesting that this length 87 snake did in fact return to the (0-127) 7-cube in an analogous way that the optimal Q_6 snake returns to its (0-31) 5-cube. This result is interesting and respectable, but the snake generated from the plan does not quite reach the current lower bound of 97. This could be because either the plan is sub optimal or the initial seed of (0,1,3,7,15,14,12,28,29) has very costly mistakes. It is also the case that the algorithm's run time was limited to 24 hours. Perhaps letting the PPBBSM run longer could have found a longer snake, but we are more interested in using the framework to gain insight into smarter heuristics that produce longer snakes in short run times. With

that in mind, we decided to design a heuristic that can improve upon good quality snakes while simultaneously exerting tight control over the branching factor.

4.3 THE ITERATIVE IMPROVEMENT HEURISTIC

Iterative improvement refers to a general optimization technique that tries to improve a candidate solution by performing a local search for better solutions. Dorn's paper [2] provides an excellent comparison of iterative improvement techniques applied to a scheduling problem. One iterative improvement technique is randomized modification and search. The PPBBSM lends itself perfectly to this approach. Our k-cube heuristic produced a length 87 snake that was thought to have room to grow. It has been shown in chapter 3 that the PPBBSM can quickly find optimal snakes through specified partial paths if enough partial paths are supplied in the initial configuration. With that in mind, we designed an iterative improvement algorithm that wraps the PPBBSM by chopping a snake into random partial path fragments and submitting the fragments to the PPBBSM. The idea behind this method is that a snake can make mistakes that eliminate vertices that are necessary to build longer paths. Section 4.1 discusses tools the framework offers that demonstrate this phenomenon. By cutting the snake into random fragments, the transition responsible for such a mistake could be removed. The PPBBSM takes the fragments and tries to build a longer snake with the freedom it gets from the gaps that have been randomly selected. The following subsection details how our iterative improvement approach produced a length 50 optimal Q_7 snake.

4.3.1 ITERATIVE IMPROVEMENT EXPERIMENTAL PROCEEDURE

First we used the RandomSubsetSelector (see appendix C.1) implementation of the ForwardNeighborSelector interface to produce a random Q_7 snake with length 35. The path the PPBBSM found was (0, 1, 3, 7, 15, 14, 12, 28, 24, 26, 27, 59, 63, 55, 54, 52, 116, 84, 86, 70, 66, 98, 96, 104, 108, 110, 111, 103, 101, 69, 77, 73, 89, 81, 83, 115). We then start the iterative procedure to grow the snake to length 50. The procedure works by breaking this snake into

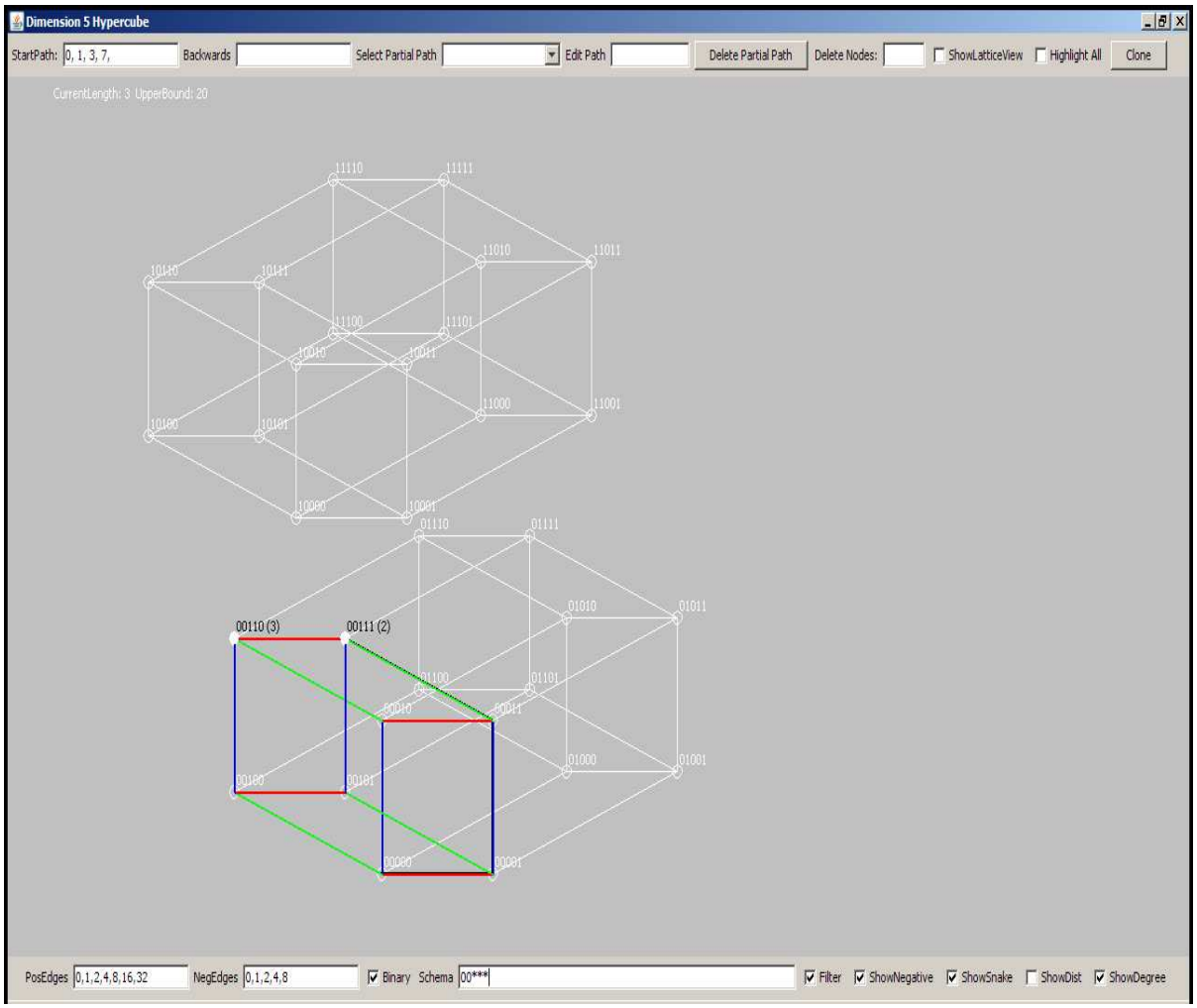

Figure 4.2: This image shows the first 3-cube in the plan defined by (00***,*11**,10***). The decimal vertex labels for the vertices in this 3-cube are (0,1,2,3,4,5,6,7). The canonical snake's first 3 transitions are always contained by this initial 3-cube. The k-cube heuristic allows the snake to make as many moves as possible within the current k-cube, but if the snake jumps off of the current k-cube, it must jump to the next k-cube in the plan.

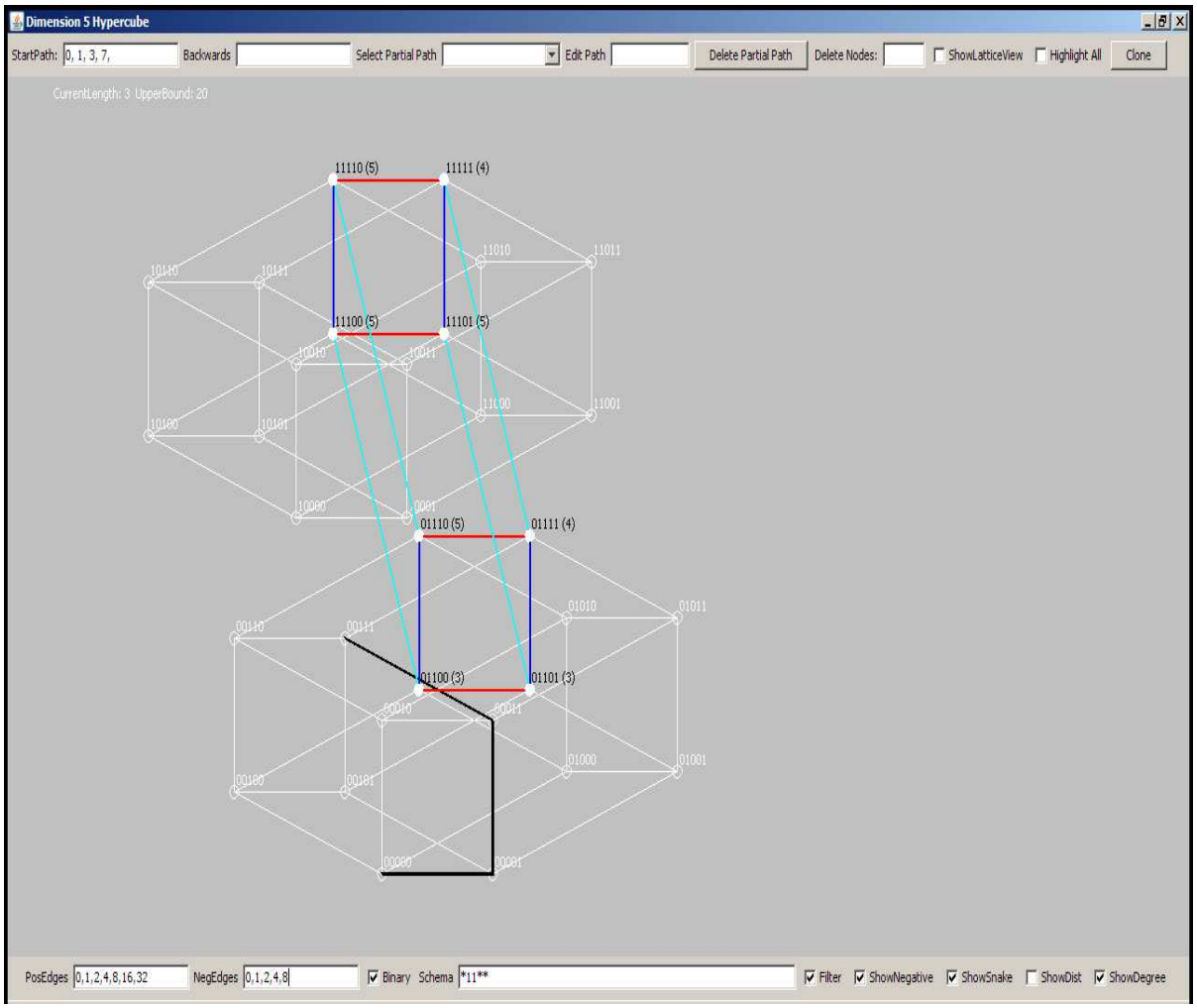

Figure 4.3: This 3-cube is represented by the schema *11**. Its decimal vertex labels are (13, 14, 15, 12, 29, 28, 30, 31). This is the second schema in the plan designated by (00***,*11**,10***). When the canonical snake is at vertex 7, it has a choice to go to vertex 6 or vertex 15. The k-cube heuristic would allow the move to either 6 or 15. If the snake goes to 6 and then 14, it would be in the second k-cube of the plan.

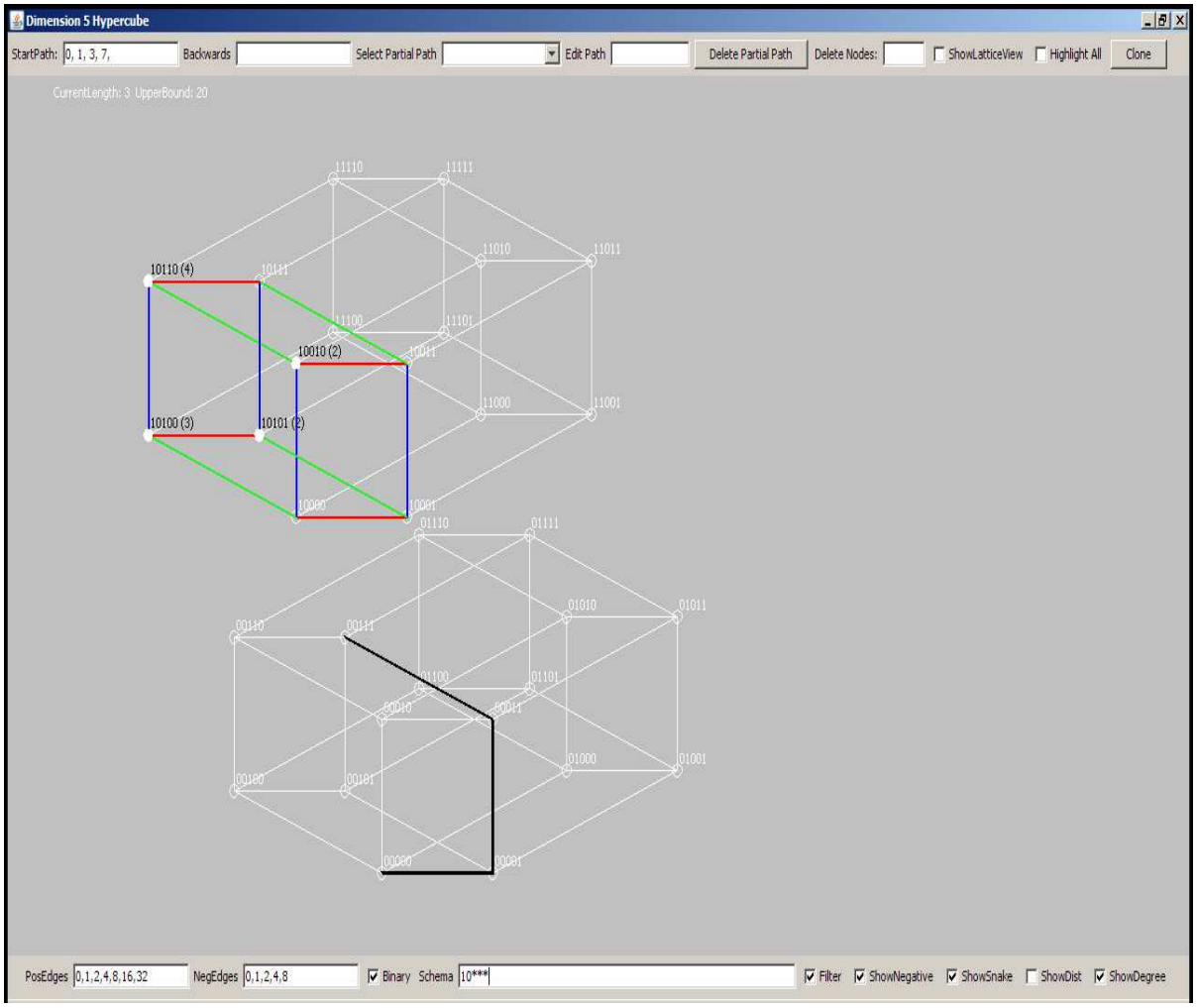

Figure 4.4: This is the final 3-cube in the plan defined by (00***,*11**,10***). The decimal vertex labels of this k-cube are (20,21,22,23,16,17,18,19). Once the snake gets here, it is no longer restricted.

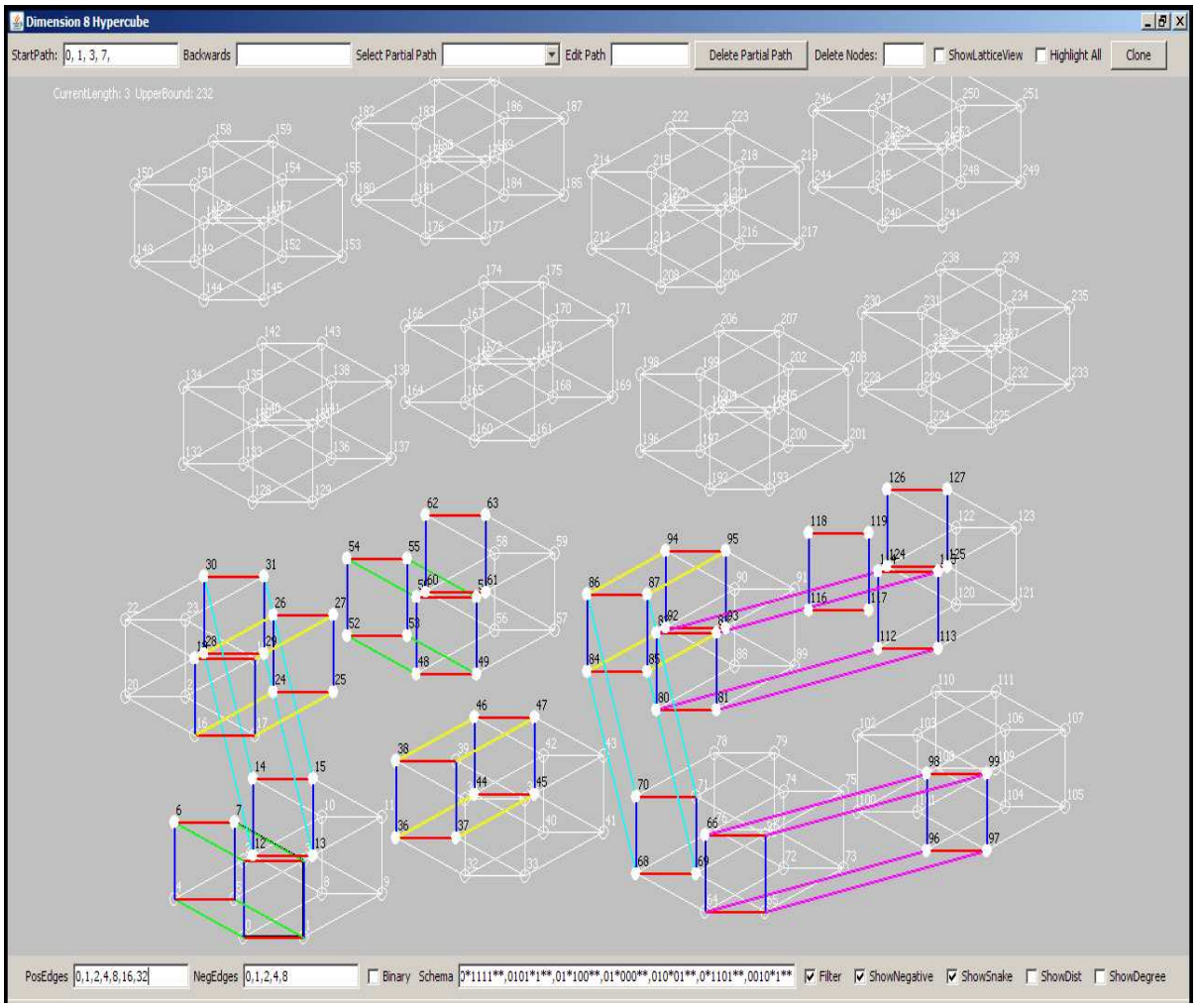

Figure 4.5: This figure shows how the IVE can be used to develop plans for the k-cube heuristic. Highlighted in this figure are vertices from the 3-cubes defined by the plan (00000***, 000*11**, 0001*0**, 00110***, 0*1111**, 0101*1**, 01*100**, 01*000**, 010*01**, 0*1101**, 0010*1**).

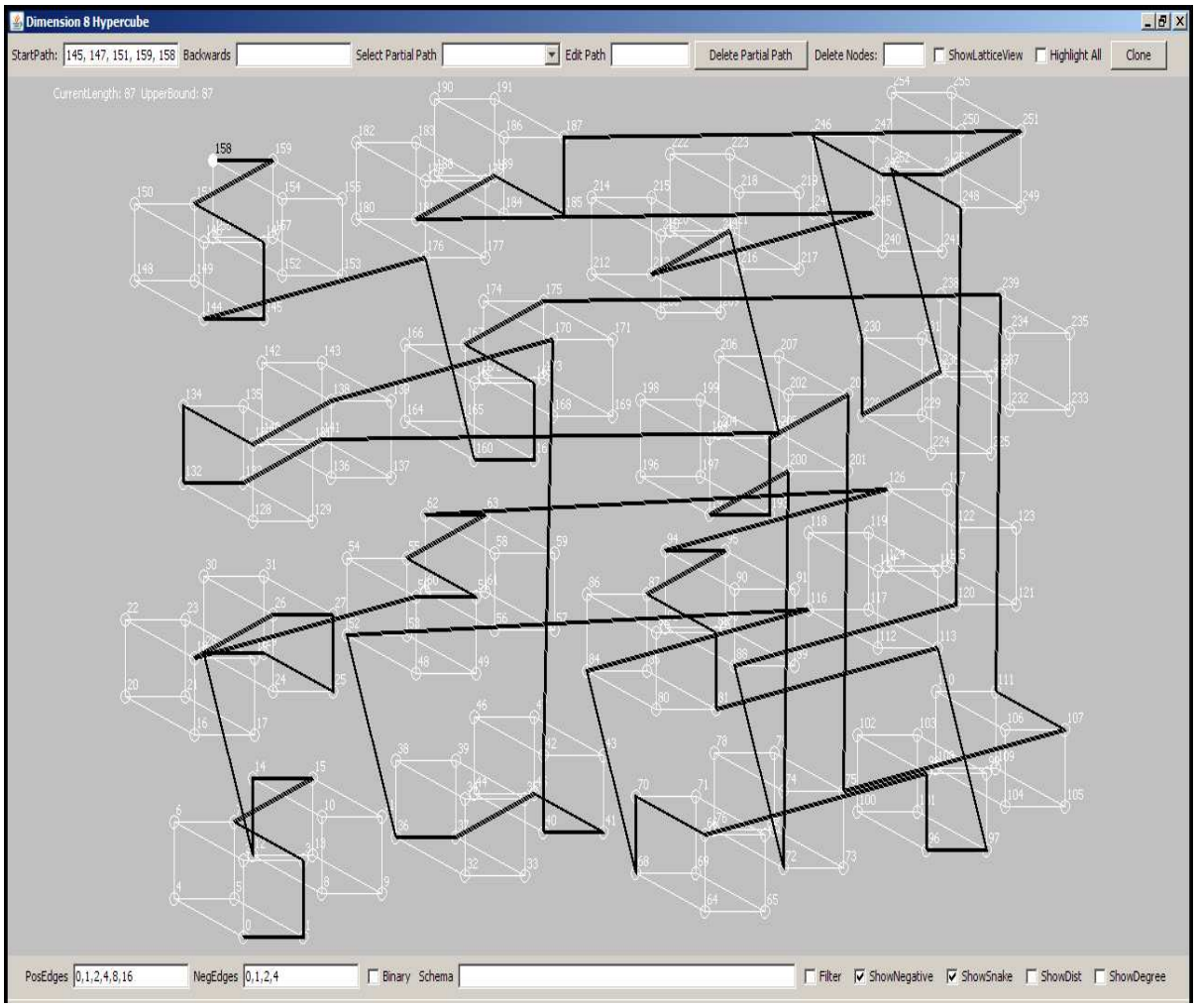

Figure 4.6: This is the length 87 snake found by the k-cube heuristic when given the plan (00000***, 000*11**, 0001*0**, 00110***, 0*1111**, 0101*1**, 01*100**, 01*000**, 010*01**, 0*1101**, 0010*1**). This snake was found in approximately 2 hours and started with the initial path of (0,1,3,7,15,14,12,28,29).

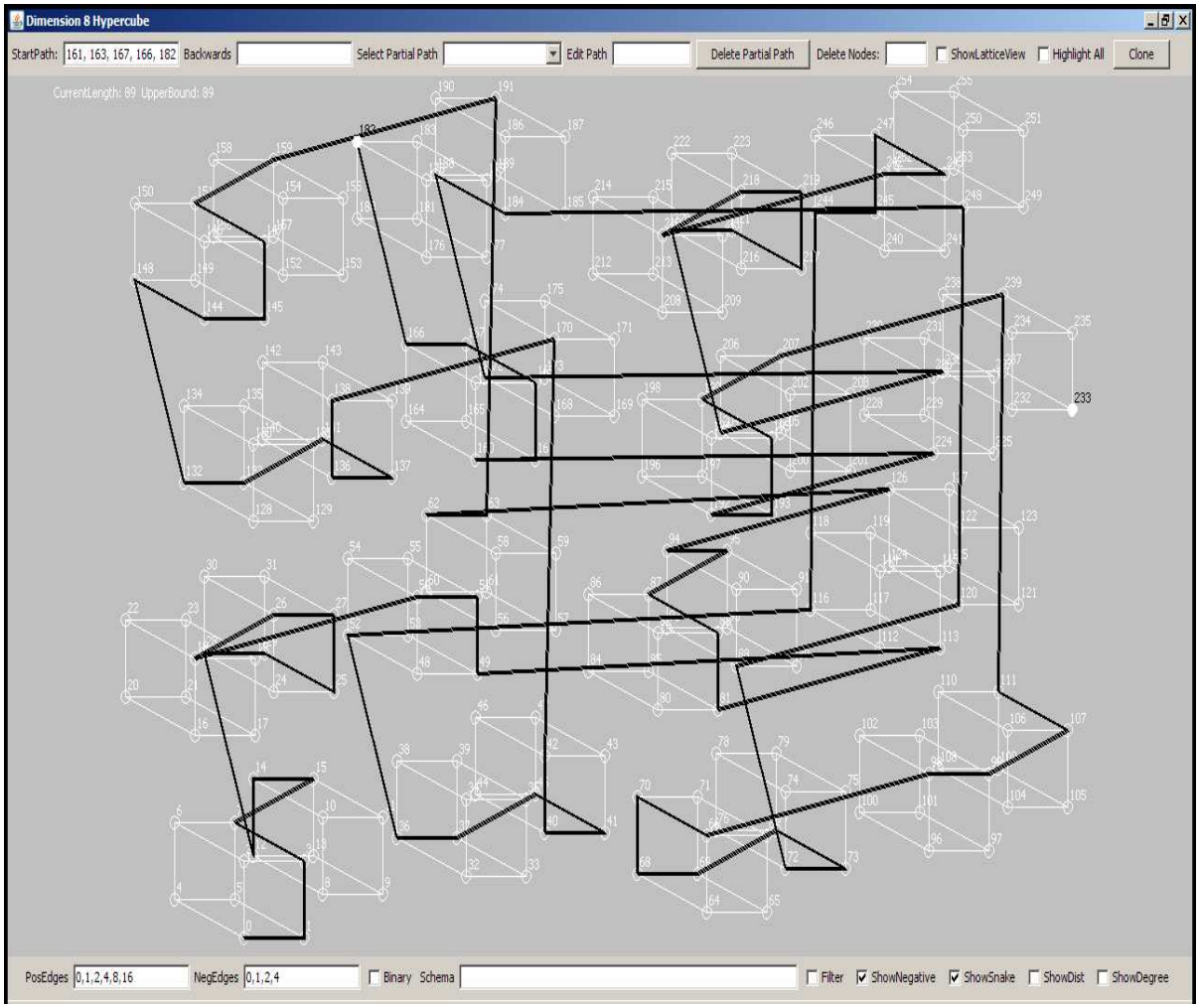

Figure 4.7: This is the length 89 snake found by the improvement heuristic when given as input the length 87 snake from figure 4.6. It took the PPBBSM less than 1 minute to find this snake once we found an initial configuration that kept only the common vertices between the two. It did, however, take 153 random tries to come up with this initial configuration.

random partial path configurations each with three partial paths plus the initial start path. For example, one random configuration from this snake could be (0,1,3,7;; 28,24,26; 55,54,52; 66,98,96). For diversity, we created 5 of these initial partial path configurations and asked the PPBBSM to find a length 44 snake from each of the 5 initial partial path configurations. We restricted the PPBBSM's run time by imposing a 2 minute time limit. The program iterates by fragmenting the resulting length 44 snakes and feeding those partial path configurations to the PPBBSM to get length 45 snakes. We increased the time limit to 5 minutes in order to allow the PPBBSM to find length 45 snakes. This procedure continued until we were left with one length 48 snake. At this point, we decided to try to focus on growing this length 48 snake into an optimal length 50 snake. We generated 40 different partial path configurations and kept the time limit to 5 minutes. The initial length 48 snake had the vertices: (0, 1, 3, 7, 6, 14, 12, 28, 29, 25, 27, 26, 18, 50, 48, 56, 40, 42, 43, 47, 45, 37, 53, 55, 119, 118, 86, 94, 95, 79, 77, 69, 68, 100, 108, 124, 125, 121, 105, 97, 99, 98, 66, 74, 72, 88, 80, 81, 83). One of the forty partial path configurations randomly generated was (0,1,3,7,6,14,12;; 29,25,27; 88,80,81; 124,125,121). From this initial configuration, the PPBBSM managed to find (0, 1, 3, 7, 6, 14, 12, 13, 29, 25, 27, 26, 18, 50, 48, 49, 53, 37, 36, 100, 68, 69, 85, 81, 80, 88, 72, 73, 75, 79, 95, 94, 86, 118, 119, 103, 99, 98, 106, 122, 123, 121, 125, 124, 60, 62, 63, 47, 43, 41, 40), which is one of the length 50 optimal snakes for Q_7.

Figure 4.7 shows that this idea also has merit in Q_8. The strategy for this experiment was to limit each run of the PPBBSM to less than 1 minute. If after 1 minute the PPBBSM could not improve on the length 87 snake, another random configuration was generated and passed to the PPBBSM. This experiment managed to produce a length 89 snake from the pieces of the initial length 87 snake, but it did take 153 different initial configurations in order to do so. It was also necessary to experiment with different fragmentation strategies. The fragmentation strategy refers to the issue of how to break a complete snake into the partial path fragments that make up the initial partial path configuration submitted to the PPBBSM. The goal of a fragmentation strategy is to produce an initial partial path

configuration with the fewest possible total number of vertices that allow the PPBBSM to give a quick answer. In dimension 7, it was sufficient to produce a configuration with only 3 partial paths with each partial path containing only 3 vertices. In dimension 8, we used the result in figure 3.8 as a basis for our strategy. It shows that an initial start path of length 14 and 8 partial paths each with length 2 (or containing 3 vertices) allows the PPBBSM to find the length 97 snake in 2.55 minutes. Experimentation showed that reducing the number of vertices on the start path and increasing the length of some of the partial paths to 3 (4 total vertices) instead of 2 improves the average time it takes the PPBBSM to find the optimal path from the initial configuration. We finally settled on a strategy to randomly choose between length 3 and length 4 partial paths. We found that it was necessary to create at least 8 partial paths in order to get an answer from the PPBBSM within 1 minute. The configuration that produced the length 89 snake shown in figure 4.7 was "0, 1, 3, 7, 15, 14, 12, 28, 29;;87, 83, 81, 113;62, 126, 94, 95;18, 50, 51;147, 151, 159;98, 66, 70;52, 36, 37, 45;42, 170, 138;107, 111, 239". Notice that we also had to include 9 vertices from the start path of the length 87 snake. The nice quality of this method is that the experiment designer has tight control over the branching factor. If many vertices are removed from the initial snake, one would expect the PPBBSM to need longer to find the better snake. The original length 87 snake was produced by the k-cube heuristic method. We tried a variety of experiments to grow a length 89 or better snake with the random scheme used to find the optimal snake in Q_7. This did not produce a snake longer than 85. The next section discusses how one might want to incorporate other search techniques to improve the Q_8 performance.

4.4 THE CASE FOR THE GENETIC ALGORITHM AND SIMULATED ANNEALING

Using the iterative improvement technique, we found a length 50 Q_7 snake from a randomly generated length 35 snake. The random snake that started the iterative process was actually our third initial random snake. The previous two snakes did not produce a length 50 snake

within the PPBBSM time limit we imposed. One very nice attribute of Q_7 is that there is a list of 12 canonical length 50 snakes. This allowed us to compare the random partial path configurations from various stages of the iterative algorithm to the list of optimal Q_7 snakes. We looked at partial path configurations from the late iterative rounds for each of the 3 initial random snake experiments. We noticed that many partial path configurations had individual partial paths that were completely found on the optimal snakes. The Genetic Algorithm (GA) would be an approach that could start with an initial population of individuals represented as a partial path configuration. Each gene could correspond to one of the partial paths in the initial partial path configuration passed to the PPBBSM. The fitness of the individual could be the longest snake the PPBBSM could find through the individual's partial paths. Crossover could be achieved by taking two partial path configurations and swapping individual partial paths to form the offspring. Holland's book [4] gives the original formulation of the GA.

Further comparisons showed that sometimes all partial paths for a single partial path configuration had vertices that could be found on a single length 50 Q_7 snake. We noticed that if some of the configurations were barely modified, the PPBBSM would have been able to find the optimal path. For example, some of the partial paths within an individual configuration had two of the three vertices from an optimal snake. The GA is a population-based approach during which different individuals of the population are combining genetic material to produce better fit offspring. Simulated Annealing (SA) on the other hand, works to optimize an individual solution by generating a neighboring candidate solution similar to it and probabilistically decides whether to adopt it as the current solution [12]. "The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects" (Wikipedia, 2007). The following is a brief sketch of how SA could be used to optimize a candidate solution.

The initial solution could be a random partial path configuration (e.g. 0,1,3,7,6,14,12;; 29,25,27; 82,80,81; 124,125,121). This configuration is very similar to the one that led to finding the length 50 snake in Q_7 described above, except that the (82,80,81) partial path is instead (88,80,81). We can imagine that the PPBBSM manages to find a length 40 snake through our initial configuration less than 1 minute, but it doesn't find a length 41 snake. Thus, we can consider that the Energy (E) of our current solution (or state) is 40. Next, we could modify our solution to generate a "nearby" state (e.g. 0,1,3,7,6,14,12;; 29,25,27; 88,80,81; 124,125,121). This is the configuration that led to a length 50 snake, but it took longer than 1 minute for the PPBBSM to find it. Let's say that we are restricting the run time to 1 minute, and that the PPBBSM managed to find a length 48 snake. The probability of accepting a solution (or moving to the next state) is $e^{-\Delta E/T}$. E in the equation would correspond to the maximum length that could be found by the PPBBSM through an initial configuration. T refers to a synthetic temperature parameter. The SA algorithm starts with a high temperature and employs a gradual cooling schedule. At the highest temperatures, the probability is almost 1.0 that we will accept the neighboring state. As temperature decreases, accepting a neighboring state depends more on whether ΔE is significant. In practice, if the candidate solution is much worse than the existing solution, a researcher may want to use a default ΔE value that minimizes the probability of accepting that state.

Chapter 5

CONCLUSION AND FUTURE DIRECTION

Artificial Intelligence offers heuristic search for problems whose computational complexity overwhelm today's computing technology. The snake-in-the-box problem is an example of such a problem. A heuristic search technique is only as good as the heuristic behind it. The framework from this thesis addresses the source of heuristic ideas. It provides what could be considered a novel heuristic development environment. The main components of the framework are its Interactive Visualization Environment (IVE), Partial Path Branch and Bound Search Module (PPBBSM), and extensible APIs and utility methods that facilitate rapid implementation and evaluation of heuristic ideas.

The experimental results from chapter 3 show that the PPBBSM can speedily (only 3 minutes on a slow laptop) find a length 50 path in Q_7 through an initial configuration consisting of several partial paths defined over a total of only 13 vertices. Actually, four of these vertices were the vertices that begin the start path for every canonical snake (0,1,3,7). In a sense, the PPBBSM allowed the Q_7 problem to be redefined as the search for three small partial paths that make up a total of 9 vertices. The fact that the PPBBSM performs best when the initial partial paths are well-spaced is actually an advantageous feature of its approach. It means that the actual placement of the initial partial paths can be constrained by defining disjoint regions from which the initial small paths are sampled. Section 4.1 describes how the framework offers tools that can be used to derive rules that govern useful initial partial path placements.

Chapter 4 presents a case study showing how interaction with the framework can yield interesting heuristic ideas. The k-cube heuristic allows a researcher to use intuition gained

from the IVE to define high a level plan that constrains the snake's initial moves and limits the branching factor. Continued and focused use of the IVE can lead to a formalization of what makes a good plan. If this happens, the plan can confidently be extended to include more k-cubes. This would impose more constraints thereby reducing the overall time to find snakes consistent with the defined plan.

A random iterative improvement heuristic technique was devised to try to grow the snake found by the k-cube heuristic. It works by chopping a given snake into random fragments and submits the fragmented snake to the PPBBSM to try to find a longer path. The results show that the technique has potential but could be improved if the modifications were more intelligently targeted. For instance, the individual partial paths that are submitted to the PPBBSM could be more rigorously inspected and modified according to a new heuristic. This would make the iterative improvement technique more informed which could lead to quicker convergence to a local or global optimum.

The framework described in this thesis will be increasingly useful as the PPBBSM takes less time to search higher dimensions with as few initial partial paths defined as possible. It has anticipated relevant heuristic approaches that future researchers should consider. It has anticipated these approaches by defining extension points that can have significant impact on the PPBBSM's performance. It also provides a wealth of tools in the form of Java utility methods that future implementations of the interfaces can freely make use of. The appendices aim to make it as easy as possible to understand the details of how to immediately start using the significant features of the comprehensive framework. Appendix A gives a basic user guide for the PPBBSM and IVE. Appendix B attempts to familiarize a future developer with the basic model of the framework and demonstrates the wealth of utilities the framework offers. Appendix C shows how easy it is to implement the interfaces that make up the extension points. Appendix D shows the main recursive method of the PPBBSM. It also shows how the PPBBSM interacts with the PartialPathHyperCubeWrapper, the implementation of the ForwardNeighborSelector, and the UpperBoundEstimate implementation. Finally, Appendix

E shows a set of test cases that offer support for the correctness and reliability of the framework itself.

Bibliography

[1] Casella, D., and Potter, W. New lower bounds for the snake-in-the-box problem: Using evolutionary techniques to hunt for snakes. In *Proceedings of the 18th Florida Artificial Intelligence Research Society Conference.* Clearwater Beach, FL. (2005) 264-269.

[2] Dorn, J. , Girsch, M., Skele, G., and Slany, W. (1996) Comparison of Iterative Improvement Techniques for Schedule Optimization, *European Journal of Operational Research*, pp. 349-361 Vol 94, No 2.

[3] Harary, F.; Hayes, J.; and Wu, H. 1988. A survey of the theory of hypercube graphs *Computational Mathematics Applications* 40:277-289.

[4] Holland, J.H. 1975. *Adaptation in Natural and Artificial Systems.* Ann Arbor: The University of Michigan Press.

[5] Kautz, W. H. 1958. Unit-Distance Error-Checking Codes. *IRE Trans. Electronic Computers* 7:179-180.

[6] Kochut, K. 1996. Snake-in-the-box codes for dimension 7. *Journal of Combinatorial-Mathematics and Combinatorial Computations* 20:175-185.

[7] Konig, Dierk (2007) *Groovy In Action.* New York, NY: Manning Publications.

[8] Dayanand S. Rajan and Anil M. Shende. Maximal and reversible snakes in hypercubes. In 24th Annual Australasian Conference on Combinatorial Mathematics and Combinatorial Computing, 1999. http://citeseer.ist.psu.edu/rajan99maximal.html

[9] D.R. Tuohy, W.D. Potter and D.A. Casella, "Searching for Snake-in-the-Box Codes with Evolved Pruning Methods", in *Proceedings of the 2007 International Conference*

on Genetic and Evolutionary Methods. (GEM '07) pp 3-9, Las Vegas, Nevada June 25-29, 2007.

[10] D. Whitley. A Genetic Algorithm Tutorial, *Statistics and Computing (4)*:65-85, 1994.

[11] Wikipedia contributors, "Branch and bound," Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Branch_and_bound&oldid=174256538 (accessed December 3, 2007).

[12] Wikipedia contributors, "Simulated annealing," Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=174716983 (accessed December 3, 2007).

Appendix A

USER GUIDE

This thesis has discussed how the PPBBSM and the IVE are main components of a comprehensive framework for the snake-in-the-box problem. Section A.1 describes how to install and run the PPBBSM for the initial configurations described in the experimental results. Section A.2 explains how to set up the IVE by discussing how to modify its configurable properties. Currently, the binary distribution of the framework comes as a single file called "snakedist.zip". Unzipping the snakedist.zip file reveals that there are four files inside. The snake.jar is all the compiled java code for the entire framework. "search.properties" is the property file for the ppbbsm. "snake.properties" is the property file for the IVE. The ppbbsm.bat file will run the PPBBSM in a windows environment. This script is just a one line call to get the java virtual machine to run the main PPBBSM class, so it can easily be ported to a Unix/Linux environment. If a user is using the IVE and wants to run a given configuration through the PPBBSM, Appendix A.2 shows how this can quickly be done using the "Clone" functionality within the IVE.

A.1 PPBBSM USER GUIDE

The PPBBSM gets its initial configuration from the "search.properties" properties file. In order to run the PPBBSM, the user must edit the search.properties file and run the windows script "ppbbsm.bat". An example search.properties file from the experimental results will help understand how to set up custom searches.

```
initialSnake=0, 1, 3, 7;;43, 59, 63; 96, 104, 72; 116, 118, 119;80,81,85
```

```
dimension=7
minimumLength=50
UpperBoundEstimateImpl=snakes.graph.compute.StandardUpperBound
ForwardNeighborImpl=snakes.graph.compute.options.ExhaustivedNeighborSelector
ForwardNeighborImplProps=2
greedy=true
```

Section 2.4 introduced the syntax used to define an initial configuration. The initialSnake in this example starts with the canonical (0,1,3,7) and has four internal partial paths. Next the hypercube dimension the PPBBSM will search is specified. The "minimumLength" property represents the user's defined lower bound. Appendix C will go into more detail for the next three fields, but for now it is important to note that the UpperBoundEstimateImpl and ForwardNeighborImpl are the properties that make up the primary framework extension points. The "greedy" property controls whether the framework seeks to automatically extend paths that can legally be extended. For instance, after a forward path move a situation in which there is only one forward neighbor can arrive. If greedy is true, it will always extend the snake automatically. The option exists because sometimes it is annoying when working with the IVE for the IVE to extend a path without the user's awareness of how it was able to be done.

A.2 IVE USER GUIDE

Section 2.2 gave an overview of the basic controls of the IVE. This Appendix will build on that overview by discussing the IVE's config file (snake.properties).

As mentioned in the introduction, one way to render Q_n from Q_{n-1} is to project a copy of Q_{n-1} onto a new axis. A computer screen is only a 2D surface, so it can quickly get confusing visualizing a 7 or 8 dimensional hypercube. One thing that helps is to divide the screen into disjoint regions and render smaller Q_k for $k < n$ in non-overlapping regions. The sample "snake.properties" file below corresponds to the IVE display found in figure 3.6. The "gd=5" in the file means that the user will need to specify the locations of the four disjoint 5-cubes. The "originLocations" property is what the IVE uses to place the zero

vertex for each 5-cube. The value of the "originLocations" property is a comma separated list of integer pairs. This helps to see how the IVE is reading this list of integer pairs to get the (x,y) location of the zero vertex for each of the four zero vertices of the 4 disjoint 5-cubes. In Java Swing, the (0,0) coordinate is the top left corner of the display. Thus, a smaller value for x puts the zero vertex closer to the left side of the screen. A smaller value for y puts a zero vertex closer to the top of the screen. The zero vertices are (0, 32, 64, 96). The edge distances property allows the user to control the number of pixels of each edge transition type. The comma-separated list values have an order. The first integer (55) corresponds to the pixel length of edges for the bit 1 transition. This means that all the red edges will be 55 pixels long. The following edges are examples from figure 3.6: (20,21),(72,73),(48,49). All bit 2 transition edges will have a pixel length of 50. These are the vertical edges such as (14,12), (15,13), (28,30), etc. For this properties file example, only the edge lengths for the disjoint 5-cubes need to be specified because the edges connecting the different 5-cubes have lengths that are necessary products of where they are placed (via the originLocations property). The easiest way to understand how to do it is by just changing one of the numbers and seeing how that affects the display.

The "initialSnake" property gives the user a quick way to get the IVE to display an initial set of partial paths without having to manually enter the paths in each of the path edit boxes. However, when the user wants to edit paths, all he or she needs to do is to edit the path fields with valid node values and hit return. The user may also delete a partial path by selecting the path from the partial path combo chooser, and then hitting the delete button. When the user wants to introduce a new partial path, the IVE knows that if the user types something entirely new in the "Edit Path" field, the user is adding a path rather than editing an existing one. This may seem confusing, but it only takes a little practice to get the hang of working with the IVE.

```
initialPositiveEdges=0,1,2,4,8,16
initialNegativeEdges=0,1,2,4,8
showDistance=false
```

```
drawNegativeSpace=false
showSnake=true
showBinary=false
showDegree=true
nodeDim=10

gd=5
pd=7

originLocations=(345,655), (350,320), (760,655), (765,325)
edgeDistances=55,50,65,95,90

initialSnake=0, 1, 3, 7;;43,59,63;96,104,72;116,118,119;80,81,85
```

Appendix B

BASIC MODEL OF THE FRAMEWORK

Figure B.1 shows a basic uml diagram of the object-oriented model of the framework. This is intended to give a programmer wanting to extend the framework the idea for how the snake-in-the-box problem and hypercube are modeled together under one framework. The main recursive method (findSnakes) of the PPBBSM is found in Appendix D. The object that is passed to the findSnakes method is of type PartialPathHyperCubeWrapper. Figure B.1 shows how the PartialPathHyperCubeWrapper can serve as the main abstraction for the PPBBSM. It is the glue that holds together hypercube information with partial path information. The uml diagram includes method names as conceptual information that the object model provides. The diagram can be interpreted linguistically as a PartialPathHyperCubeWrapper that extends the functionality of a basic HyperCube. The PartialPathHyperCubeWrapper is an aggregate of a collection of HyperCubePaths. PathFromStart and PathFromEnd are derived from HyperCubePath. HyperCubePath itself, extends the java.util.LinkedList. PartialPathHyperCubeWrapper is also associated with an EndPointCoveringTest. The EndPointCoveringTest deals with the constraint checking described in Section 2.4.3.

Figure B.2 shows a group of utility classes that offer static utility methods. These utility methods proved very useful when implementing the functionality to check constraints. The PathMergeUtility handles the complicated task of merging the forward path to partial paths. The PathSetUpdateManager handles the job of knowing when the IVE of PPBBSM is wanting to extend an existing path, redefine an existing path, or create a new path. It then takes care of updating the working version of the PartialPathHyperCubeWrapper instance.

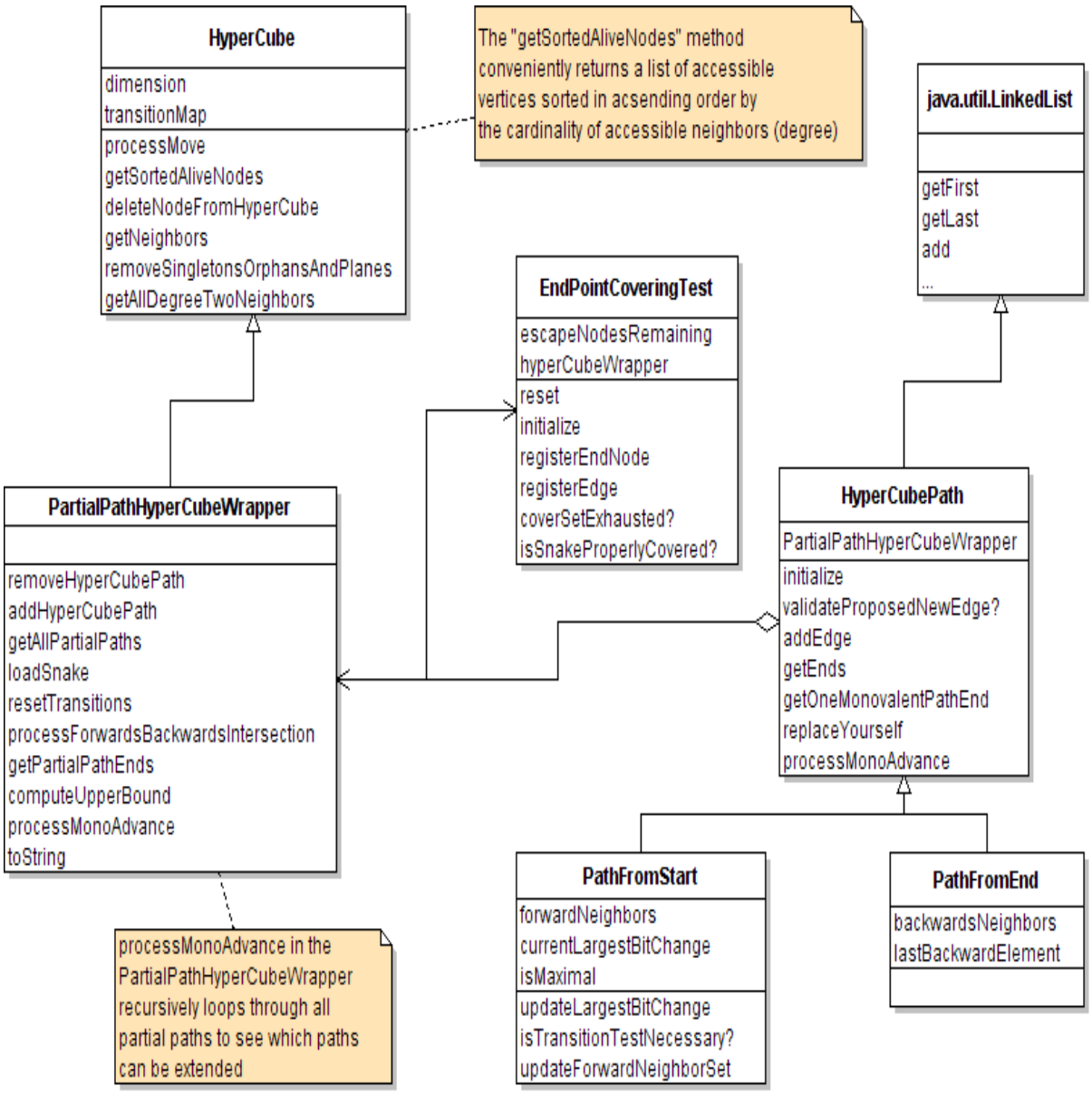

Figure B.1: A display of the object-oriented model of partial paths and the hypercube. The main class is the PartialPathHyperCube wrapper. The PPBBSM works primarily with it for its representation of potential solutions. It keeps track of all partial paths and initiates all the constraint checking that has been described in this thesis. Each box has a sample of some of the most useful methods that characterize each class. As this is just a conceptual diagram, the method full method signatures are not displayed. The "?" symbol at the end of method name denotes a boolean return value method.

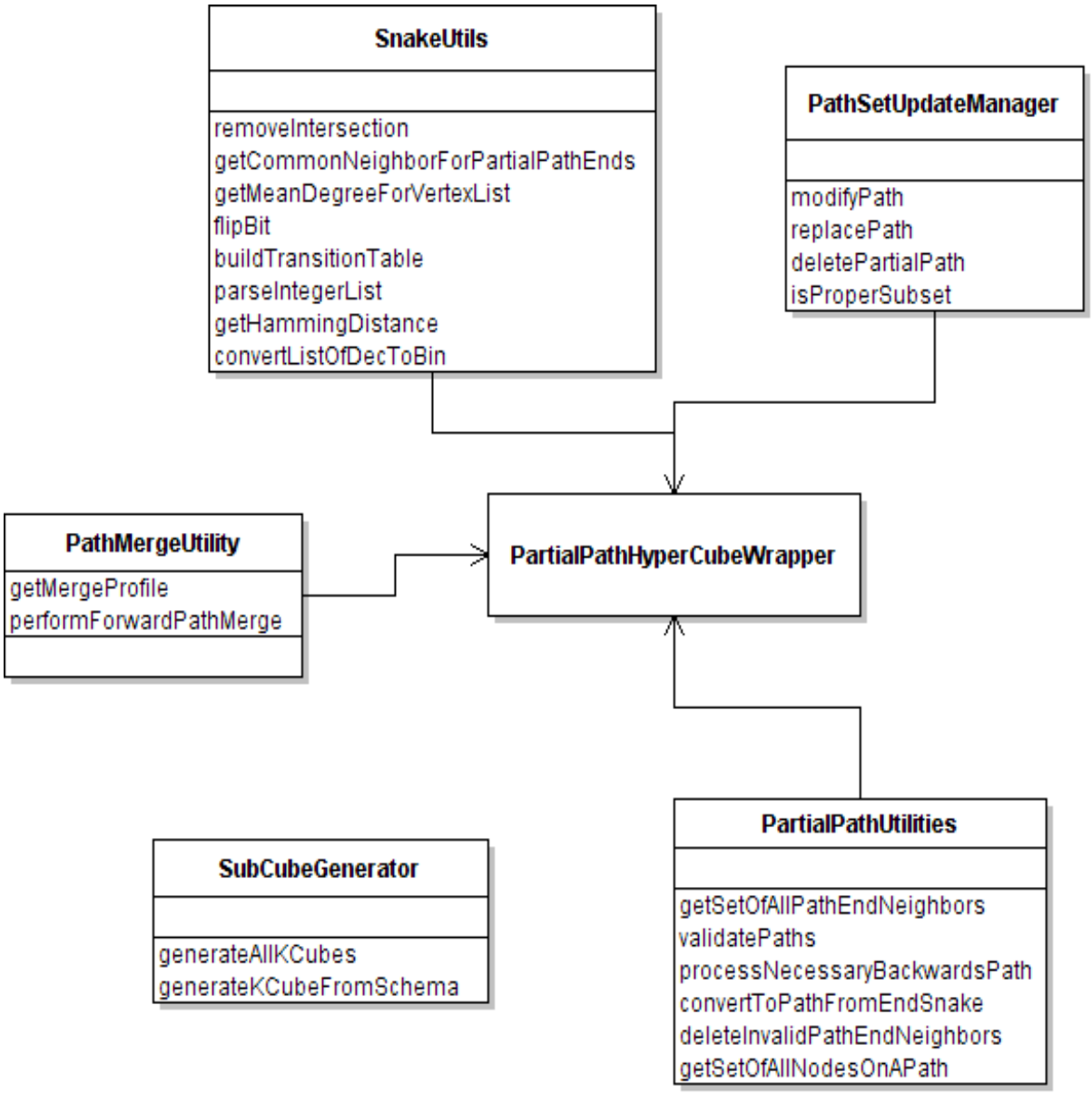

Figure B.2: A display of the static Java Utility Classes that help make extension from extension points a simpler task. This diagram shows that these utility classes provide information about an instance of a PartialPathHyperCubeWrapper. In some cases, the PartialPathHyperCubeWrapper instance passes itself to these utility classes so that they can inspect and update the instance.

Appendix C

EXAMPLE INTERFACE IMPLEMENTATIONS

Appendix B shows the basic model of the framework and utility methods that can be useful for anyone wanting to extend the framework from the previously mentioned extension points. The Sections of this Appendix will show how simple it is to implement the interfaces that designate the framework extension points.

C.1 FORWARD NEIGHBOR SELECTOR

Any implementation of ForwardNeighborSelector Java Interface must implement two methods.

```
public List<Integer> selectForwardNeighbors(
                PartialPathHyperCubeWrapper ppGraphWrapper);

public void setProperties(String commaSepProperties);
```

Imagine a scenario in which a researcher wants to limit the branching factor by only selecting at most 2 forward neighbors for the PPBBSM to consider. Furthermore, imagine that the researcher wants to come up with a way to identify the 2 "best" forward neighbors to try. In order to validate their approach, let's say the researcher wants to compare it to a baseline method of using 2 random vertices. Given the framework, it would be a simple matter to implement such a scheme. In fact, the Java code below is all it takes in order to do so.

```
package snakes.graph.compute.options;
```

```
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import math.utils.CollectionUtils;
import snakes.graph.PartialPathHyperCubeWrapper;
import snakes.graph.PathFromStart;

public class RandomSubsetSelector implements ForwardNeighborSelector {
  protected int max;
  Random random;
  public RandomSubsetSelector () {
    random = new Random();
  }

  public List<Integer> selectForwardNeighbors(
     PartialPathHyperCubeWrapper ppGraphWrapper) {

  PathFromStart fPath = ppGraphWrapper.getForwardPath();
  Set<Integer> fNeighbors = new HashSet<Integer>(fPath.getForwardNeighbors());
  List<Integer> randomForwardNeighbors =
     CollectionUtils.convertSetToList(fNeighbors);
  if (randomForwardNeighbors.size() > max) {
    do {
      if (random.nextBoolean()) {
        int toDelete = random.nextInt(randomForwardNeighbors.size());
        randomForwardNeighbors.remove(toDelete);
      }
    }
    while (randomForwardNeighbors.size() > max);
  }
   return randomForwardNeighbors;
}
public void setProperties(String commaSepProperties) {
  String[] props = commaSepProperties.split(",");
  max = Integer.parseInt(props[0]);
 }
}
```

C.2 UPPER BOUND ESTIMATE

The following source code implements phase 3 described in 2.4.1.

```
package snakes.graph.compute;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import snakes.graph.PartialPathHyperCubeWrapper;

public class ShallowUpperBound implements UpperBoundEstimate {

  public int estimateUpperBound(PartialPathHyperCubeWrapper
       partialPathHyperCubeWrapper) {

    Set<Integer> ppEnds = partialPathHyperCubeWrapper.getPartialPathEnds();
    int upperBoundLength =
         partialPathHyperCubeWrapper.computeUpperBoundSnakeLength();

    if (partialPathHyperCubeWrapper.getForwardPath().isMaximal()) {
      return partialPathHyperCubeWrapper.getForwardPath().size()-1;
    }
    else {
      Set<Integer> processedNeighbors = new HashSet<Integer>();
      // prevents double counting
      int necessaryHyperCubeSubtraction = 0;

      for (Iterator iter = ppEnds.iterator(); iter.hasNext();) {
        Integer ppEnd = (Integer) iter.next();
        try {
          Set<Integer> neighborsCopy = new HashSet<Integer>(
              partialPathHyperCubeWrapper.getAdjacentNodes(ppEnd));

          neighborsCopy.removeAll(processedNeighbors);
          necessaryHyperCubeSubtraction += Math.max(0,neighborsCopy.size()-1);
          processedNeighbors.addAll(
                  partialPathHyperCubeWrapper.getAdjacentNodes(ppEnd));
      }
      catch (Exception e) {
        e.printStackTrace();
        return -1;
      }
    }
    upperBoundLength -= necessaryHyperCubeSubtraction;
  }
  return upperBoundLength;
  }
```

```
}
```

Appendix D

PPBBSM SOURCE CODE

The source code below is the main recursive method of the PPBBSM. It is basically a standard depth-first search, but the search space and backtracking comes from information sources offered by the framework. The PPBBSM stops if the snake is maximal and its length is greater than the user defined lower bound.

```
public void findSnakes(PartialPathHyperCubeWrapper ppGraphWrapper) {
  String currentSnakeString = ppGraphWrapper.toString();
  PathFromStart startPath = ppGraphWrapper.getForwardPath();
  List<Integer> lstOfForwardNeighbors =
  forwardNeighborSelector.selectForwardNeighbors(ppGraphWrapper);

  if (lstOfForwardNeighbors != null) {
    for (int i=0; i<lstOfForwardNeighbors.size(); i++) {
      try {
        List<Integer> newElements =
             getNewElements(startPath, lstOfForwardNeighbors.get(i));

        PathSetUpdateManager.modifyPath(startPath, newElements);
        if (ppGraphWrapper.getForwardPath().isMaximal() &&
             (ppGraphWrapper.computeCurrentSnakeLength()>=lowerBound)){

           System.out.println("FOUND GOAL!!" + ppGraphWrapper.toString());
           return;
        }
        else {
          if (!ppGraphWrapper.getForwardPath().isMaximal() &&
                (ppGraphWrapper.computeUpperBound() >= lowerBound)) {

            findSnakes(ppGraphWrapper); // recursive call
            if (ppGraphWrapper.getForwardPath().isMaximal() &&
                (ppGraphWrapper.computeCurrentSnakeLength()>=lowerBound)){
               return;
```

```
            }
          }
        }
      }
      catch (Exception e) {
        e.printStackTrace();
      }
      ppGraphWrapper.reset();
      ppGraphWrapper.loadSnake(currentSnakeString);
      startPath = ppGraphWrapper.getForwardPath();
    }
  }
}
```

Appendix E

VALIDATION JUNIT TEST CASE EXAMPLE

Putting in all the framework functionality involved a significant programming task with numerous possibilities for bugs to emerge. The JUNIT test framework was immensely helpful to ensure that new functionality didn't break existing functionality during the development effort. There are several Test classes similar to the following. There are a total of 40 test cases included in the JUnit test suite.

```
package test;

import snakes.graph.HyperCubePath;
import snakes.graph.PartialPathHyperCubeWrapper;
import snakes.graph.PathFromStart;
import snakes.utils.PathSetUpdateManager;
import snakes.utils.SnakeUtils;
import junit.framework.TestCase;

public class BasicPartialPathTest extends TestCase {
  PartialPathHyperCubeWrapper ppGraphWrapper;
  PathFromStart forwardPath;

  protected void setUp() throws Exception {
    super.setUp();
    ppGraphWrapper = new PartialPathHyperCubeWrapper(6, true);
    forwardPath = new PathFromStart(
       SnakeUtils.parseIntegerList("0,1,3,7"), ppGraphWrapper);

  }

public void testPPathAdd() {
  HyperCubePath aPath =
    new HyperCubePath(
       SnakeUtils.parseIntegerList("53,37,36"),ppGraphWrapper,true);
```

```
  System.out.println(ppGraphWrapper.getAllDefinedPaths());
  assertTrue(ppGraphWrapper.getPartialPathEnds().contains(36));
}

public void testPartialPathMerge() {
  HyperCubePath aPath =
    new HyperCubePath(
      SnakeUtils.parseIntegerList("37, 36, 38"),ppGraphWrapper,true);

  HyperCubePath aPath2 =
    new HyperCubePath(
      SnakeUtils.parseIntegerList("62, 63, 59"),ppGraphWrapper,true);

  try {
    PathSetUpdateManager.modifyPath(
      aPath, SnakeUtils.parseIntegerList("37, 36, 38, 46"));


    assertEquals(ppGraphWrapper.getForwardPath().getLast().intValue(), 13);
    assertEquals(ppGraphWrapper.getAllDefinedPaths().size(), 2);

    System.out.println(ppGraphWrapper.getAllDefinedPaths());
   } catch (Exception e) {
     e.printStackTrace();
   }
   assertFalse(ppGraphWrapper.getForwardPath().isAtDeadEnd());
  }
}
}
```