A Program to Play Havannah

Terry Rogers

BSc Computer Science - Mathematics
2003/2004

Summary

This project concerns the board game Havannah. A long standing problem is to develop a computer implementation of Havannah that is capable of beating a human player.

The aims of this project are to develop some simple strategies for the game and implement them within an existing system for the game developed over the past year. And to evaluate the quality (within the scope of this project) of the strategies (ease of implementation and time complexity as well as quality of play).

Implementations of other board games should also be considered, though it is known that techniques used in these games tend to fail when applied to Havannah. Consideration is also given to why Havannah is such a computationally complex problem, particularly in comparison with other board games.

Acknowledgements

I would like to thank

- Haiko Muller, my project supervisor for his help and advice
- Johannes Waldman, for the excellent Havannah server system
- Jörg Endrullis, for the Havannah applet
- Chris Freeling, for the game Havannah

Contents

Chapter 1

Introduction

1.1 The Game Havannah

Havannah is a strategy board game. Mindsports [1] gives the rules:

- Two players (white, black) play Havannah on a hexagonal grid as depicted below.
- Two cells are adjacent if they share an edge.
- The game starts on an empty board. Players move in turn to place one stone on an empty cell. White moves first.
- The game is won by the first player to complete a ring or a bridge or a fork.
- All of these are chains: closed connections of one colour.
  - A ring is a chain around at least one cell (empty or occupied by a stone of either colour).
  - A bridge is a chain linking two corners.
  - A fork is a chain linking three sides. Corners do not belong to sides.
- A draw occurs when all hexagons on the board are occupied and no player has reached a victory condition (this is very rare in practice).

A detailed move history of a test game is given in appendix B.

Board Size

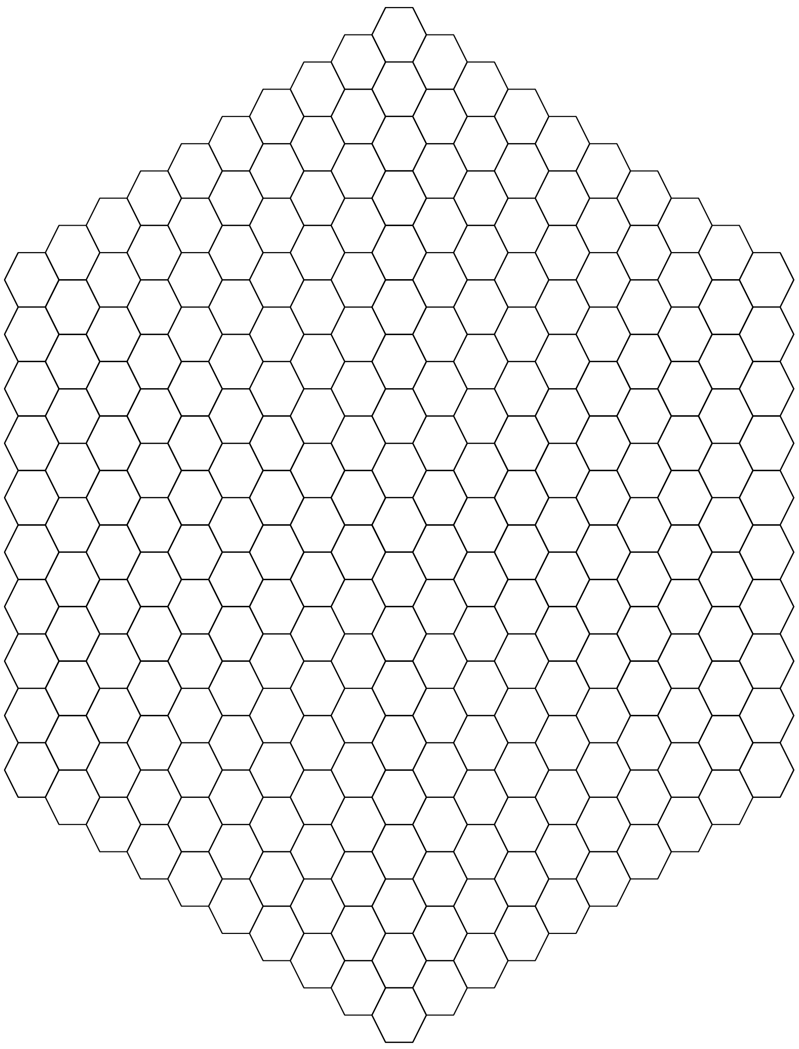
For development and testing purposes Havannah is often played on differing size boards, typically the size of the board is referred to by the number of hexagons along each side, with board size 10 (and sometimes size 8) being the most common version played.

Numbering

For implementation and reference purposes it is helpful to number the hexagons on the Havannah board, this is done by letting the bottom left hand corner be A1, then letters increment along each north-east diagonal (so left-to-right) and numbers along each south-west diagonal (so bottom-to-top).

This can be seen on the example game in appendix B.

Figure 1.1: The Havannah Board



1.2 The Project

The aims of this project are the development and implementation of one or more strategies for Havannah.

The agreed minimum requirements are:

- Review literature on development of computer implementations of board games
- Develop a strategy for the game, and how a computer could be taught to play
- Develop a computer system that plays legal Havannah and recognises victory conditions
- Compare this program with other computer implementations of the game and evaluate performance

Scope of the Project

This project will not produce an implementation that can consistently beat a human player. That is a long standing development aim, with a €1,000 prize attached to it. This project instead intends to develop strategies that are possible to implement in the existing system (§1.3) and then evaluate their performance, in terms of both time complexity and quality of play.

Possible Extensions

- Develop more than one possible strategy, and attempt to develop strategies that can compete with existing implementations.
- Discuss the reasons for the failure of computers to compete with humans at Havannah, in comparison with games such as chess.

1.3 The Havannah Server

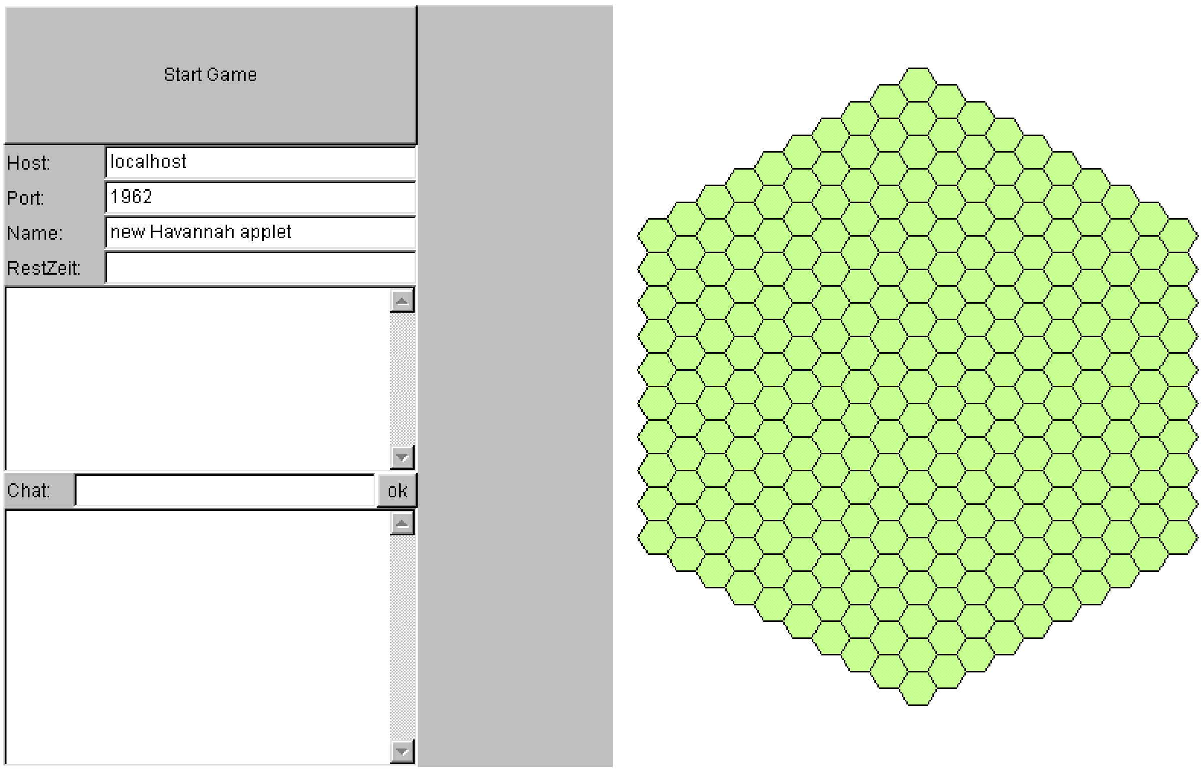
Last year a programming competition was run at the University of Leipzig by Professor Johannes Waldman to develop a Havannah system. Although no implementations were produced by the students a standard Havannah server was developed [2] (following systems developed for other games), and a “test player” was developed.

The server is written in Haskell (a purely functional programming language). The user plays against the server through a Java applet on a web page at
`http://141.57.11.163/havannah/different-applet/`

I shall implement any strategies I develop within this standard system.

The server works by waiting for connections on a specified port (several ports can be open, offering different games and/or board sizes). An implemented strategy can then connect to a server port and wait for another implementation or the Havannah applet to connect to the port. Then the game can be played.

The ‘test player’ or ‘example strategy’ developed essentially builds randomly a group of stones in a small area, to attempt to build a ring.



Host: localhost
Port: 1962
Name: new Havannah applet
RestZeit:
Chat:

Figure 1.2: The Havannah Applet by Jörg Endrullis

Chapter 2

Computer Games

2.1 Introduction

There is a long history of implementing board games on computers, and a lot of work has been done in this area. Chess has been the focus of large scale projects such as the Deep Blue[1] project [3, 4] in 1997. This is also an area where people judge the relative intelligence of computers and humans; along with the Turing test[2] for natural language development a chess computer that can consistently beat human Grand Masters is a key development aim for artificial intelligence.

Computer games are also a common and interesting area for research and development projects at all academic levels. Huang [5, 6] suggests several different areas of projects for strategy games and for the game of Go, another game like Havannah that has not been successfully implemented.

2.2 Minimax Game Algorithm

There is a standard method of solving simple two player games where the "complete game tree" is known. The game tree is a tree starting with a root representing the game's initial state (e.g. an empty board) and branches from each node representing all possible moves to new positions, with the leaves representing the end of the game (victory or draw).

We assign one player to be the maximizer and the other to be the minimizer.

Bruin, Pijls & Plaat [7] discuss the method in more detail.

To build the complete game tree first search down to the leaves and assign 1 to a leaf with a maximizer victory, -1 to a leaf with a minimizer victory and 0 to a leaf with a draw.

[1]See also official Deep Blue project site at `http://www.research.ibm.com/deepblue`

[2]For more information on the Turing test see `http://cogsci.ucsd.edu/~asaygin/tt/ttest.html`

Then apply the following function to get the the minimax values for the other nodes:

$$f(n) = \begin{cases} max\{f(c) | \text{ c is a child of n}\} & \text{if n is a max node} \\ min\{f(c) | \text{ c is a child of n}\} & \text{if n is a min node} \end{cases}$$

(a node is max [min] if it represents a move by the maximizer [minimizer]) [7]

Jones & Thuente [8] shows that the strategy for a perfect player for the maximizer is:

```
At my move
IF there is a node with a $1$, choose it.
ELSE IF there is a node labeled with a $0$, choose a random $0$ node.
ELSE choose any random node (all of whose labels are $-1$)
```

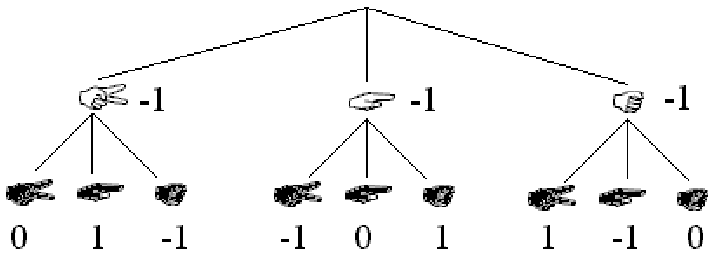


Figure 2.1: A Complete Game Tree

Figure 2.1 shows the complete game tree for “Scissors, Paper, Stone”, which in this case is a second player victory, as we have assumed that the second player will see the first players move before playing. This is merely to illustrate how to construct the tree with a very simple game. If white is the maximizer and black the minimizer then we can assign a value to each of the leaves as to whether it represents a white victory, black victory or draw (scissors beat paper, paper beats stone, stone beats scissors, draw if both have the same). Then for each move above the leaves (in this case there is only one level) we apply the minimax algorithm as described, the internal nodes represent a black move, so we minimize the values below to get -1 in each case.

2.3 Tic Tac Toe and Connect 4

Jones & Thuente [8] discusses the development of a computer system to play the commercial game of connect four.

They approach this through the simpler games of tic tac toe and connect three, and consider the game on different size boards. By analysing the outcomes they were able to show that tic tac toe was a “forced draw”, i.e. if both players play according to a perfect strategy the result will be a draw. The perfect strategy for tic tac toe is also unbeatable (a perfect player will only win or draw, never lose).

Connect three is similar to tic tac toe, but the size of the board is variable. Jones & Thuente observed that the outcomes of perfect strategies depended on the board size, e.g. “first player win” for boards 3*4, 3*5, “forced draw” for boards 3*3, 4*3.

Full game trees for tic tac toe and connect three are not difficult (for a computer) to construct. In connect three and tic tac toe Jones & Thuente were able to come up with simple strategies to show the “first player win” and “forced draw” situations (depending on the opponent’s moves).

For connect four the complete game tree is much larger, so computing the complete game tree becomes impractical

> “The game tree has a branching factor of about seven and up to 42 levels” [8].

So they use methods to reduce the game tree and estimate the outcomes. By continually improving their method they could accurately predict the winner of a game within five moves of completion in around 80% of cases.

2.4 Chess

Chess has been implemented many times on computer with commercial games becoming available as early as 1979[3]. In 1996 and 1997 a major chess project was held between the ‘Deep Blue’ computer and the then reigning Grand master Gary Kasparov. The computer managed a victory of $3\frac{1}{2}$ to $2\frac{1}{2}$ in 1997 [3, 4]. This was achieved through an analysis of many thousand of previous games played by the world’s foremost chess players, particularly including Kasparov himself. This analysis was developed into an ‘opening book’, detailing the analysis of the opening section of the game (as had been used in many computer implementations before), and an ‘extended book’ specifying moves later in game in board layouts where there is an obvious advantage to making a specific move (e.g. moving a queen out of danger). Using this data and complete analysis of several moves down the line from the current position the computer can play a very good game. There is however still a very inhuman element to any computer players of chess [4].

[3]My Chess, see `http://www.the-underdogs.org/game.php?id=2276`

2.5 Havannah

In many games it is relatively easy to detect how well a player is doing. For example the length of a connection in Connect 4, the number of pieces remaining in chess or the amount of territory under control in Go. For Havannah this is not so simple. The aim for a player must be to develop a frame and then to complete it into one of the victory conditions. However it is not a trivial matter even to tell when a frame is formed, even experienced players do not always agree on this [1]. How a frame can be developed is then a complex problem, and not going to be simple to form algorithmically. Even more complex would be how to detect a potential frame being formed by the opponent.

2.6 Quality of Literature

Many of the documents referred to in this project are published journal articles [3, 5, 6, 7, 8]. These have all been examined by several experts in their field, and so can be trusted as high quality sources.

The Mindsports website [1] has not been subject to such reviews, it is however written by people who know a lot about the game and other similar games that can be played through their website. Information taken from here should be considered more carefully, and not taken on trust, some elements may be simply the view of the authors.

The other website referred to, on the subject of chess [4] was written by Timothy McGrew a member of staff at the Department of Philosophy in the Western Michigan University (USA). The document is hosted by the university and so we can trust the source to know enough about the development of computer chess, and it's limitations, for our purposes.

Chapter 3

Strategies

3.1 Introduction

Strategies for Havannah are not so well developed than strategies for games such as chess. In chess both human and computer players take great note of the exact moves made in previous victorious games, in many ways this is impractical for Havannah. One strategy would be simply to work out every possible move for each player, §3.2 shows the impracticality of this.

Some detail is known on tactics for Havannah from human players, §3.3 discusses the detail provided from Mindsports [1], but this is aimed at humans, and is not necessarily easy (or indeed possible) to implement algorithmically).

Beyond this strategies need to be developed that can be implemented, §3.4 and §3.5 give two such strategies.

3.2 The Full Havannah Game Tree

The most complete strategy for this and any similar game is to follow the minimax algorithm as discussed in §2.2. However this requires that the complete game tree be found.

Clearly the complete game tree for Havannah is going to be very large. Havannah has no illegal moves (beyond playing in already occupied cells), so each player can play at any unoccupied position on the board. This means that the game tree will have a factorial branching factor. Even a small board with side three has 19 positions, so a tree of size $19! \approx 1.22 \times 10^{17}$, which is impractical to work with on a computer, requiring at least 10^5 Tera-bytes of memory!

The number of hexagons on a board of size b is given by $f(b) = 3b^2 - 3b + 1$, so the full board of size 10 has 271 hexagons, and a complete game tree around 10^{543} nodes.

3.3 Known Havannah Strategies

3.3.1 Safety vs Speed

Mindsports [1] discusses the “safety vs speed” issue, i.e. is it better to place stones in “safe” configurations, so your opponent cannot block your strategy, or to build your bridge(s), fork(s) and ring(s) quickly, largely ignoring the opponent’s moves.

3.3.2 Frames

An important issue discussed is the development of a “frame”. From [1]:

A frame is a connection aiming at a ring, bridge or fork, that, though still incomplete, cannot be broken by the opponent. The last property is essential. With safety taken care of, it gives rise to two simple strategic truths:

- Attacking a frame pushes it right into victory! The only defence is: having a faster frame, or at least threatening a faster connection in the process of making one.

  This may seem like ‘kicking in an open door’, but even experienced players do not always recognize a frame as such, until they discover that their attempts to cut or block were in fact counterproductive and would better have been left undone. In short: only defend if it can be defended.

- Balanced games, that is: games that are not decided by tactical oversight, will eventually take the character of a race. Assuming that both players eventually frame (the alternative would be tactical oversight), both will engage in the difficult process of counting. The faster player will make it a race without bothering about the opponent other than to answer local tactical threats.

3.3.3 Mindsports Strategies

[1] offers advice to human players of the game and outlines several strategic methods for consideration. The strategies discussed there are aimed at human players, and some of the issues are hard to formalise into any algorithmic method, so hard to implement in a computer player.

3.4 “Straight Fork/Bridge”

A strategy that can be implemented is to start in a corner (chosen at random) connect up to the neighbouring two sides. Then if a chain of stones can head away from this corner whenever it reaches another corner or side hexagon (baring the same sides as we started on) will create either a fork or a bridge.

The chain can follow a direction generally away from the initial corner, and will need to deflect around existing stones.

This strategy takes no account of the opponents moves, so is very easy to beat, it is however a very safe strategy, if the opponent is slow at forming and completing their winning chain then it stands a chance of winning.

It is also clear very quickly to the opponent what strategy is being used, so they can easily adopt a strategy to beat it.

3.5 “Straight Fork/Bridge with Defence”

An improvement to the above strategy would be to implement some defence into it, instead of looking immediately to extend the main chain a check could first be made as to whether the opponent is nearing (one move away from) victory.

Chapter 4

Implementation

4.1 Introduction

As discussed before work has been done to develop a Havannah server (§1.3). The strategies discussed have been implemented within this server.

The server is written in Haskell, a purely functional programming language. Strategies are added by writing a function taking a Havannah board state (object of type 'Havannah') together with a memory state and return a move ('Satz'). A move is normally 'Put' followed by a Point object representing a hexagon on the board, though it could be 'Pass' or 'Resign'.

Running the Software

The server is run using the Glasgow Haskell Compiler, it simplifies updating the software if it is run through the interpreter rather than being statically compiled, so scripts are included with the code to start the '`gchi`' program with the required packages. The player itself also needs to be run separately (scripts are provided for the purpose).

The server when run reads from a '`Havannah.conf`' file which specifies which ports should be opened on the server for players to connect to, the port should be specified with a Havannah board of a specified size (the server also allows for connections for different games). Note that only boards of size 10 can be connected to the Havannah applet for visualisation against a human player. Each player has a `main` function which will dial the server (on a specified port) and wait for a human player to arrive and begin a game, a `loop` function is also provided which will automatically redial when a game is complete, this is normally what we want so humans can play repeatedly against the same computer player through the specified port.

Programming Language

The decision to use Haskell as the programming language was to some extent pre-decided by the existing implementation from Leipzig [2]. It would be more complicated but not impossible to develop a player in another language, almost any language can connect to the port and understand the protocol used.

There are advantages to using Haskell, it is a very efficient language and it's 'lazy' evaluation means that we can implement strategies that only evaluate some values if they are needed, relatively simply.

Interpreters and compilers for Haskell are also easily available. The Glasgow Haskell Compiler was used for this project. The development of the server software was done for 'GHC', so it works best with this. Note that only the Linux implementation of GHC provides the 'posix' library required for the server port communication. Windows implementations of port communication exist, but there would have to be significant reprogramming of the network interaction.

Haskell is also a programming language I am familiar, through doing the 'functional programming' module at the school this year.

The main advantage to using Haskell though is that concepts such as the Havannah board, and point references have already been implemented into the system, allowing a great deal of code reuse.

Design

It was inappropriate to use a fully detailed methodology system for this project. Much of the system design was done with the existing code [2]. I did not have access to any design documents for this code, only the final code itself.

The design complexity of the code developed was relatively simple, and worked by simply plugging in new functions to the existing code.

4.2 Full Game Tree Searching

First a data type must be defined for the tree, which is defined as a branch with a root and a list of sub trees. The root consists of a Havannah board - representing state at that point in the tree, a pair of integers - representing the move made to get from the parent node to this, and an integer - representing the minimax value of -1, 0 or 1 (as in §2.2).

The most complex part of this strategy (computationally at least) is the building up of the full game tree, this is done when it is first called upon, i.e. on the computer's first move.

To build the tree we start with an empty board and recursively make every possible move, checking each time for white or black victory. When a victory is found, that node is a leaf of the tree (has no sub trees), and is assigned a minimax value of 1 for white (the maximizer) or -1 for black (the minimizer).

Once the tree has been built fully the minimax values can be calculated for the remaining nodes (through back track in the recursion). Any node with no sub trees that has not already been allocated a value by the victory conditions must be a draw situation so we assign 0 to it. For the other nodes we should take either the maximum (white move) or minimum (black move) of the minimax values of the sub trees. A boolean is used to keep track of whose move is represented by each depth in the tree.

Once the complete game tree has been calculated it can be stored in memory (assuming it can be), and at each move the top of the tree can be removed.

Then at each move the computer makes, it uses a breath first search of the tree currently stored, to find the current board situation (we expect it to be only one level down the tree so breath first search will be considerably more efficient than depth first). Then we get the tree headed by the current board, from this first we look for any children with our minimax value (1 if white, -1 if black), if we find one we use the move stored in that child node (the first such node we find), if there are none, we look for children with minimax value of 0, and choose the first of those we find, if there are no 0's then we must chose a node with the minimax value against us. (There must be at least one possible move to make).

Finally we return the tree rooted at the new board state after our move (to be stored in memory), and the move made to get there as our move.

4.3 “Straight Fork/Bridge Strategy”

To implement this strategy is helpful to label each corner from 0 to 5, starting with 0 in the bottom left hand corner and numbering anti-clockwise.

Then the direction away from each corner can be referred to by the numbering of the corners.

The memory data will store the current direction, the unused directions(we will need to know when all corners have been exhausted) , the number of this moves (for initialisation) and the history of our moves (to allow back-tracking).

Direction dependent functions can then be defined for movement and to give the co-ordinates of the corner hexagons. For flexibility these are defined in terms of a `boardsize` variable, so the game can be tested on boards of differing size:

```
initialise :: Int -> (Int,Int)
initialise dir = case dir of
  0 -> (1,1)                               -- bottom left
  1 -> (boardsize,1)                       -- bottom middle
  2 -> (2*boardsize-1,boardsize)           -- bottom right
  3 -> (2*boardsize-1,2*boardsize-1)       -- top right
  4 -> (boardsize,2*boardsize-1)           -- top middle
  5 -> (1,boardsize)                       -- top left

move1 :: (Int,Int) -> Int -> Point
move1 (x,y) dir = case dir of
  0 -> mkPoint(x+1,y+1)
  1 -> mkPoint(x,y+1)
  2 -> mkPoint(x-1,y)
  3 -> mkPoint(x-1,y-1)
  4 -> mkPoint(x,y-1)
  5 -> mkPoint(x+1,y)
```

First we randomly order the six possible corners giving the first as the initial direction and the others as the unused directions.

The first three moves are fairly simple, but we must check that the opponent has not already taken any of the three initial points. If the opponent is occupying any of these hexagons we must start again and use the next direction available (from the list of remaining directions). If there are no remaining directions the strategy does not know what to do, and so we must resign.

After the three corner stones have been placed we want to head off in a direction away from the corner. If possible we want to go in the position given by the `move1` function with the current direction, we go there unless there is already a stone there (of any colour). If the hexagon is occupied we must try to move in another direction, this is done by adding or subtracting 1, 2 or 3 from the direction, modulo 6 (as there are 6 directions in total). This means we make a deflection around stones already there:

```
point1  = move1 (x,y)   dirn                -- straight on, first choice
point2a = move1 (x,y) ((dirn+1+6) `mod` 6)  -- deflecting by 1
point2b = move1 (x,y) ((dirn−1+6) `mod` 6)  -- and the other way by 1
point3a = move1 (x,y) ((dirn+2+6) `mod` 6)  -- by 2
point3b = move1 (x,y) ((dirn−2+6) `mod` 6)  -- by 2
point4  = move1 (x,y) ((dirn+3+6) `mod` 6)  -- going backwards
```

If all these points specified are occupied we must back track to our previous move and try the process again to make a move from that point (back tracking again if necessary).

If at any move (after the initial 3) we find that our previous move has reached a hexagon on the side, and we have not won, then we must be in the situation of being forced to backtrack so far as to return to one of the sides we intended to push away from, in this case it is best to assume that we cannot win in that direction and to start again from the next available direction.

When we check whether hexagons are occupied we do not distinguish between our own stones and enemy stones, it may be possible if there is one of our stones blocking the way to skip through it and continue the sequence on the other side. But if we attempt to do this it will cause problems when back tracking, as we will simply return forwards to the stone already exhausted.

4.4 “Straight Fork/Bridge with Defence”

To implement a defensive strategy I have chosen to check from the position that the opponent last played (which is relatively easy to extract given the way Havannah is implemented in the server). From this we first wish to calculate which points are connected to that move directly (in the same chain), and then which are only one move away from being connected.

Distance in the Havannah board is not as simple as distance would be from the two dimensional co-ordinates used to specify position, B2 and C3 are adjacent, but B2 and A3 are not. We are only interested in whether distance is 0, 1, 2 or greater so the (computationally) simplest method is simply to define a distance function by cases.

Then we can split up the set of points connected and 2-connected (connected except for a gap of one square) to the opponent’s last move. This is done through the `checkConnection` function, which recursively modifies three lists of points: those connected that have had their neighbourhood checked, those connected that have not been checked and those unconnected (to checked points at least); through a distance tolerance (1 or 2). The function then works through the list of unchecked connected points, the `checkThis` sub-function checks each of the currently unconnected points to see if the distance between them is less than or equal to the specified tolerance. This gives a new set of connected points, to add on to the unchecked points, and a replacement set of the remaining unconnected points.

Once these sets of connected points are known we can detect if there are potential bridges and forks (rings have to be considered separately). We will only try to find a blocking point for a bridge or fork if the current chain already connects 1 corner (bridge) or 2 sides (fork). To do this we have to check the number of different corners and sides are 2-connected, using the `isCorner`, `isSide` and `unique` functions to extract a list of single direction numbers for the corners and sides. If there are more than one corner or two sides connected then we search for a blocking point.

The `findBlock` function then looks at each of the connected points and by considering around the six directions (`checkbyDir`) it tries to find any points that neighbour both a connected and an unconnected point (by moving one move from each connected point, then checking each unconnected point to find one distance 1 away.

If a point is found first we must check it’s not already been played by us (otherwise we will choose it again next time and infinitely loop). If `checkTaken` says it’s not taken we return that point as the result. If we find no such point we return ‘`Nothing`’.

Note that if a chain is about to complete a fork or a bridge with the final move being the side or corner move then this strategy is unable to block it, there are no unconnected stones to block between.

For rings we need to look around the connected points, for any points that are two away that are not immediately connected. This is done by checking (`checkRound`) each pair of connected points and examining those that are two away. Then we calculate the `midpoint`, i.e. (one of) the hexagons immediately between the two points. If the `midpoint` is not in the connected points, and has not already been taken by us then we should move in that position.

Note that this is not exactly ring blocking, it does not check that there is at least one hexagon around which the ring is formed, so it will block small cyclical chains which do not form rings, however this is often a useful tactic, and helps block some more complex bridge and fork structures.

Each of the ‘maybe’ type constants `bridgeBlock`, `forkBlock` and `ringBlock` are then either ‘`Nothing`’, indicating that no blocking needs to be done, or give a point at which we should play to block the opponent.

The key part of the movement function can then be modified to first check the bridge condition, then the fork and the ring, and only continuing movement as before if all three result in ‘`Nothing`’

```
(theMove,newMoveNo,newDirn,newDirs,info)
 = maybe (maybe (maybe (nextMove dir moveNo dirs) -- normal move
    (\r -> (mkPoint r,moveNo,dir,dirs,
      "Defending - Blocking Ring")) ringBlock)
    (\f -> (mkPoint f,moveNo,dir,dirs,
      "Defending - Blocking Fork")) forkBlock)
    (\b -> (mkPoint b,moveNo,dir,dirs,
      "Defending - Blocking Bridge")) bridgeBlock
```

Chapter 5

Evaluation

5.1 Introduction

What makes a good Havannah strategy? The obvious answer is one that wins consistently, however as discussed earlier this is impractical to implement on a computer. Considering the scope of this project I will judge the strategies on a number of criteria:

1. Is the strategy easy to describe in an algorithmic fashion?

   We are interested only in strategies that can be converted into computer program instructions. Strategies can be described which are impossible to implement on a computer within a suitable time complexity.

2. How fast can an implementation run?

   The server has a strict time limit for moves to be made, is the strategy likely to run out of time while 'thinking'? Will all moves take roughly the same time or will early or later moves take longer?

3. Does the strategy make some attempt to win?

   i.e. does it attack, a strategy that is entirely defensive is unlikely ever to win, as it's moves are likely to become very fragmented in comparison with the opponent.

4. Does the strategy make some attempt at defence?

   If no attempt is made at defence then any opponent can build a very fast winning configuration that is very unsafe (possibly in as few as 6 moves).

5. Can the strategy beat other implemented strategies?

   Is it better or worse than other known strategies. How does it perform against itself?

6. How easy is it for the opponent to ‘spot’ the strategy?

   Is it easy for a player (human not computer) to work out the strategy the computer is using. If it is very easy to spot then the player can quickly learn how to beat it.

5.2 Testing

Formal test plans are inappropriate to these implemented strategies, instead the implementations have been tested by playing against humans and other implementations. Each of the separate possible implemented paths through the strategy have been tested several different ways (e.g. testing all three types of blocking in the third strategy).

5.3 Full Game Tree Searching

This strategy is referred to as the “perfect” strategy [8], because it wins in every case it can do. However when considered against the above conditions it becomes less attractive.

1. The strategy can be described algorithmically, but not all that simply, particularly how to construct the complete game tree.

2. Time is where this strategy really fails, although the strategy once the game tree is complete can run in $O(b^2)$ time (where b is the board size) as it only needs to check (normally) one level down the tree and the largest branching factor possible is the number of hexagons, given by $3b^2-3b+1$.

   However the initial construction of the game tree is far more complex, this is $O((b^2)!)$, due to the factorial branching factor in the number of hexagons. This is unlikely to complete before the end of the universe let alone the game!

3. It will always win if it’s possible to do so!

4. Defence is always considered, definite opponent victory paths are only considered if there is no alternative.

5. If it could be run in time it would beat any other strategy.

6. It is difficult to spot the strategy, as it always makes a move likely to make it win.

5.4 “Straight Fork/Bridge Strategy”

1. This strategy is very simple to describe algorithmically, as shown in §3.4.

2. The time complexity of this strategy depends not on the size of the entire board, but on the number of hexagons it tries and fails to place. In the very worst case, of back-tracking across most of the board, this will be no worse than $O(b)$ time, and will often run in $O(1)$ constant time.

3. An attempt is made to win and create a bridge or a fork, if it ever reaches another side of the board.

4. No attempt at all is made at defence!

5. A series of games was run between this strategy and the 'test player' provided by Leipzig [2]. The 'example' strategy works by placing stones randomly around in a single small group, eventually this will form a ring. Because this ring forming strategy is quicker than the very slow, but safe strategy of fork or bridge building we do, the 'example' wins every time.

   A record is kept of each game played between the strategies by the server, an example of which can be found in appendix B.

   If played against itself this strategy will choose two corners and attempt to build from them, at some point the chains are likely to meet, and from then on each strategy will deflect around the other, leading to parallel chains towards one side, whoever gets there fastest then wins. Hence against itself this is often a first player victory (though sometimes one side can get in the way of the other to more effect).

6. The strategy is very easy to spot, a human player can easily follow it's strategy and so perform a faster victory, or even force a resignation by blocking all corners.

5.5 "Straight Fork/Bridge with Defence"

1. This strategy is still simple to describe algorithmically, see §3.5.

2. Time complexity is now increased, due to the implementation the complexity of the additional processing depends on the size of the chain to which the opponent last adds a stone, in theory this could be as bad as $O(b^2)$, but is unlikely to be worse than $O(b)$. It's still certainly no worse than polynomial time.

3. As before an attempt is made to win and create a bridge or a fork, if it ever reaches another side of the board.

4. Now a real attempt to defend against the opponent's victory is made, this is far from perfect though, if there is more than one hexagon position that the opponent can place we can only block one of them for example, no attempt is made to counter this sort of strategy. The strategy also blocks at times when no blocking is required. Bridges and forks can only be blocked if the opponent has stones on each of the corner or side positions they intend to connect.

5. Again several games were run against the 'test player'. This time there is an attempt to block the rings about to be formed by the 'example' strategy, but this does not always work, sometimes we block things that will not immediately become rings. Because the 'example' builds lots of stones in a small area we have to do a lot of blocking stones two apart but unconnected. So often the 'example' finds a way around the ring blocking before we can complete a fork or bridge. This strategy does however win in some cases, around 1 in 10 games it will beat the 'example'.

   Because single non-looping chains of stones are never blocked (the strategy requires some 'unconnected points', or potential rings) it is completely unable to defend against it's own strategy. When run against itself, or against the simpler version without defence it behaves the same way, and no different to the simpler version against itself. So normally a first player victory (against itself and the simple 'Straight Fork/Bridge').

6. The strategy is still easy to spot, but this time the human needs to be a little more subtle, they cannot just build a very fast ring, but need to consider their safety and when the strategy is likely to block them.

Chapter 6

Conclusions

6.1 Introduction

Of the three strategies that have been implemented the best one is the third, “Straight Fork/Bridge with Defence”. Clearly this is an improvement on the simpler version without defence, it also scores over the ‘brute force’ algorithm, as that is impractical to work with on boards of any real size.

6.2 The Implemented Strategies

6.2.1 Complete Game Tree

The complete game tree would be the best strategy if it could be run in a reasonable time, but an $O((b^2)!)$ algorithm is going to expand very quickly with the size of b. This makes it completely impractical on technology of the foreseeable future.

6.2.2 “Straight Fork/Bridge”

This strategy is somewhat limited in it’s actions, it’s easy to see what it’s doing and very easy to beat. But it is a strategy that will win if given enough time to do so. In terms of this project it is a reasonably good strategy as it can be (and has been) implemented algorithmically in roughly constant time. No account is made of defence however, the opponent can construct their chains without safety considerations, and so build very quickly.

6.2.3 “Straight Fork/Bridge with Defence”

This modification to the above strategy allows it to defend in some situations. Thus it significantly improves the quality of the strategy at only a small increase in the time complexity to compute a move.

This is the best strategy I have implemented, but it is still fairly simple to beat, the player just needs to consider their safety more than for the simpler strategy. The defence is not complete, for example the way bridges and forks are blocked single chains are not checked, thus it doesn't play too well against itself.

6.3 Why are computers so bad at Havannah?

A computer player for Havannah than can consistently beat human players has been a development aim for some time. The originator of Havannah, Chris Freeling, has offered a prize of €1,000 [1] for an implementation that can win in merely 1 in 10 games (against him). So why is Havannah such a complex game for a computer to understand?

The game tree is the key, the branching in the Havannah game is much larger than for other games. Consider for example the first move, in Havannah you have a choice of any hexagon on the board, so 271 choices for a board size 10, in Chess there are 20 initial moves, and in Connect 4 only 7.

To compare the size of the game tree it is useful to consider two factors, the branching level - how many children is each node in the tree likely to have, and to some extent how much this varies can be important. And the number of levels in the tree, i.e. the largest number of moves before the end of a game. Connect 4 has around 42 levels and a branching factor of 7, and this is far to much for complete game tree computation. Chess has a significantly larger game tree, estimated at a branching factor of around 38 [4], the number of levels in the complete tree for chess is not fixed, as it is possible for board positions to 'cycle around', in fact games often to not complete to checkmate, or stalemate, but one player often resigns, so the game is not played down the full tree.

For Havannah the game tree becomes even larger, since draws only happen when the entire board is full (and there are lots of draw layouts) the complete tree must have a level for each position on the board, i.e. 271 levels. The branching factor for Havannah is also very large, it starts with every possible board position and then linearly decays by 1 each time a move is made - as you can play anywhere that a stone does not already occupy. Hence an average branching factor for a board size 10 would be $\frac{271+270+269+\cdots+2+1}{271} = \frac{\frac{1}{2}(271\times272)}{271} = \frac{1}{2}272 = 136$

Game	Levels	Branching
Scissors,Paper,Stone	2	3
Tic Tac Toe	9	5
Connect 4	42	7
Chess		38
Havannah (2)	7	4
Havannah (3)	19	10
Havannah (5)	61	31
Havannah (8)	169	85
Havannah (10)	271	136

Table 6.1: Number of levels and average branching factor for various games

Table 6.1 shows the relative game tree sizes for various games and Havannah on boards of varying size, clearly the tree for larger Havannah boards is much larger than that of several other games.

Implementations of chess exist that can beat even the best of human players, so why not for Havannah on boards around size 5?

Another issue is the amount of symmetry on the Havannah board. The board itself has rotational symmetry of order 6, and numerous reflexive bisections. In chess moving a piece to 'king's bishop 3' is a completely different move to moving it to 'queen's bishop 3' despite the fact that they are in symmetrically equivalent positions, in Havannah it does not matter which corners or sides are connected, and a ring can be formed at any position on the board.

Chess players also use a lot of information gathered from other games, particularly as played by human grand masters (as do grand masters themselves), in the form of an 'opening book' (and possible an 'extended book'), this means the computer is making it's decisions based on analysis of many thousands of previous games. [3, 4].

In chess and games such as Go there is a relatively simple procedure to tell how well a player is doing at any stage in the game. The remaining chess pieces or the amount of territory controlled in Go can simply be counted. In Havannah there is no such simple way to check progress. When a frame is formed is not clear, even to experienced players [1], and how near or far a player is from victory is very difficult to tell.

Even more complex is the calculating how far a given configuration is from becoming an unbreakable frame, which would be needed for advanced defensive strategies. Also in Go the concept of how far a chain is from being 'safe' (or 'alive') is a key concept in computer implementations [6]. This is also difficult to implement, and has not been done so entirely successfully despite the ability to easily calculate territory under a player’s control.

6.4 Achievement of Project Aims

6.4.1 Minimum Requirements

Review literature on development of computer implementations of board games

Very little literature has been written on the game Havannah itself, beyond the Mindsports website [1]. More however has been written on other computer games. Chess in particular has a lot of literature, though much of it is irrelevant to this project some sources have been useful [3, 4]. Papers on other games, or computer games in general, [5, 6, 7, 8] has been more useful, particularly the detail on the minimax or complete game tree method [7, 8] used for the first strategy.

The quality of the literature sources is discussed in §2.6.

Develop a strategy for the game, and how a computer could be taught to play

Three strategies have been successfully implemented, chapters 3 and 4 cover the development and implementation of these strategies.

Develop a computer system that plays legal Havannah and recognises victory conditions

Though much of this was already covered in the work done in previous years at Leipzig [2] some corrections and improvements have been necessary (for differing hardware/software). Also understanding how this system works and how to implement new strategies within it was a major part of the project. See chapter 4 for details of the implementation.

Compare this program with other computer implementations of the game and evaluate performance

Chapter 5 concerns the evaluation of the implemented strategies, including comparison of the implemented strategies with the ‘test player’ from Leipzig [2].

6.4.2 Extensions

Further to the development of a simple strategy an improved strategy including defensive capability was developed.

Section 6.3 detailed the main reasons for the computational complexity of the game Havannah, i.e. why it has remained such a long standing open problem while computer's have been taught to play games such as chess with great success.

6.4.3 Project Management

This project did not follow a standard methodology throughout, because it is impossible to predict how long the development and implementation of strategies will take. The background reading and consideration of the game Havannah was completed as intended in the first semester.

Implementation of the strategies was a complex task and took somewhat longer than expected. Originally I had hoped to develop further strategies, but the programming and implementation dominated the time, meaning there was not enough time to implement any further strategies.

Writing up of the project was also delayed by the length of time spend on implementation. Thus the rather long process of writing up such a large report took right up until the deadline for submission. Ideally the report should have been completed in draft form earlier, to allow more time for final alterations.

6.5 Possible Future Enhancements

The solution could be further developed with the implementation of more strategies. Strategies that are less easy for the human opponent to spot would be advantageous. Also it may be possible to refine the defence developed in the third strategy so as to block bridges and forks when the final move is on the side, or improving the ring blocking so as not to block cycles that are not large enough to form a ring.

More complex an implementation could possibly be developed that attempts to form a frame and then complete it. And (apparently extremely computationally complex) a computer could in theory attempt to use some technique to try to spot, and thus block, the formation of enemy frames.

It may also be possible to develop a strategy that learns. However how it should learn is not clear. The absolute positions are in many ways irrelevant, it does not matter where on the board a ring frame is formed, and even the precise location of the central sections of forks or bridges do not matter.

Bibliography

[1] Mindsports. Havannah tutor. `http://www.mindsports.net/Arena/Havannah/`. [25th April 2004].

[2] Johannes Waldmann. Havannah server software. Available via anonymous cvs to `theo1.informatik.uni-leipzig.de:/var/lib/cvs/havannah`. University of Leipzig.

[3] Murray Campbell. Knowledge discovery in deep blue. *Communications of the ACM*, 42:65 – 67, 1999. Also available as `http://portal.acm.org/ft_gateway.cfm?id=319396&type=pdf&coll=portal&dl=ACM&CFID=20582054&CFTOKEN=86607412`.

[4] Timothy McGrew Department of Philosophy Western Michigan University. The simulation of expertise: Deeper blue and the riddle of cognition. `http://www.arn.org/docs/odesign/od191/deeperblue191.htm`. [23rd April 2004].

[5] Timothy Huang Department of Mathematics and Computer Science Middlebury College. Strategy game programming projects. *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 205 – 213, 2001. Also available as `http://portal.acm.org/ft_gateway.cfm?id=378708&type=pdf&coll=portal&dl=ACM&CFID=20582054&CFTOKEN=86607412`.

[6] Timothy Huang Department of Mathematics and Computer Science Middlebury College. The game of go: An ideal environment for capstone and undergraduate research projects. *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 84 – 88, 2003. Also available as `http://portal.acm.org/ft_gateway.cfm?id=611939&type=pdf&coll=portal&dl=ACM&CFID=20582054&CFTOKEN=86607412`.

[7] Aske Plaat Arie de Bruin, Wim Pijls. Solution trees as a basis for game tree search. Technical Report EUR-CS-94-04, Erasmus University, Department of Computer Science, May 1994. Also available as `http://www.cs.vu.nl/˜aske/Papers/tr9404.pdf`.

[8] David J. Thuente Rhys Price Jones. The role of simulation in developing games playing strategies. *Proceedings of the 23rd annual symposium on Simulation*, 20:89 – 97, 1990. Also available as `http://portal.acm.org/ft_gateway.cfm?id=99647&type=pdf&coll=portal&dl=ACM&CFID=14597156&CFTOKEN=7178249`.

Appendix A

Reflections

A.1 Choice of Project

The area of developing and implementing strategies for computer games is one that has interested me for some time. In the past I have looked into strategies for Tic Tac Toe (and developed an unbeatable strategy). When the opportunity to do a project in this area arose, from project topic suggestions, I chose that as an interesting area for development.

A.2 Experience Gained

This is the largest scale project that I have worked on in my academic career. The software project management module (SE22) that I took last year also concerned the development of a software project. This module was very helpful in terms of project planning and setting suitable milestones for development.

A key part of this project was understanding and developing within an existing system, something of which I have little experience. Because the team that developed that system are resident in a completely different country, it was necessary to develop most of my understanding simply from the code provided. Communication with the originator Johannes Waldmann was possible through e-mail and he provided the necessary information to allow my system to be developed.

For the writing up I chose to use LaTeX, despite not being familiar with the language beforehand. I thought that the development during the first semester up to the mid-project report would allow most of the required concepts to be learned during the relatively early stages of the project. Then when it was time to develop the final report the previous code can be adapted. LaTeX makes the handling of references and citations very simple, and makes handling figures and tables much simpler than most commercial word processors. Thus I did not encounter some of the problems I have had writing smaller reports in the past with Microsoft Word. I would recommend LaTeX to other students for their project reports.

A.3 Project Management

The implementation phase was by far the most complex and time consuming. It would have been nice to have developed and implemented more strategies, but owing to coursework commitments and implementation time this became impractical. Writing up of the report should really have been started and completed earlier than it was, and more writing completed over the Easter break, in addition to completing implementation and testing.

Appendix B

A Test Game

The following pages detail the movement history of a single run of the ‘Straight Fork/Bridge with Defence’ strategy against the ‘Example’ strategy. This is one of the cases where our strategy won.

A similar record is kept, by the server, of every game played through it, so testing can be later reviewed.

The detail is split into 20 move sections, to aid readability, each of the moves is numbered. `X` represents a white move, and `O` represents a black move. In this case our strategy was white and the example black.

Dots represent empty hexagons. Connectivity of chains is shown using ASCII connections (`|`, `/`, `\`, `<`).

As can be seen from the detail `O` makes many attempts to construct a ring, which `X` sucessfully blocks. Eventaully `X` manages to complete a fork, starting in the top middle corner, and completing to L3 on the bottom right side.

```
1 : Put J19, 2 : Put H16
3 : Put K19, 4 : Put I16
5 : Put I18, 6 : Put G16
7 : Put I17, 8 : Put J17
9 : Put J18, 10 : Put H15
11 : Put G15, 12 : Put H14
13 : Put K18, 14 : Put F15
15 : Put K17, 16 : Put H17
17 : Put K16, 18 : Put I15
19 : Put K15, 20 : Put E14

                                      19
                                  18     1:X\
                              17     5:X< |  3:X
                          16     16:O |  9:X< |   .
                      15     6:O< |  7:X/    13:X     .
                  14     14:O    2:O\    8:O  |   .       .
              13     20:O    11:X |  4:O/    15:X     .       .
          12      .       .      10:O |   .   |   .       .       .
      11      .       .       .   |  18:O    17:X     .       .       .
  10      .       .       .      12:O     .   |   .       .       .       .
      .       .       .       .       .      19:X     .       .       .       .
  9       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  8       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  7       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  6       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  5       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  4       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  3       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  2       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  1       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
          .       .       .       .       .       .       .       .       .
      A       .       .       .       .       .       .       .       .       S
          B       .       .       .       .       .       .       .       R
              C       .       .       .       .       .       .       Q
                  D       .       .       .       .       .       P
                      E       .       .       .       .       O
                          F       .       .       .       N
                              G       .       .       M
                                  H       .       L
                                      I       K
                                          J
```

```
21 : Put K14, 22 : Put G14
23 : Put F14, 24 : Put F13
25 : Put E13, 26 : Put E12
27 : Put G13, 28 : Put D13
29 : Put D12, 30 : Put I14
31 : Put K13, 32 : Put J15
33 : Put J16, 34 : Put I13
35 : Put H13, 36 : Put F12
37 : Put K12, 38 : Put J13
39 : Put J14, 40 : Put J12
```

```
                                      19
                                  18      X \
                              17      X < |   X
                          16      O   |   X < |   .
                      15      O < |   X /     X       .
                  14      O /     O \     O   |   .       .
              13      O /     X   |   O /     X       .       .
          12     28:O    23:X     O < |  33:X |   .       .       .
      11      .      25:X    22:O |   O \     X       .       .       .
  10      .      29:X    24:O     O < |  32:O |   .       .       .       .
      .       .      26:O |  27:X    30:O     X       .       .       .       .
  9       .       .      36:O    35:X |  39:X |   .       .       .       .
      .       .       .       .      34:O    21:X     .       .       .       .
  8       .       .       .       .      38:O |   .       .       .       .
      .       .       .       .       .   |  31:X     .       .       .       .
  7       .       .       .       .      40:O |   .       .       .       .
      .       .       .       .       .      37:X     .       .       .       .
  6       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  5       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  4       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  3       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  2       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  1       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
          .       .       .       .       .       .       .       .       .
      A       .       .       .       .       .       .       .       .       S
          B       .       .       .       .       .       .       .       R
              C       .       .       .       .       .       .       Q
                  D       .       .       .       .       .       P
                      E       .       .       .       .       O
                          F       .       .       .       N
                              G       .       .       M
                                  H       .       L
                                      I       K
                                          J
```

```
41 : Put I12, 42 : Put E11
43 : Put K11, 44 : Put C12
45 : Put K10, 46 : Put D11
47 : Put C11, 48 : Put C10
49 : Put D10, 50 : Put G12
51 : Put H12, 52 : Put J11
53 : Put F11, 54 : Put J10
55 : Put K9, 56 : Put I9
57 : Put I10, 58 : Put J9
59 : Put K8, 60 : Put I11

                                      19
                                  18      X \
                              17      X < |   X
                          16      O   |   X < |   .
                      15      O < |   X /     X       .
                  14      O /     O \     O   |   .       .
              13      O /     X   |   O /     X       .       .
          12      O /     X /     O < |   X < |   .       .       .
      11     44:O     X /     O < |   O \     X       .       .       .
  10      .       X /     O /     O < |   O   |   .       .       .       .
      .      47:X     O < |   X \     O /     X       .       .       .       .
  9       .      46:O |   O \     X   |   X < |   .       .       .       .
      .      48:O    42:O    50:O |   O \     X       .       .       .       .
  8       .      49:X    53:X    51:X     O   |   .       .       .       .
      .       .       .       .      41:X |   X       .       .       .       .
  7       .       .       .       .       O   |   .       .       .       .
      .       .       .       .      60:O |   X       .       .       .       .
  6       .       .       .       .      52:O |   .       .       .       .
      .       .       .       .      57:X |  43:X      .       .       .       .
  5       .       .       .       .      54:O |   .       .       .       .
      .       .       .       .      56:O |  45:X      .       .       .       .
  4       .       .       .       .      58:O |   .       .       .       .
      .       .       .       .       .      55:X      .       .       .       .
  3       .       .       .       .       .   |   .       .       .       .
      .       .       .       .       .      59:X      .       .       .       .
  2       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
  1       .       .       .       .       .       .       .       .       .
      .       .       .       .       .       .       .       .       .       .
          .       .       .       .       .       .       .       .       .
      A       .       .       .       .       .       .       .       .       S
          B       .       .       .       .       .       .       .       R
              C       .       .       .       .       .       .       Q
                  D       .       .       .       .       .       P
                      E       .       .       .       .       O
                          F       .       .       .       N
                              G       .       .       M
                                  H       .       L
                                      I       K
                                          J
```

```
61 : Put K7, 62 : Put H11
63 : Put G11, 64 : Put H10
65 : Put H9, 66 : Put G10
67 : Put K6, 68 : Put H8
69 : Put I8, 70 : Put G8
71 : Put G9, 72 : Put E10
73 : Put F10, 74 : Put E9
75 : Put F9, 76 : Put D9
77 : Put C9, 78 : Put D8
79 : Put K5, 80 : Put B9

                                      19
                                  18      X \
                              17      X < |   X
                          16      O   |   X < |   .
                      15      O < |   X /     X       .
                  14      O /     O \     O   |   .       .
              13      O /     X   |   O /     X       .       .
          12      O /     X /     O < |   X < |   .       .       .
      11      O /     X /     O < |   O \     X       .       .       .
  10      .       X /     O /     O < |   O   |   .       .       .       .
      .       X /     O < |   X \     O /     X       .       .       .       .
  9       .       O < |   O \     X   |   X < |   .       .       .       .
      .       O /     O /     O   |   O \     X       .       .       .       .
  8      80:O     X   |   X \     X \     O   |   .       .       .       .
      .      77:X    72:O |  63:X     X   |   X       .       .       .       .
  7       .      76:O |  73:X    62:O     O   |   .       .       .       .
      .       .   |  74:O |  66:O |   O < |   X       .       .       .       .
  6       .      78:O    75:X    64:O     O   |   .       .       .       .
      .       .       .      71:X     X   |   X       .       .       .       .
  5       .       .       .      65:X     O   |   .       .       .       .
      .       .       .      70:O     O < |   X       .       .       .       .
  4       .       .       .      68:O     O   |   .       .       .       .
      .       .       .       .      69:X     X       .       .       .       .
  3       .       .       .       .       .   |   .       .       .       .
      .       .       .       .       .       X       .       .       .       .
  2       .       .       .       .       .   |   .       .       .       .
      .       .       .       .       .      61:X     .       .       .       .
  1       .       .       .       .       .   |   .       .       .       .
      .       .       .       .       .      67:X     .       .       .       .
          .       .       .       .       .   |   .       .       .       .
      A       .       .       .       .      79:X     .       .       .       S
          B       .       .       .       .       .       .       .       R
              C       .       .       .       .       .       .       Q
                  D       .       .       .       .       .       P
                      E       .       .       .       .       O
                          F       .       .       .       N
                              G       .       .       M
                                  H       .       L
                                      I       K
                                          J
```

```
81 : Put K4, 82 : Put J8
83 : Put K3, 84 : Put K2
85 : Put L3

                                      19
                                  18      X \
                              17      X < |   X
                          16      O   |   X < |   .
                      15      O < |   X /     X       .
                  14      O /     O \     O   |   .       .
              13      O /     X   |   O /     X       .       .
          12      O /     X /     O < |   X < |   .       .       .
      11      O /     X /     O < |   O \     X       .       .       .
  10      .       X /     O /     O < |   O   |   .       .       .       .
      .       X /     O < |   X \     O /     X       .       .       .       .
  9       .       O < |   O \     X   |   X < |   .       .       .       .
      .       O /     O /     O   |   O \     X       .       .       .       .
  8       O /     X   |   X \     X \     O   |   .       .       .       .
      .       X /     O   |   X /     X   |   X       .       .       .       .
  7       .       O < |   X /     O \     O   |   .       .       .       .
      .       .   |   O   |   O < |   O < |   X       .       .       .       .
  6       .       O /     X \     O /     O   |   .       .       .       .
      .       .       .       X \     X   |   X       .       .       .       .
  5       .       .       .       X /     O   |   .       .       .       .
      .       .       .       O \     O < |   X       .       .       .       .
  4       .       .       .       O /     O   |   .       .       .       .
      .       .       .       .       X   |   X       .       .       .       .
  3       .       .       .       .      82:O |   .       .       .       .
      .       .       .       .       .       X       .       .       .       .
  2       .       .       .       .       .   |   .       .       .       .
      .       .       .       .       .       X       .       .       .       .
  1       .       .       .       .       .   |   .       .       .       .
      .       .       .       .       .       X       .       .       .       .
          .       .       .       .       .   |   .       .       .       .
      A       .       .       .       .       X       .       .       .       S
          B       .       .       .       .   |   .       .       .       R
              C       .       .       .      81:X     .       .       Q
                  D       .       .       .   |   .       .       P
                      E       .       .      83:X     .       O
                          F       .       .      85:X     N
                              G       .      84:O     M
                                  H       .       L
                                      I       K
                                          J
```