SEARCHING AND PLANNING IN THE GAME OF HEX

by:

Hugh Brendan McMahan

A thesis submitted in partial fulfillment
for graduation with Honors in Mathematics

Whitman College
2000

Certificate of Approval

This is to certify that the accompanying thesis by Hugh Brendan McMahan has been accepted in partial fulfillment of the requirements for graduation with Honors in Mathematics.

Douglas R. Hundley

Whitman College
April 27, 2000

Contents

List of Figures

Searching and Planning in the Game of Hex

1 Introduction

The game of Hex may be unfamiliar to the reader, so before proceeding to our primary results we provide some background on the game.

1.1 The Game of Hex

The Danish mathematician and poet Piet Hein invented (or discovered) the game of Hex in 1942, and John Nash rediscovered the game independently in the United States in 1948 while at Princeton. Martin Gardner brought the game to wider attention when he discussed it in his Mathematical Recreations column in Scientific American (see [9]).

The simple rules and elegant strategy of the game have an intrinsic appeal. Two players, black and white, alternate placing stones on a $n \times n$ grid of hexagons. Two opposite sides of the board, typically the top and bottom, are designated as the black goals, and the other two sides are the white

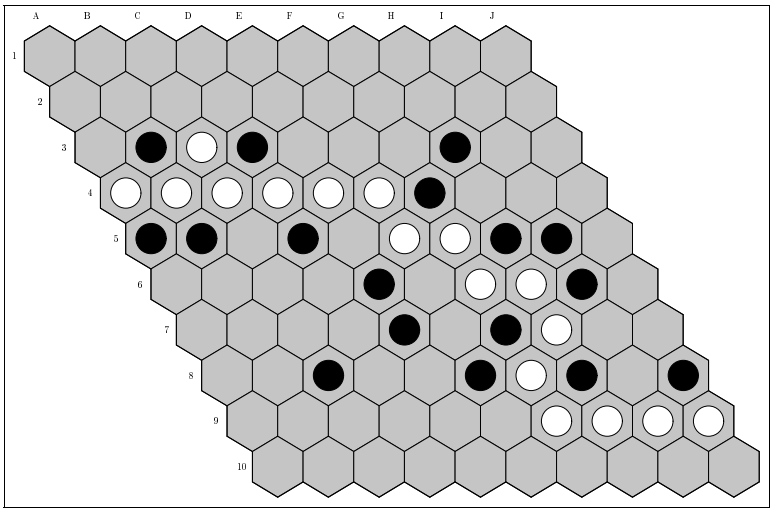

Figure 1: A Game Won by White

goals. Black attempts to establish a path of black pieces connecting the top to bottom, and white attempts to establish a path of white pieces from left to right. Figure 1 shows the final position in a typical game.

1.2 The Goal of This Thesis

In addition to being a fascinating game to play, the game of Hex has received the attention of researchers in mathematics and artificial intelligence. The game is of interest to mathematicians because it has deep combinatorial structure. Hex is in fact equivalent to an important problem in graph theory; the details of this interpretation will be explored in Section 2.1. Certain properties of Hex (and generalizations of Hex) have interesting analogues in topology ([8], [2]).

Computer scientists working on game-playing programs have become interested in Hex for other reasons. One reason is that Hex, like Go, has a high branching factor. That is, in most positions the player to move can choose from a large set of legal moves. Brute force searching strategies are greatly limited by this high branching factor, which has made writing computer programs to play such games difficult. Games such as Othello, checkers, and chess all have much lower branching factors which has made possible the creation of world-championship level computer programs for those games.

Hex is also a good testbed for ideas in searching and game playing because the simple rules and clean mathematical structure of the game make testing and analyzing algorithms easier than for more complex games. Also, since Hex can be played on a board of arbitrary size, the branching factor and hence game complexity can easily be varied. Several recent papers make note of these advantages and describe other approaches to creating programs for Hex ([3], [12]).

In this thesis we will describe a computer program called Hexane that plays the game of Hex. It does so by using a specialized search algorithm to discover local features of a game position (namely, forced connections), and then attempts to use this local information to determine the important or active region of the board and select a good move. We begin by considering a graph-theoretic description of Hex and establishing some useful terminology in Section 2. We then give an overview of the goals of the Hexane project in Section 3. Section 4 details the searching algorithm used by Hexane.

The section begins with an overview of the minimax search technique and then Section 4.2 provides some details of Hexane's implementation. Various techniques for using the local connections discovered by the search algorithm are discussed in Section 4.4.

2 Theoretical Results and Terminology

Before discussing the Hexane program in detail we introduce some useful notation and definitions, and mention some important theoretical results about Hex. First we consider how the game of Hex can be described in graph theoretic terms.

2.1 Hex and the Shannon Switching Game

The game of Hex can be viewed as a special instance of the Shannon switching game played on the vertices of a graph (SSGV).

An instance of the Shannon switching game is given by a connected graph $G = \langle V, E \rangle$ with two special vertices (say u and v). There are two players in the game, a "positive" player and a "negative" player. The positive player tries to select vertices that comprise a path between the vertices u and v and the negative player tries to select vertices in such a way as to prevent this by blocking all paths between u and v. The players alternate selecting vertices until one of the players wins. The positive player wins by selecting all vertices on a path between u and v. The negative player wins by selecting enough vertices so that no such positive u-v path can be marked by the positive player. A more detailed description of the Shannon switching game is provided in [5].

It should be noted that for the Shannon switching game on the edges of a graph (the players select edges rather than vertices) a polynomial-time

algorithm exists to determine which player wins and to find the winning strategy. However, the general problem of the Shannon switching game on the vertices of a graph is computationally difficult, and no efficient algorithms for solving it are believed to exist [10].

The game board for Hex can be described as a graph. We let each hexagon on the board correspond to a node in the graph, and add an edge between nodes in the graph if the two corresponding hexagons are adjacent on the game board. We also introduce four special nodes, one for each side of the game board. These nodes are connected to all nodes on their respective edges of the game board. For example, using the board from Figure 1, in the corresponding graph we define a "top" node adjacent to the nodes for hexes $A1, B1, \ldots, J1$. Since the top edge is a black goal, we consider this top node to always be played by black. Similarly, we define a black bottom node and white left and right nodes. We call this graph a Hex board graph. Figure 2 shows a 3×3 Hex board graph, where the four larger nodes represent the goal nodes. The two nodes labelled B are black's goal nodes corresponding to the top and bottom edges of the Hex board, and the two nodes labelled W are white's goal nodes, corresponding to the left and right edges of the Hex board. We use the term SSGV graph to refer to a general instance graph for an SSGV problem, but assume nodes in the SSGV graph are colored white, black, or empty as with the Hex board graph.

It can be proved that a Hex board where all the hexes have been played contains either a path of black hexes connecting the top and bottom or a

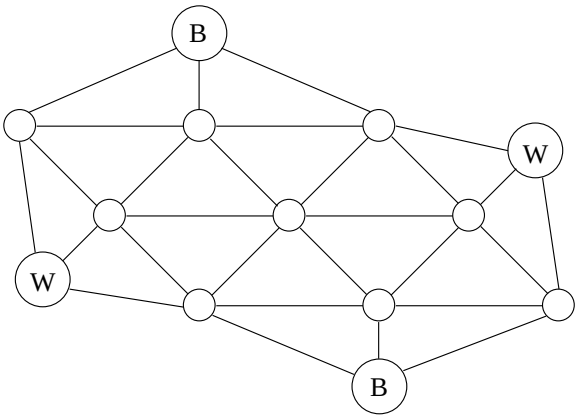

Figure 2: A 3×3 Hex Board Graph

path of white hexes connecting the left and right, but not both. Though this result may seem intuitively obvious, a formal proof requires a little work. One proof can be found in [4]. This no-draw result turns out to be equivalent to the Brouwer fixed point theorem in topology [8].

Since success for black in connecting the top and bottom implies that no path exists for white, Hex is equivalent to the Shannon switching game on a Hex board graph where the top and bottom nodes are viewed as the goal nodes and black is the positive player. We can also view Hex as an SSGV where the left and right nodes are the goal nodes, and white is the positive player. Thus for any given Hex position there are in fact two equivalent instances of the Shannon switching game on the vertices of a graph.

2.2 Some Definitions

Before proceeding to discuss the specific approach to playing Hex taken by the Hexane program we introduce some useful definitions.

player and color: There are two players in Hex, one called white and one called black. We will sometimes speak of a general player P; the other player will then be denoted $\bar{P}$. Hexes on the the hex board can be referred to as having a color; the color of a hex can be either white (the hex has been played with a white stone), black (the hex has been played with a black stone), or unplayed (the hex is open).

hex: A hex is a single hexagon on the Hex board, or the equivalent node on a Hex board graph. The terms hex, node, and vertex are used synonymously. For clarity, we will capitalize Hex when referring to the game, and use lowercase when referring to a node on the board. While the term hex is used throughout these definitions, they apply to any instance of the SSGV, unless otherwise noted.

group: A group of hexes is a maximal set of hexes R that have all been played by a player P such that there is a path (in the graph-theoretic sense) between any two hexes in R using only other hexes in R. The color of a group is the color of P, and we may refer to groups belonging to P as P-groups. Note that every played hex on the board belongs to a unique group. A hex is adjacent to a group R if it is adjacent to some hex in R.

group condensed graph: A group condensed graph $\hat{G}$ is formed from a given Hex board graph G by representing each group in G by a single vertex in $\hat{G}$. Every unplayed node in G has a corresponding node in $\hat{G}$, but each group R in G is collapsed to a single vertex v in $\hat{G}$, where v receives the same color as R. The new vertex v is adjacent to all unplayed nodes in $\hat{G}$ whose corresponding nodes in G were adjacent to R. It is easy to show that these two graphs are equivalent with regard to the SSGV and that a given Hex position or SSGV instance can be mapped to a unique group condensed graph.

directly connected: Two hexes are said to be directly connected if they are in the same group.

forcing connection: A forcing connection between two nodes u and v played by a player P on an SSGV graph G exists if in every sequence of alternating moves where player $\bar{P}$ plays first, player P can play in such a way as to eventually arrive at a direct connection between u and v. This concept is only useful if we define a set A of nodes associated with the forcing connection. The set A defines a minimal subset of nodes of G such that the forcing connection exists on the subgraph induced by the vertices of A. We call the set A the *associated set* of the connection. We say u and v are force connected for P if there exists a forcing connection between u and v. The analogous concept of a virtual connection is defined in [3]. This paper generalizes the idea so that virtual connections can exist where not both end

vertices (u and v) have been played by P.

force k-connected: Two hexes u and v played by a player P are force k-connected for P if P can play k moves (without any opponent moves) and arrive at a position where u and v are force connected. Thus, a virtual connection or forced connection is a forced 0-connection.

bridge: A bridge is a fundamental type of forcing connection. Suppose u and v are two nodes played by a player P. Let U be the set of unplayed hexes adjacent to u and let V be the set of unplayed hexes adjacent to v. A bridge exists between u and v if $|U \cap V| \geq 2$. The most common situation is where $|U \cap V| = 2$. We call the nodes u and v the nodes on the bridge, while any two nodes taken from $U \cap V$ form an associated set for the bridge.

2.3 Distance on a Hex Board

The distance between two hexes can be measured in a number of ways. The *literal distance* between two hexes u and v for a player P is the smallest number of hexes that need to be played so that u and v are in the same P-group. If either u or v have been played by $\bar{P}$, the literal distance between them is defined to be infinity. Literal distance is a metric on any hex position.

The literal distance is easy to compute via a slightly modified form of breadth-first search. Dijkstra's algorithm could also be used, but the need for a priority queue makes such an approach slower. Hexane's breadth-first search approach works in the same manner as the algorithm described in [6],

but when a P-group is encountered, it is "expanded" so all of the effective edges through the group are considered. That is, if we are calculating the distance for P, an unplayed node adjacent to a P-group is effectively adjacent to all other unplayed nodes adjacent to the group.

We also define a distance that more closely matches the effective tactical distance between two nodes in a game of Hex. The *bridge distance* between u and v for P is the minimum number of hexes that P needs to play so that a forcing connection using only bridges exists between u and v. Unfortunately, this definition of distance does not give a metric.

The bridge distance is a natural definition of distance, but its calculation is somewhat complicated due to the fact that it is not a metric. The fundamental problem comes from positions like that shown in Figure 3. In this figure the bridge distance from A2 to C1 is zero, as is the bridge distance from C1 to B3. However, there is not a forcing connection from A2 to B3. Thus, the bridge distance is one, not zero, and so transitivity is violated. The problem is that the node B2 is associated with both of the bridges, and so those bridges cannot be combined to get a connection from A2 to B3.

Hexane calculates the bridge distance between two nodes with the same breadth first search approach used for the literal distance. However, rather than just looking at all edges from a given node the search also considers nodes that are *bridge adjacent* to the node being processed. By bridge adjacent we mean a node that can potentially be reached via a bridge. The two nodes of the bridge need not be played for a bridge adjacency to exist,

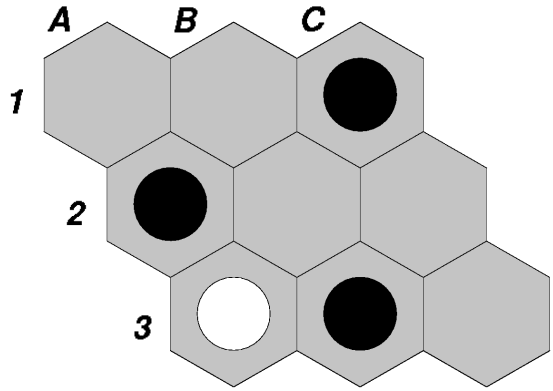

Figure 3: Bridge Distance is Not a Metric

though the bridge itself will not exist until they are played. The algorithm also disallows paths where bridges overlap, as in Figure 3. Of course, this algorithm requires some representation of bridge adjacencies in the graph. While calculating these bridge adjacencies is easy on a Hex board graph, Hexane also uses a simple (but efficient) brute force method to calculate all the bridge adjacencies on an arbitrary graph.

The *forcing distance* is a natural generalization of bridge distance. The forcing distance between u and v for P is the minimum number of hexes that P needs to play so that a forcing connection exists between u and v. We will rarely be able to calculate this value exactly. In fact, if we could calculate the forcing distance in polynomial time, we could solve Hex in polynomial time. This result holds even if we cannot directly calculate the set of nodes that P needs to play to establish the connection.

Suppose we have such an algorithm for calculating forcing distance. The first player to move, say P with goal nodes u and v, wins by the following strategy: At the start of the game both players have a forcing distance of 1 between their respective pairs of goal nodes, because we know the first player to move wins Hex (establishes a forcing connection between the appropriate pair of goal nodes). At the start of the game, P calculates the forcing distance between u and v in each position reached from the start of the game by a legal move. Since the forcing distance at the start of the game is 1, there must be some move that results in a position where P has forcing distance zero. Player P plays such a move, establishing the forcing connection. By playing with this strategy at every move, player P maintains the forcing connection and wins the game.

A *forcing path* between hexes u and v for a player P is a set of hexes S such that if P plays all of the hexes in S then u and v will be force connected. The length of the path is $|S|$. A bridge path is a forcing path where the forcing connection uses only bridges, and a literal path is a forcing path where all the forcing connections are in fact direct connections. We will usually be interested in shortest paths of these three types.

We need to extend the idea of associated sets to forcing paths. The set A of associated hexes of a forcing path is the set of associated hexes for the forcing connection that exists between u and v after all the hexes in S have been played. For a bridge shortest path, the set A is simply the union of all the hexes associated with any bridge used on the path after all the hexes

in S have been played. Thus, a path can be fully specified by the 4-tuple (u, v, S, A). For consistency, we define the associated set of a literal shortest path to be the empty set. In the following discussion, any reference to a path in a SSGV graph will mean either a literal, bridge, or forcing path depending on context.

We note that the a forcing path (u, v, S, A) exists if and only if there is a forcing $|S|$-connection between u and v. However, if there is a k-connection between u and v there may be multiple forcing paths of length k between u and v with different sets S and A.

3 An Overview of the Hexane Project

The Hexane program allows a user to play a game of Hex and during the game perform various computer-aided analysis tasks. The program allows for computer versus computer play, human versus human play, or human versus computer play. The players for each side can be changed dynamically during the game, so a person can set up a position playing as human versus human, and then actually play that position against the computer. Analysis options include distance calculations and automatic checking for virtual connections.

The program is written in a modular fashion and the user interface code has been separated from the main program classes in such a way that additional analysis and testing features can be added with relative ease. It is hoped that Hexane will thus provide a useful base of code for other people working on Hex playing programs.

Useful basic classes for the representation and manipulation of Hex boards and positions are also provided. Classes allow the representation of Hex boards, but also of arbitrary graphs. Most of the routines used in Hexane in fact work on arbitrary instances of the SSGV. In a game of Hex such graphs are almost always either Hex board graphs or subgraphs of Hex board graphs. These classes should facilitate the writing of many Hex related algorithms not related to my current approach to playing Hex.

Hexane's current algorithm for finding a good move in a Hex position has two main components: a searcher that is efficient at discovering forcing

connections, and a graph structure that helps organize these local forcing connections and allows the detection of overall connections through their combination.

We will first describe Hexane's searching capabilities.

4 Searching For Connections

Hexane uses a modified version of the minimax algorithm to search for forcing connections between groups on the Hex board. Before detailing the specifics of Hexane's algorithm we will describe the basic principle of the minimax game tree search algorithm.

4.1 The Minimax Algorithm

The minimax algorithm applies to any two-player zero-sum game of perfect information. The goal is to determine the outcome of the game from the current position assuming perfect play by both players. For example, given an arbitrary position at some point in a Hex game with black to move, we wish to determine whether or not black can win the game, and if so what move black can make in order to win. This question can be decided recursively: the current position is a win for black if and only if some successor position (position reached after a black move) is a win for black. The analogous rule allows us to decided whether or not a position where white has to move is a win for white.

The minimax algorithm uses this principle to evaluate a position. For Hex, we assign a numeric value of 1 for a black win, and -1 for a white win. We then propagate this information back up the game tree using the following recursive rule for the minimax value of a position p,

$$\text{minimax}(p) = \begin{cases} \text{value}(p) & \text{if } p \text{ is a completely filled board} \\ \max\{\text{minimax}(s) \mid s \text{ is a successor of } p\} & \text{if black to move in } p \\ \min\{\text{minimax}(s) \mid s \text{ is a successor of } p\} & \text{if white to move in } p \end{cases}$$

where value(p) is 1 if black has a winning path and -1 if white has a winning path. A move s is a successor of p if there is a move in position p that produces position s. It is infeasible to generate the complete game tree and hold it in memory; fortunately, this is not necessary, for we can compute the minimax value using a depth first search.

An example may prove useful. Figure 4 shows a portion of the minimax game tree for 2×2 hex. Black moves first, and tries to connect top to bottom. The root of the tree is the unplayed board, where black is to move. The next level shows all possible moves for black, and so on. We have labelled each move (edge in the game tree) uniquely with a letter to make the algorithm easier to describe. The minimax algorithm proceeds by a depth-first search of this tree, first examining the sequence of moves $a - c - i$ to arrive at the first leaf. Black has a top-bottom path at this node, so we label the node 1 for a black win. The search then looks at the next possible reply to the white move c, namely j. After j white must play u, and we arrive at position that

is won for white, so we label it -1. It's parent node (at $a-c-j$) has only one child, so -1 is also the value of that node. We have now examined all the children of the $a-c$ node. Since this node is black to play, we give it the value of the maximum of all it's children's values, namely 1. The algorithm continues and finds the values for all of the children of the position reached by move a. Since a is a white to move node, we pick the minimum value, which is -1. Thus, if black plays first in the upper left corner, we see that white wins via the move d. The algorithm then proceeds to black's second possible move, b, and after propagating values up this subtree determines that this move leads to a position of value 1. Thus, the value of the root node is the tree is 1, because black has a winning move, namely b.

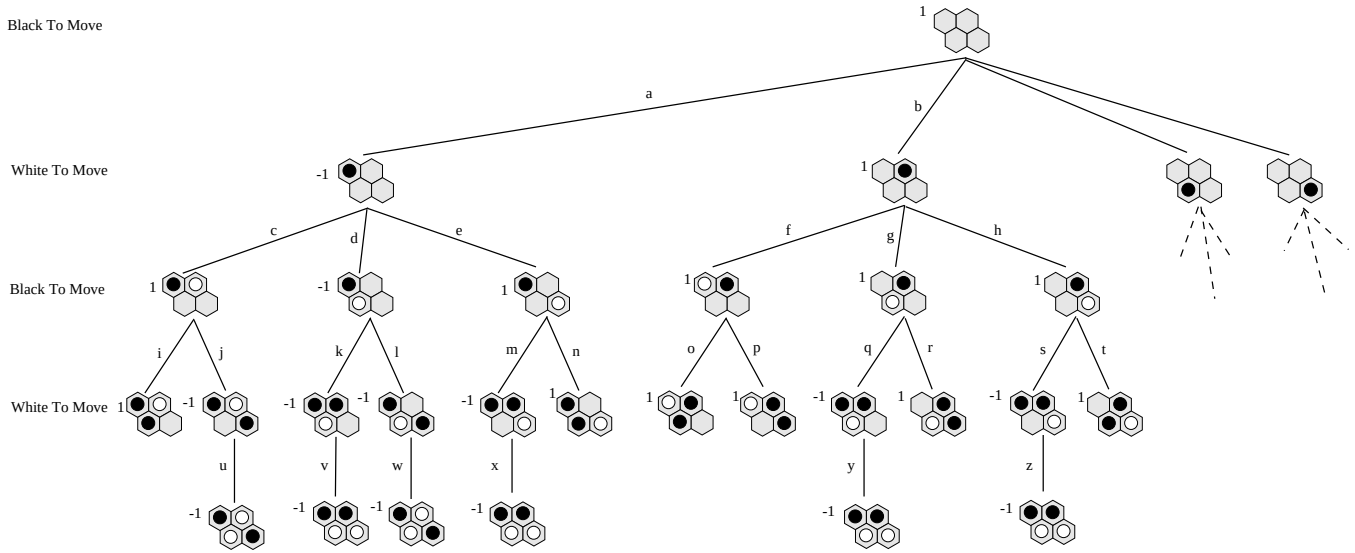

Figure 4: A Portion of the Minimax Game Tree for 2×2 Hex

For games such as chess and Othello, one cannot feasibly search the entire game tree, and so most programs search to a fixed depth and then determine a value for nodes at the maximum depth with a static evaluation function. For example, a simple static evaluation function in chess might consider which player is ahead in material, and by how much. The values given by the static evaluation function are then propagated back up the tree to the root node using the minimax rule. The idea is that the search between the current position and the application of the static evaluation will allow it to take into account forcing sequences, captures, combinations, etc. Most static evaluation functions will not return only 1 or -1. Instead, such a function usually returns a number that estimates the value of the position with a number from, say, -100 to 100. Many specialized searching and pruning algorithms have been developed for this search situation. The alpha-beta pruning algorithm is the most straightforward improvement to minimax available for game tree search with an evaluation function. For more information on alpha-beta and other pruning techniques, see [11] or [14]. For a survey of more recent (and more complex) solutions to this searching problem, see [7].

Hexane, however, does not use a static evaluation function. Instead, it deals with huge game trees by doing multiple searches that are restricted to subsets of the total board in an effort to find virtual connections. Hence, a position always has a value of 1 or -1; either the virtual connection exists, or it doesn't. Thus, a trivial pruning algorithm is available: suppose the minimax algorithm is examining all the possible moves for black in a given

position, trying to find a forcing connection for black. We don't really have to look at all of black's possible moves, however. We need only to find a single black move that leads to a connection (that is, a single child node with a value of 1). Similarly, at a node where white is to move we only need to find a single move for white that blocks the connection (a single child node with a value of -1). Thus, at each node we would like to order the possible successor moves so we check the best moves first. This approach should minimize the total number of nodes we need to search.

This simple pruning algorithm is in fact equivalent to using the alpha-beta algorithm with a search window of $(-1, 1)$, that is, where we set $\alpha = -1$ and $\beta = 1$ at the start of the algorithm. Thus, to search a minimax tree of depth d and constant branching factor b, we must search $\mathcal{O}(b^d)$ positions. With the above pruning implemented, we need to search only $\mathcal{O}(b^{d/2})$ nodes, and so can search approximately twice as deep as we could without the pruning. The result is similar for the decreasing branching factor for Hex. This result assumes a perfect move ordering, but if we had such a move ordering we would know the best move in any position, and wouldn't need to search at all. However, it turns out that even a rough move ordering will yield a speedup of the search that is close to optimal. The relevant analysis of minimax versus alpha-beta can be found in [11] or [14].

The actual algorithm used in Hexane is a variation of the negaMax minimax algorithm: rather than having one function for white's nodes where we pick the minimum valued successor, and one for black where we pick the

```
negaMax(pos, playerToMove)
     if(connected)
          if (playerToMove is playerToConnect)) return 1
          else return -1
     endif
     if(no successor moves)
          if(playerToMove is playerToConnect) return -1
          else return 1;
     endif
     for each successor position newpos
          value = -1*negaMax(newpos, otherPlayer)
          if(value == 1) return 1
     endfor
     return -1
end negaMax
```

Figure 5: Pseudo-Code for Hexane's Minimax Implementation

maximum, we have one function that always returns the value of the node from the point of view of the player to move. Our static evaluator evaluates from the point of view of the player to move. The pseudo-code for the algorithm is Figure 5.

Note that there are two base cases, and for each one we must check whether or not the playerToMove at the given node is the playerToConnect, because we always evaluate from the point of view of the playerToMove. When checking for connections, Hexane actually calculates the bridge distance between the two nodes in question, and returns true if this distance is zero. This detects connections earlier than only considering direct connections and hence significantly speeds the search. Also note that to get the

value of a particular successor we multiply the result of the call to negaMax by -1 to get the evaluation from the point of view of the playerToMove. The best we can hope for is to see a successor position of value 1, so as soon as we see such a position we return. If after looking at all the successor positions we have not returned then we must not have seen a position of value 1, so all successors must have all been value -1. Thus, we return -1.

4.2 Hexane's Minimax Implementation

The main negaMax searching algorithm used in Hexane is designed to answer questions about the existence of connections. Namely, given two groups of a given color, we wish to determine if there is a forcing connection between them. If there is no virtual connection, we can ask if we can form one by playing a single move (equivalently, is there a forced 1-connection). These two questions correspond to searching for a connection where the enemy plays first (the first case) and where our player plays first (the second case). We can easily generalize the algorithm to search for forcing k-connections.

If we had an algorithm that could test for the existence of an arbitrary forcing connection in a reasonable amount of time we could solve Hex by simply looking for virtual connections between our goal nodes. This is clearly not possible, so we are content to design an algorithm that is fast on search sets of reasonable size and that does not falsely report the existence of connections. Our algorithm, however, will fail to find certain virtual connections. That is,

the search is sound but not complete. This is similar to the approach taken in [1].

4.2.1 Iterative Widening

A simple minimax algorithm could be used to answer these questions. However, if we naively ask if a connection exists on a Hex board, looking at all possible moves for both players, we will need to search a game tree with a branching factor that is prohibitively large. The point of searching for connections between groups other than our goal nodes is to make the search feasible by reducing the branching factor at each level of the search. To accomplish this we adopt the following approach. Suppose we want to find a forcing connection between groups A and B. We first find a set of hexes on which a forcing connection is likely to exist (that is, a set that is likely to be the superset of the set of nodes associated with some connection between A and B). We restrict the allowed moves for searching to these hexes. That is, we search on some vertex-induced subgraph of the overall SSGV instance graph. It should be clear that when we restrict all moves to such a subgraph and we find a forcing connection in the subgraph the connection will be valid in the game graph.

There are two criteria influencing the selection of this search set: we want the set to be as small as possible so our search is fast (the size of the search set determines the branching factor). However, if we make our set too small and there is a forcing connection we may exclude some necessary

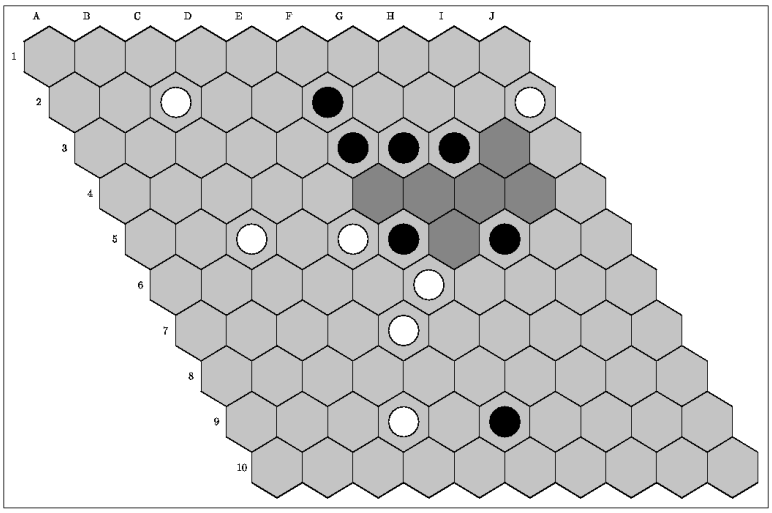

Figure 6: Nodes in a Search Set from H5 to the H3 Group

(associated) nodes from our search set, hence precluding the possibility of our search discovering the forcing connection.

We adopt an iterative widening of the search set to balance these two competing criteria. The algorithm first selects an initial search set that is as small as possible. The search is then performed; if we find a connection, then we are done. However, if we don't find a connection the cause may be that the desired connection does not exist or it may be that we excluded from the search set nodes that are need for a connection. Thus, if the first search fails we "widen" the search set by adding more nodes that we think might be needed by a connection. This process is iterated until a connection is found, or the search set becomes so large that it cannot be searched in a reasonable amount of time. We call this approach *iterative widening*, due to

the similarity to the iterative deepening search algorithm commonly applied in chess programs.

Suppose we are searching for a connection from some group A to some group B. Hexane's algorithm for selecting the initial search set is straightforward: it takes the union of the set of all nodes on some standard shortest path from A to B, the set of all nodes on a bridge shortest path from A to B, and the set of all nodes associated with some bridge shortest path from A to B. If the initial search set selected by this union is too large, we simply return immediately that no connection can be found. Otherwise, we perform the search. An example of the unplayed nodes in an initial search set is given in Figure 6.

When a search fails, Hexane widens the search set by adding to the search set all nodes adjacent to nodes in the current search set. This approach is naive, for some of these nodes are much more likely to be useful than others; either using better heuristics for selecting the nodes to add, or actually using information from the previous search to select the nodes to add might be profitable.

For the remainder of this section, we will be describing how the search functions on a fixed search set. A number of techniques are used to speed up the search. Some of these techniques are admissible, which means that using the technique cannot change the search outcome. However, we also use techniques that are inadmissible; that is, they may result in not detecting a forcing connection that exists on the search set. First, however, we describe

a way to modify minimax search so that we can also discover an associated set for any forcing connection found.

4.2.2 Determining the Associated Set

The goal of our searches is to detect forcing connections. However, it is not enough to simply know that a forcing connection exists. We need to know the associated set of the forcing connection. Among other things, knowing the associated nodes of our forcing connections allows us to determine whether two forcing connections can be combined, or whether they interfere (have intersecting associated sets). There may be many possible associated sets for a given connection. We need to make sure our search can generate a set that contains as a subset some associated set. Ideally this set is as small as possible. Certainly the last search set upon which we discovered the forcing connections contains an associated set. However, this search set probably includes many nodes not in the associated set. To get a smaller set, we modify our negaMax algorithm to keep track of all the nodes that we might need to play in preserving our forcing connection.

Suppose we are at a node where a forcing connection exists. If playerToConnect is to move, the moves that may need to be played to preserve the connection from this position are the winning successor move, and all the moves that may need to be played from that successor position. If the other player is to move (but the connection still exists), then the playerToConnect may need to play any move that might need to be played in any of

the successor positions. This rule for determining an associated set is made precise by the following recursive definition, where S is the set of successor positions to p, and x is a successor to p reached by move m such that the desired forcing connection exists in x.

$$\text{associated}(p) = \begin{cases} \emptyset & \text{if the connection exists in } p \\ \bigcup_{s \in S} \text{associated}(s) & \text{if otherPlayer to move in } p \\ \{m\} \cup \text{associated}(x) & \text{if playerToConnect to move in } p \end{cases}.$$

The associated nodes of the root node in a search can be calculated using this rule during the negaMax search. By always returning the associated nodes of whoever "wins" a given node we can modify the above rule so we calculate the set of moves that the winner of the search may need to play. If the connection exists, this set is the set of nodes associated with the connection. If no connection exists, then we get the set of nodes that the defensive player (other player) may need to play in preventing the connection.

4.2.3 Iterative Deepening

The most basic minimax algorithm always searches to a leaf of the game tree. We would like to do this, for it would guarantee that we find a connection if one exists on the given search set. However, if a connection exists the longest possible sequence of play to get the connection will usually not be too long.

For example, a reasonable guess is that no sequence of moves required to achieve the connection will be longer than twice the length of the shortest path from A to B. Thus, we usually specify a fixed maximum depth for our search; when the search reaches a node at this depth, if we do not have a connection we return a no connection result immediately — we assume any further moves will not lead to a connection. While this makes it possible that we miss some connections with deep lines, it greatly decreases the amount of searching done in cases where there is no forcing connection. Using a hard cut-off is also justified by looking at complete analyses of smaller board sizes. For example, if the first player plays perfectly, a game of 6×6 hex can be won in 14 moves — even though the longest possible game is 36 moves [13].

Rather than searching all the way to this maximum depth immediately, we first search to some shallower depths; that is, we use an iterative deepening approach, a technique long used in chess programs [11]. Our initial search depth is the minimum depth where a win can actually be detected. If the bridge distance from A to B is d, then we must search to a depth of at least $2d$ (where we assume the initial position is at depth 0), because the opponent plays first, so the first evaluation of a position where the player-to-connect has played d moves will be at depth $2d$. After searching to this depth, we then increment the depth of our search by one and search again. This process is then repeated until the maximum depth is reached.

Using iterative deepening search (versus only searching once to the maximum search depth) usually increases search performance when a forcing

connection actually exists, because connections are usually relatively short, and iterative deepening insures that our deepest search is to the minimum depth necessary to find the connection. However, if no connection exists, then the iterative deepening search requires more time, because portions of the search are repeated, with no change in the outcome — it would have been quicker just to search to the maximum depth on the first search. However, we can make up for some of this performance loss by using values from the transposition table from earlier searches to speed evaluation in later searches. We will speak more about transposition tables later.

We have already talked about two performance enhancing features of the search. First, as soon as we find a winning move, the search at that level terminates. This optimization does not affect the outcome of the search, it only makes it faster. Specifying a maximum search depth is an inadmissible optimization in that it may affect the outcome of the search by making it impossible to find forcing connections where some line is too deep. However, this technique provides a critical speed increase in the case where no connection exists.

4.2.4 Move Ordering Heuristics

We have already argued that our return-on-win pruning is accurate, and provides performance equivalent to alpha-beta. The increase in speed gained by this pruning depends on how soon in our list of successors the first winning move occurs (assuming there is one). Thus, to really benefit from this

speedup we want to order our successor moves so that moves likely to yield a win are searched first. Several move ordering heuristics are used by Hexane. Most of these heuristics are designed from the point of view of the player to connect, as they are based on looking at the shortest path between the two nodes to be connected. However, it is generally true that a move that is good for the player trying to connect is also good for the defensive player. There are certainly many cases where this approach may not give a perfect ordering, but as long as we are right a lot of the time, we will achieve the desired speedup.

The move ordering used by Hexane assigns each potential move a value, and then sorts the moves so that the move with the highest value is searched first, and so on. The factors used in determining the value of a hex in the search set for a search from A to B are listed below in order of importance.

1. If the hex in question is on a bridge shortest path from A to B we give it a higher rating.

2. It is generally a good idea to first play moves closer to the goal node (A or B) that has the lowest unplayed degree (has the fewest unplayed adjacent hexes in the search set). Thus, we give moves closer to this low-degree node a higher rating.

3. It is seldom useful to play a hex that is associated with a bridge that is already completed (both end nodes on the bridge have already been played). These nodes receive a lower rating.

This move ordering appears to be fairly good, but exact statistics have not yet been compiled.

4.2.5 Additional Pruning Techniques

Pruning is the general technique of deciding not to search certain nodes in the game tree. Pruning strategies can be either admissible or inadmissible. The return-on-win strategy is an admissible pruning technique, as is the more general alpha-beta technique. The fixed depth cutoff is an inadmissible pruning technique.

Another type of admissible pruning can be performed when the player to connect is at a distance one from achieving the connection. In this situation, the defensive player must play a move that is on every shortest path; otherwise the player to connect will be able to connect. Thus, we calculate this intersection at each applicable node. If the intersection is empty then we know that our player can connect and need do no more searching. Otherwise, the set of defensive moves that must be considered is given by the intersection.

We can perform another type of calculation to detect failed searches if we are searching to a fixed depth. Suppose at some point in the search we are $2m$ moves from terminating the search and it is the player to connect's turn to move. This player will get at most m moves from this position before we reach the maximum depth. If the distance for connection is greater than m, then our player has no hope of connection, and we can return -1 for the

player immediately. We call this optimization the depth-distance cutoff.

We cannot prune any possible defensive moves (moves for the player trying to prevent the connection) that might change the search outcome, for not considering such a move might be equivalent to overlooking a defensive sequence that prevents a connection. That is, inadmissible pruning of the opponents move might result in the "discovery" of forcing connections that are in fact not valid. We cannot allow this. However, since we are under no obligation to discover forcing connections (though we certainly want to discover them), we can inadmissibly prune some of the player to connect's moves. For example, it may be useful to search deeper but only check moves that we think are fairly likely to lead to a connection, rather than searching all possible moves for the player to connect to a shallower depth. The most obvious pruning strategy that falls into this category is to only consider moves for the player to connect that are on some shortest path. Hexane can be set to use this strategy, or to switch to this strategy once a certain depth is reached.

4.2.6 Transposition Tables

An examination of Figure 4 reveals that several positions appear multiple times. The basic minimax algorithm will search a position each time it occurs in the game tree, even if the same position has already been seen earlier in the tree where it was reached via a different move ordering. This situation is known as a transposition. Transposition tables attempt to reduce

such repeated computation by storing values for positions we evaluate. When these positions are seen again the values can be read immediately from the table instead of being completely recalculated.

A transposition table is thus a dictionary data structure, where the keys are positions and the values are the minimax valuations of the positions. Hexane's transposition table is implemented via a direct lookup table rather than a hash table.

Normally such an approach would use a prohibitively large amount of memory, but the memory needed for such a lookup table is actually competitive with more sophisticated implementations thanks to a nice combinatorial representation of positions. We will show how only two bits are needed per position in such a lookup table, while in a standard hash table there would be significantly more overhead (at least 3-4 bytes per position stored). The key is an effective representation of game positions at a fixed depth.

Hexane uses a transposition table in its minimax searcher for forced connections. Thus, our positions should not be thought of as full positions in a Hex game, but as positions on the graph induced by the search set of nodes.

Suppose we have a position where the sequence of moves

$$b_1, w_1, b_2, w_2, b_3, w_3, \cdots, b_k, w_k$$

has been played on a search graph G. Each w_i or b_i is the number of a node in the search graph. Suppose our search graph is of size n and nodes

are numbered from 1, so that for $1 \leq i \leq k$ we have $1 \leq w_i, b_i \leq n$. It should also be clear that no two entries in this sequence are equal. In the actual Hexane implementation, these moves are numbers of nodes in a local representation of the search set graph; the relationship between this graph and the Hex board is not known to the searcher. The following discussion will assume that both players have played the same number of moves, but the situation where one player has played one additional move is analogous.

A position is uniquely indicated by this sequence of moves, but there are of course multiple sequences of moves that reach the position (that is the point of the transposition table). The actual position can be uniquely defined by specifying the subset of the nodes of G that has been played and then giving the subset of this subset that has been played by white (or black, equivalently). It is this idea that leads to our representation. We show how to derive a pair of integers (p, w) that uniquely specifies a position; p will indicate the subset of G that has been played, and w will indicate the subset of this subset that has been played by white.

To calculate p, we consider the set $S = \{w_i, b_i \mid 1 \leq i \leq k\}$, that is, the set of all nodes on G that have been played. We then sort the elements of S to come up with a canonical move sequence. Let the elements of S be s_i for $1 \leq i \leq 2k$ such that $s_1 < s_2 < s_3 < \cdots < s_{2k}$. Denote this sequence by Q. This ordering can be viewed as a $2k$-combination, that is, one of the $\binom{n}{2k}$ $2k$-subsets of the nodes of G.

An ordering can be established on all such $2k$-combinations in such a way

that Q can be assigned a number in the ordered list of all $2k$-combinations. The most natural ordering to establish on such combinations (subsets) of a fixed length is a lexical ordering. For example, if we consider 3-combinations of the set $\{1, 2, 3, 4\}$, the natural ordering of the $\binom{4}{3} = 4$ 3-combinations is 123, 124, 134, 234. Note that each combination occurs exactly once in this list, and we can now assign a number to each combination: 123 is 1, 124 is 2, and so on. In Chapter 4 of [5], a technique for generating all r-combinations of the set $\{1, 2, \ldots, m\}$ in lexicographic order is developed. But more importantly for us, there is straightforward way to determine the number in the ordering of a combination, without generating the whole list of combinations. If our combination of $\{1, 2, \ldots, m\}$ is $A = a_1, a_2, \cdots, a_r$, with $a_1 < a_2 < \cdots < a_r$, then the number of the permutation in the lexicographic ordering is given by

$$\pi_m(A) = \binom{m}{r} - \binom{m - a_1}{r} - \binom{m - a_2}{r - 1} - \cdots - \binom{m - a_{r-1}}{2} - \binom{m - a_r}{1}.$$

A complete derivation of this formula can be found on page 112 of [5]. This formula gives a bijective function π_m from r-combinations of $\{1, 2, \ldots, m\}$ to the set $\{1, 2, \cdots, \binom{m}{r}\}$. Thus, we can use this function to define the value p for our pair (p, w) by letting $p = \pi_n(Q)$. We know that $\pi_n(Q)$ uniquely defines the position because π is a bijection. This is exactly the property we need.

We use the same procedure to determine the number w. However, we

now consider k subsets of the set $S = \{s_1, s_2, \ldots, s_{2k}\}$. We define a mapping ϕ from S to the set $\{1, 2, \ldots, 2k\}$ by $\phi(s_i) = i$. The set we wish to represent is $W = \{w_1, w_2, \ldots, w_k\}$, where we have re-chosen labels so that $w_1 < w_2 < \cdots < w_k$. Note that $\phi(W)$ gives us the corresponding subset of $\phi(S)$. Thus, the number w from (p, w) is given by $\pi_{2k}(\phi(W))$, where we assume $\phi(W)$ gives us the appropriate ordered sequence (rather than simply the set).

Suppose we wish to create a transposition table that stores positions at depth $2k$ in a search graph of n nodes. Then, we have $1 \leq p \leq \binom{n}{2k}$ and $1 \leq w \leq \binom{2k}{k}$. It follows that there are $N = \binom{n}{2k}\binom{2k}{k}$ possible positions at this depth, and we can assign each position a unique number by $(p-1)\binom{2k}{k} + w$. Our transposition table is then implemented by two arrays of bits of length N. The first bit for a given position stores whether or not a value for the position has been calculated and stored in the table. The second bit is valid only if the first bit is set, and when valid it is set to true if and only if the position is a forced connection for the player to connect. A multi-depth transposition table is easily implemented via a collection of these single-depth transposition tables. Hexane usually uses such transposition tables at depths 3, 4, and 5.

The unique number determined for each position should be an excellent hash function, and so it would be easy to adapt this technique to a more standard hash table approach. Such an implementation combined with an effective replacement scheme would be necessary to store values of positions at larger depths.

4.3 Empirical Evaluation of Hexane's Search Routines

The effectiveness of the various optimizations we have described can be assessed by measuring the performance of the search algorithm with different combinations of features enabled on a suite of test positions.

The suite of test positions we used consisted of 14 test instances. Each instance is specified by a position in a partially completed Hex game and a designated pair of nodes that have been played by some player. For each instance, the search algorithm checks for a forcing connection between the designated nodes. Forcing connections existed in 8 of the 14 instances, while no forcing connections were possible in the remaining 6 instances. The instances where forcing connections exist are termed positive instances, and the remainder are negative instances.

We tested 5 different combinations of the optimizations on this suite of test positions. The search was performed with no optimizations enabled, with only the distance-depth cutoff enabled, with only the transposition table enabled, with only the move-ordering enabled, and finally with all the optimizations enabled. For the first three parameter sets the move ordering was done by generating a pseudo-random number for the value of each possible successor move. Performance for these different cases is summarized in Figure 7.

Search Configuration	total		positive cases		negative cases	
	search time	nodes searched	avg. search time	avg. nodes searched	avg. search time	avg. nodes searched
No Optimizations	575,354	2,050,815	66,533	235,596	7,180	27,674
Transposition Table Only	516,314	1,897,431	62,226	227,875	3,083	12,404
Depth-Distance Pruning Only	489,384	1,683,122	58,828	201,884	3,125	11,340
Move Ordering Only	29,522	119,227	3,281	12,871	545	2,709
All Optimizations	8,287	30,601	915	3,345	160	639

Figure 7: Results of Search Algorithm Testing

The search algorithm was tested with five different levels of optimization on a suite of 14 test positions. All times are in milliseconds. The first two columns indicate the total time taken and number of nodes visited by the given set of search parameters on all 14 test instances. The following two columns give average values for time and number of nodes visited on the 8 test instances where connections were possible. The final two columns give average values for time and number of nodes visited for the 6 test instances where connections did not exist.

For each of the five configurations the search was to a fixed depth of 8 with no iterative deepening, and the maximum search set size was 18. Except for the depth cut-off, all optimizations used were admissible, and so as expected all searches found the same connections. With the given parameters the searches actually discovered 5 of the 8 connections; by using a larger search set and a deeper search, all of the connections could be discovered, but of course the time for the searches would also increase.

The optimizations discussed clearly had a dramatic effect on search time in the test, reducing the total search time and number of nodes visited by approximately a factor of 70. This factor is of course completely dependent on the search depth and search set size chosen. For deeper search depths the increase in performance will be even greater.

It is clear from the data that the move ordering was by far the most important of the optimizations. This makes sense as the move ordering should lead to an expected exponential speedup. However, when the transposition table and distance-depth cutoff were combined with the move-ordering, the number of nodes visited was reduced by an additional factor of 4, indicating that these were also worthwhile optimizations.

Examining the figure one sees that the average time spent on successful search is almost an order of magnitude greater than the time spent on a failed search. Some of this extreme difference is due to the fact that three of the six negative instances were never attempted by the search algorithm because these instances had initial search sets of size greater than 18. Thus,

these searches registered effectively zero time and zero nodes searched. The three negative instances that were search happened to be relatively trivial, and hence could be detected quickly.

One interesting fact can be determined from an examination of the averages for positive and negative search instances: the transposition table was generally more useful on failed searches compared with the depth-distance cutoff, which was comparatively more effective on successful searches. This is a rather counter-intuitive result, as one would expect the depth-distance cutoff to be most useful in failed searches.

The results for negative searches should be taken with a grain of salt because the data is based on three rather limited searches. A future goal is to automatically generate a much larger set of test instances during games being played by Hexane, and use these instances to run more extensive tests.

4.4 Using Connection Information

We have described how Hexane finds local connections. Now we turn to the task of using this information to "understand" a Hex position and to choose a good move.

The group graph is the fundamental representational structure used by Hexane to accomplish these goals. A group graph G is a graph whose nodes correspond to groups in some Hex game graph H (there are no nodes in G for unplayed hexes). All the groups in a particular group graph have the same

color. We will usually form two group graphs from a given Hex position, one where all the white groups are nodes (G_W), and one where all the black groups are nodes (G_B). We allow multiple edges between nodes.

The edges in the group graph represent information about (shortest) forcing paths between groups. If a forcing connection exists between two groups, we represent that connection via an edge in the group graph of the appropriate color. Since the edge corresponds to a connection we give it a length of zero and also store with the edge the set of associated hexes. Similarly, if we can play a single move and arrive at a forcing connection between two groups, we store an edge with length 1 together with the move to play and the associated hexes for the connection that will be formed. Between every two groups we can add an edge with length equal to the standard distance between the groups, and an empty associated set. Similarly, we can add edges for bridge distance and any forcing k-connections we have found. In general, each edge in the group graph can be thought of as a path in H. Thus, each edge e in G has a length d_e, a set of nodes on the path S_e, and a set of associated hexes A_e.

The group graph provides a convenient representation for the collection of connections found by the search algorithm. For example, connections can sometimes be combined to form connections between more distant groups between which a connection was not directly discovered.

There are several rules we can use to combine forcing 0- and 1-connections into "longer" forcing connections. Suppose we have a forcing connection

from A to B with associated set S_{AB} and a forcing connection from B to C on associated set S_{BC}. We can join these connections to create a forcing connection from A to C provided that $S_{AB} \cap S_{BC} = \emptyset$.

Suppose we have two forced 1-connections from group A to group B with associated sets S_1 and S_2. If $S_1 \cap S_2 = \emptyset$, then we can combine these two connections to form a forced 0-connection from A to B with associated set $S_1 \cup S_2$. More generally, if we have a set of k forcing 1-connections from A to B with associated sets S_1 through S_k, then we can combine them to form a forcing 0-connection from A to B on the set $\bigcup_{i=1}^{k} S_i$ provided that $\bigcap_{i=1}^{k} S_i = \emptyset$. This generalization was first described in [3]. In this paper, Anshelevich describes an automated theorem proving approach to combining small virtual connections (bridges and direct adjacencies) to build up a set of virtual connections on the Hex board. We are actually interested in using these rules to combine connections we have already found. Note that these rules generate new forcing connections that can then be used in later applications of the rules.

Hexane currently uses a brute force approach rather than the more elegant theorem-proving technique. The group graph is relatively small, so we can use search techniques to identify sequences of compatible forcing connections.

Suppose we are examining the group graph for player P, with goal nodes u and v (the goal nodes in the original SSGV graph are in some group, and these groups become the goal nodes in the group graph). We would like to estimate the forcing distance from u to v in H. We do this by finding a

shortest valid path from u to v in G. This technique allows the use of all edges in the group graph, whether they are shortest paths or forcing 0- or 1-connections.

First, we need to establish a formal definition for a valid path in a group graph. We establish our definition so that a valid path in G corresponds to a forcing path in H. A valid $u - v$ path in a group graph G is a set of edges E in G that form a $u - v$ path in G in the graph theoretic sense that also satisfies the condition that for any $y, z \in E$ we have $A_y \cap A_z = \emptyset$. That is, no pair of edges on the path conflict with each other in H. For a valid path T in the group graph G, we construct the overall set of nodes in H on the path, $S_T = \bigcup_{e \in T} S_e$, and the set of all nodes associated with the path $A_T = \bigcup_{e \in T} A_e$. Note that a valid path in G corresponds to a forcing path in H given by (u', v', S_T, A_T), where u' and v' are played nodes in H corresponding to the group nodes u and v in G. We can have forcing paths in H with no corresponding valid path in G, unless G contains an edge for every forcing connection in H.

Valid paths in G give upper bounds or estimates on the forcing distance between u and v. If we find a valid path with length zero, then we have a forced connection between our goals, and a valid path of length one indicates that if it is our move we can play to win. Otherwise, this forcing path gives us some idea of where we wish to focus our play.

To be precise, for white we construct a forcing path T_W on G_W between the white goal nodes. Similarly, we construct a path T_B on G_B between the

black goal nodes. The set

$$R = (S_{T_W} \cup A_{T_W}) \cap (S_{T_B} \cup A_{T_B})$$

is a set of hexes on H that hopefully corresponds to the region of the board where both players are likely to want to play. Playing a move from R will serve both an offensive and defensive purpose.

Finding valid paths on a group graph is a difficult problem. In fact, the generalized problem where the associated set of each path is chosen arbitrarily from some set is likely to be NP-complete. However, group graphs for Hex on a reasonably sized board are likely to be small, and so the exponential time of the algorithm described below is not prohibitive.

We would like to find all shortest valid $u - v$ paths. An obvious though naive way to approach this problem is to enumerate all valid $u - v$ paths, and pick the shortest ones. We use a modified recursive depth first search to examine all valid paths from u. The algorithm in pseudo-code is given in Figure 8.

The `buildPath` function is first called with a trivial path containing no edges and only the vertex u. Our path structure has a head and a tail. The head is fixed at the starting node, in this case u, and the tail is the last edge/node added. The algorithm works by building up valid paths from the head by adding edges.

The first `if` statement in the pseudo-code handles the base case where we

```
buildPath(Path P)
     crntNode = P.tailNode();
     if(crntNode == v)
          if(P.length == shortestPathLength)
               add clone of P to list of shortest paths
               return
          else if(P.length < shortestPathLength)
               clear list of shortest paths
               add clone of P to list of shortest paths
               shortestPathLength = P.length
          else return
          endif
     endif
     if(P.length > shortestPathLength) return
     for each edge e to node w from crntNode
          if w not on P and P.compatible(e)
               buildPath(P.addNewEdge(e))
               P.removeLastEdge()
          endif
     endfor
     return
end buildPath
```

Figure 8: Pseudo-Code for Finding Paths on a Group Graph

.

have reached v. If we have another shortest path (or an even shorter path) we modify our list of shortest paths appropriately. Otherwise, we simply return.

The second `if` statement is a simple form of pruning that speeds the search. We realize that if our path is already longer than the shortest path to v, then our path cannot possibly become a shortest path to v, so we abandon the path immediately.

We process each edge from the tail node. The function `P.compatible(e)` returns true if the edge `e` can be added to the path `P` without making `P` invalid. After checking an edge we remove the edge from the path and try another edge, until all edges have been tried. In the actual Hexane implementation edges are ordered so that shorter edges are searched first. This move ordering improves the effectiveness of the above mentioned pruning technique.

It should be clear that this algorithm traverses the group graph in a depth first manner and will find all shortest $u - v$ paths.

Hexane uses the above algorithm to find valid paths in G and then interprets these as forcing paths in H to help identify the active region of the board via the set R. There is a potential problem with this approach. Some edges in a group graph G_P may correspond to paths that player P has no hope of completing in the game. To account for this, we want to limit the edges in the group graph to paths that are in some way "feasible." Hexane currently is very aggressive in doing this, for it requires all paths in H that form edges in G to not use any hexes in a forcing path of $\hat{P}$. The idea is that

a path that crosses a forcing connection can never be in the game, because $\hat{P}$ can simply play to complete the connection and hence block the path. The approach taken by Hexane is too restrictive, in that it does not allow some valid paths. It would be useful to formalize the concept of a feasible path and to establish algorithms for finding such paths.

5 Results, Analysis, and Conclusions

Hexane plays Hex at the level of a relative beginner. This is mostly due to the fact that the majority of work went into the search algorithm. Currently, Hexane plays a random move selected from what it perceives as the active region of the board if it cannot directly block an opponents connection or make one of its own. To play more strongly Hexane needs another component that selects a good move once the active region of the board is identified. However, in forcing situations Hexane is good at playing the required defensive move. Similarly, the program is quite good at detecting forced wins; even though it plays at the level of a beginner it often realizes it has lost far before its opponent does. This, in fact, presents another challenge. Currently the program effectively gives up and plays relatively random defensive moves once it realizes it has lost. However, it should try to play in a way that maximizes the number of moves before the loss, thus forcing the opponent to play the more difficult lines.

Some of the techniques used in Hexane's search algorithm might also be useful in the creation of endgame evaluators or tactical search engines for games such as Go and chess. However, many of the techniques seem uniquely suited to Hex.

References

[1] L. V. Allis, H.J. Van Den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12(1):7–23, 1996.

[2] Steve Alpern and Anatole Beck. Hex games and twist maps on the annulus. *The American Mathematical Monthly*, 98(9):803–811, 1991.

[3] Vadim Anshelevich. The game of hex: An automatic theorem proving approach to game programming. Will appear in AAAI-2000 proceedings.

[4] Anatole Beck, Michael N. Bleicher, and Donald W. Crowe. *Excursions into Mathematics*. Worth Publishers, Inc., 1969.

[5] Richard A. Brualdi. *Introductory Combinatorics*. Prentice Hall, 2nd edition, 1992.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[7] Arie de Bruing and Wim Pijls. Trends in game tree search. In Keith G. Jeffery, Jaroslav Krl, and Miroslav Bartosek, editors, *SOFSEM'96 : Theory and Practice of Informatics : 23rd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 255–274, 1996.

[8] David Gale. The game of hex and the Brouwer fixed-point theorem. *The American Mathematical Monthly*, 86(10):818–827, 1979.

[9] Martin Gardner. *The Scientific American book of mathematical puzzles & diversions*. Simon and Schuster, 1959.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[11] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[12] Jack van Rijswijck. Are bees better than fruitflies: Experiments with a hex playing program. Will appear in AI'2000: The Thirteenth Canadian Conference on Artificial Intelligence.

[13] Jack van Rijswijck. Hex opening theory web page (generated by queenbee). http://www.cs.ualberta.ca/~queenbee/openings.html, September 1999.

[14] Patrick Henry Winston. *Artificial intelligence.* Addison-Wesley, 2nd edition, 1984.