MEng Final Year Project Report

Seasonal Hex and Other Alternate Rules and Players for the Game of Hex

Author:
Andrew Trevorah

Supervisor:
Dr. Mark Herbster

April 30, 2010

Abstract

Hex is a 2 player game played on a board of hexagonal tiles. It can be seen as having a similar complexity as chess or go, but with simple rules and a clear winning state that allows new players to quickly pick up and play with ease. The game offers intricacies that make the game rewarding to master and, combined with the impossibility of a draw-game, gives a satisfying game.

We will define and implement a generalisation of Hex called "Seasonal Hex". This variant rules that players can only play on sets of tiles at a given time specified by a "Season". We shall also implement Random Turn Hex, another generalisation of Hex that decides which player can move by a coin toss rather than a turn by turn basis.

For both generalisations, we will design a variety of Artificial intelligence (AI) players that can not only play to these rules, but also take advantage of the nuances that they offer. To observe the implementations of these new rules and players, we will create a stable platform on which to experiment with the details of each design and observe their results.

Contents

Chapter 1

Introduction

1.1 Aims

The ideas raised by this paper can be seen as the solution to the following motivational problem:

> Mr Black and Mr White are two rival builders that are attempting to build two separate canals, Mr Black connecting the north with the south, and Mr White connecting the east with the west. To stop the two teams clashing, the local Mayor has split the Hexagonal land into areas where building can only commence on certain days. The Mayor must also decide on new rules to keep the confrontation fair.

The Mayor can be seen as the implementation of the rules. For this, we shall define three separate rules - Seasonal Hex, Random Turn Hex and Classic (original) Hex. The new rules must be fair enough so that two equally matched players would gain no advantage by swapping their starting position with the opponent or by using a different turn order. This idea of "fairness" must also be limited in the sense that it is possible to be a better player. We will maintain the classic hex board structure so no game can end with a draw. The aim is then to modify the playing rules and not the board.

Seasonal Hex is the idea that by cutting off players from areas of the board at times individual to them, it may be possible to render potential perfect strategies useless. However, this is a difficult task as it could be very tricky to maintain a balance of restriction between the two players. Deciding when and where a player can play could either be too open and be identical to classic Hex, or too limiting. This severe restriction could make the game too complex to be enjoyable or give one player an advantage over the other.

Random Turn Hex looks outside of the usual turn-by-turn play style used by Hex and tries adding rules that severely limit the predictability of the game. This has an added benefit of producing players that are fast and effective enough to be run on boards that would normally suffer from an excessive branching factor (too many possible moves). This short-sighted style of play may produce fast beginner level players but could potentially alienate players looking for more depth.

The view of a canal builder is the problem that the AI players are designed to face. This will be done by developing several distinct methods of determining the value of a potential move, known as an evaluation function. Obviously these evaluation functions will mainly depend on the new rules set, but existing strategies may still hold true.

For Seasonal Hex, our two AI players will focus on two possible areas - simple masking of the possible plays, and a more in depth player that can take advantage of the season patterns. Masking is a technique that can be applied to any player, but its lack of deep season mechanics may give it a handicap. We will create an AI player that has a deeper understanding of the mechanics than just 'masking' positions. This deeper player will be built off the principles of the masked player, so it will be intriguing to see how effective this extra layer of understanding is.

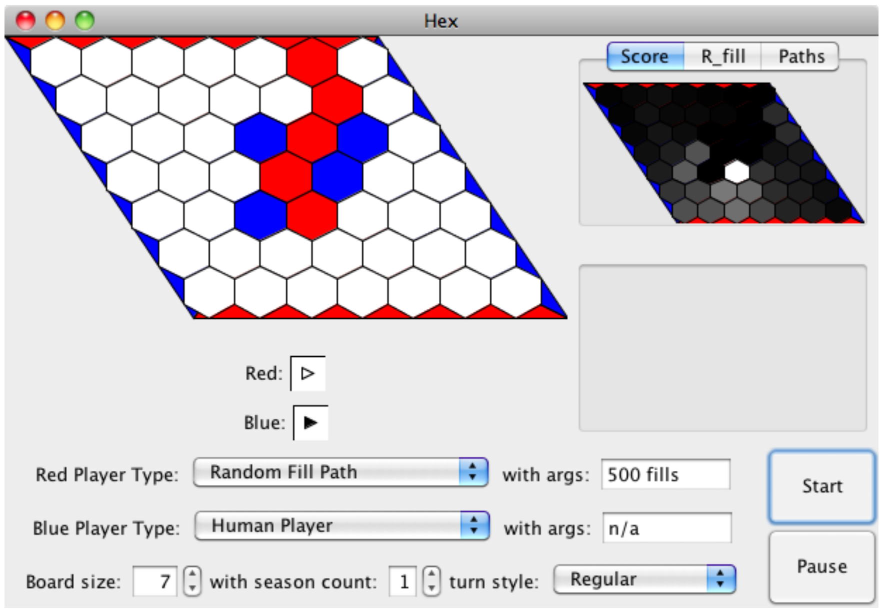
For Random Turn Hex, we shall look at previous work done on the theoretically optimum player who plays by judging randomly sampled virtual games, and see how effective its implementation is. We shall also create an additional player that is designed to cope better with a smaller subset of these virtual games. It will be very interesting to see which performs better when both have to work with less of these games.

It will also be interesting to see how well these evaluation functions can perform under the each others rules and even the rules of classic hex. The new rules may also enable players that use techniques previously seen as being unsuitable under classic hex rules to flourish.

In order to reliably test these rules and respective players, we must build a stable program platform. Our use of clear and carefully abstracted code not only enables regular hex to be played by a human in an intuitive manner, but to also feature modifications of the rules and AI techniques.

This abstraction even enables us to mix the rules providing additional unique new aspects of gameplay. All possible AI players can also operate on any possible combination of rules which allows us to test and force their effectiveness outside of their original intended constraints. Correct use of interfaces also allows additional work on new AI players to be created easily.

The platform also monitors the working memory of each player and outputs the data in a visual manner to give us an insight into the mechanics of an evaluation of a game state (as shown in figure 1.1). These insights give us more data to draw conclusions about the AI players than simply running a round robin tournament.



Red Player Type: Random Fill Path with args: 500 fills

Blue Player Type: Human Player with args: n/a

Board size: 7 with season count: 1 turn style: Regular

Figure 1.1: Screenshot from the new Hex Program

1.2 Report Structure

Chapter 1 Describes the aims of the project

Chapter 2 Gives an outline to the history of Hex and previous work on the subject

Chpater 3 An in depth look at how an instance of the board is represented and manipulated by the players

Chapter 4 Introduces Seasonal Hex and other Hex rule variants

Chapter 5 Describes the designs of the “Seasonal Player” AIs

Chapter 6 Describes the designs of the “Random Turn Player” AIs

Chapter 7 Experiments run on the players to determine their effectiveness

Chapter 8 Concludes the results and looks into possible areas of improvement or investigation.

Chapter 2

Background

Hex is a game of perfect information (all players have access to the same total knowledge) that is played on an area of hexagonal tiles that are laid out in the shape of a rhombus. Each player picks whether to play as black or white, with black playing first. White aims to connect the left and right edges by a path of its own counters, while black tries to connect the top with the bottom with its own path of black counters. The winner is the first player to make a connecting path.

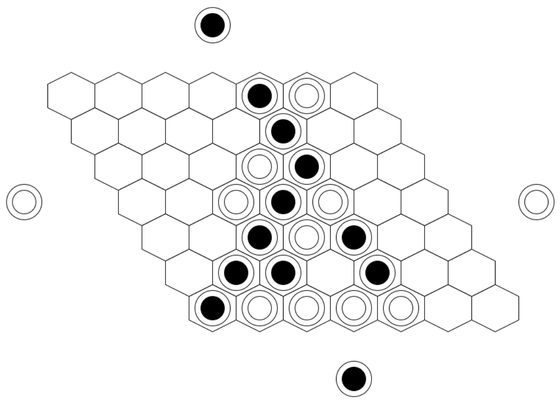


Figure 2.1: example of a 7×7 game where black has won

2.1 History

The Danish mathematician Piet Hein invented the game in 1942, which then appeared in the Danish newspaper Politiken with the name of "Polygon". The same game was also independently invented by John Nash in 1947 [1], and the game was given the name of "Nash". However, the Parker Brothers were the first to release the commercial version which goes under the name of "Hex". It was played on a 11×11 board which is now regarded as the standard board size [8].

2.2 Impossibility of a draw

The proof of the lack of a draw in Hex is based on a separate similar proof for Hex and the Brouwer Fixed-Point Theorem by David Gale [3], who in fact built the first board for Nash [1]. Please note that the graph representation in the proof is different to the adjacency graph that will be shown in chapter 3.

Let G be a planar 2-connected graph in which all vertices have degree 2 or 3. The hexagonal shapes that these vertices create can be seen as the Hex cells of a Hex Board. The nodes that form the corners of the board shall be called a, b, c and d. We can assume that all of these cells that we have formed are occupied as a game of Hex cannot end before a player wins or no cell is available. We can then make a subgraph of G whose vertices are those that have contrasting colours on either side. This 'edge detection' subgraph shall be called G'.

G' will consist only of vertices of value 0 or 2, namely because of the fact that no vertex can be visited twice. It is also a fact that graphs with vertices of value 2 or less will consist only of cycles, isolated vertices and non splitting paths. The only vertices of value 1 are the four corners (a, b, c and d) and thus there must be only two paths which cannot cross each other. We can clearly see that there are only two possible cases for these paths. Either a connects to b and so c to d resulting in one player connecting their borders and winning, or a connects to c and so b to d resulting in the other player connecting and winning.

2.3 Players Advantage

As stated before, Hex is a finite, perfect information game that cannot end in a tie. This means that either the first or second player must possess a winning strategy. It can also be noted that ownership of an extra move for either player will improve that player's position. Therefore, if the second player were to possess a winning strategy, then the first player could simply copy it by making a meaningless first move and then follow the second player's strategy. If the strategy required to place a piece on an already taken hex, then an additional meaningless move it then played. This strategy would ensure a win for the first player.

This proof was originally shown by John Nash in 1949 [1], but we must note that this proof only proves that a winning strategy exists, but does not offer that winning strategy. However, Jing Yang in 2003 has found the perfect strategies for the 7×7, 8×8 and 9×9 board sizes [4].

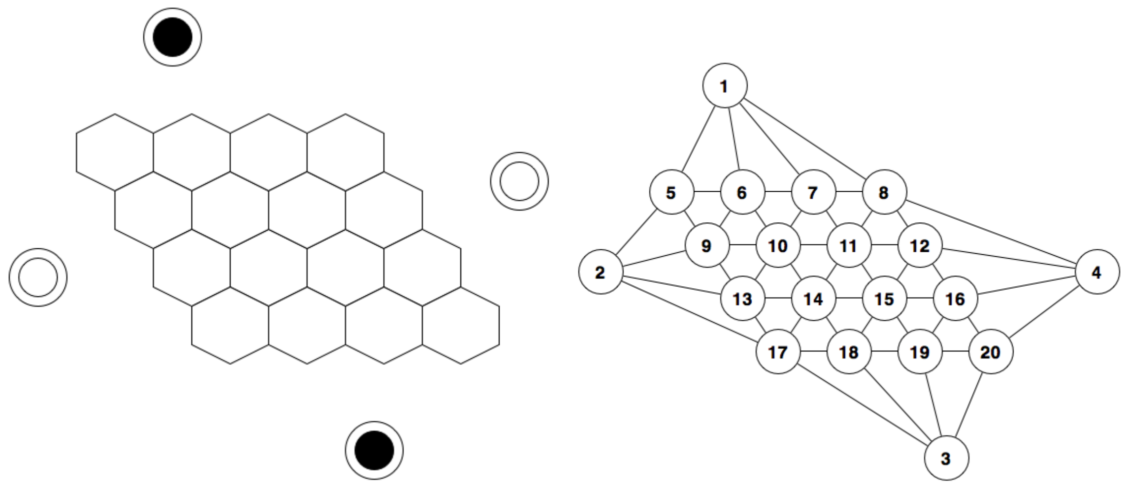
2.4 Complexity

Hex is said to have a computational complexity that is P-Space complete [7]. This puts it in the same frame as games such as checkers. Hex's difficulty stems from the large range of potential moves that a player can make. This high "branching factor" can be compared to the branching factor of other games such as chess. Chess can be seen as having a low branching factor due to the relatively small amount of legal moves per turn. As Hex has a high branching factor, the need for a powerful evaluation becomes apparent. A powerful evaluation function means that we do not need to look to far into a possible future to evaluate a move's worth.

Chapter 3

Board Representation

When playing with just human players, an $n \times n$ board could simply be represented by a two dimensional array as shown in figure 3.1a. But in order to take advantage of the relationships between positions, a computerised artificial intellegence player could use the graph form shown in figure 3.1b instead.

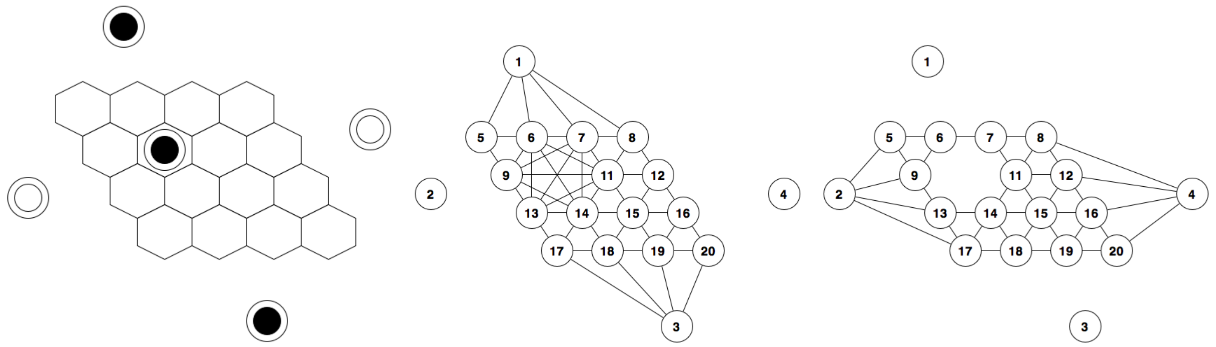


(a) Hex board (b) Graph form

Figure 3.1: A 4 x 4 Hex board with equivalent graph form

In the graph form, each node is a position on the board, with each of the four borders represented as additional nodes north, south, east and west of the main nodes. If a position on the hex board is adjacent to another position, then this is noted on the graph by connecting the two representative nodes with an edge.

Once the game begins, we can represent that there are two different views of the board by creating two different adjacency graphs each with their opponent's special border nodes detached from all other nodes. By having two graphs, we can represent how one players ownership of a position is helpful to them, but a hindrance to the opposing player. In figure 3.2 we can see how the board representations differ when a player owns a position. If a player occupies a position, the neighbours of a position are then "linked" to each of the other neighbours through the taken node while the node itself has it's own links severed. This can be seen in figure 3.2b as bypassing the given node.



(a) Board (b) Player Point of view (c) Opponent Point of view

Figure 3.2: Game with black owning position (2,2)

The view of the opponent would of course differ. When a position is taken by another player, then it is of no use and all relationships to other positions are severed. This can be shown by removing all edges connected to the given node as seen in figure 3.2c.

We can see that when a path is formed of player owned positions from one boarder to another (a winning state), then the bypassing of each node would mean that the border nodes would then be directly connected. The opponent's view of the winning state situation would be a path of unconnected nodes cutting though the board, making their own connected path impossible.

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{pmatrix}$$

Figure 3.3: Matrix view of Figure 3.2b

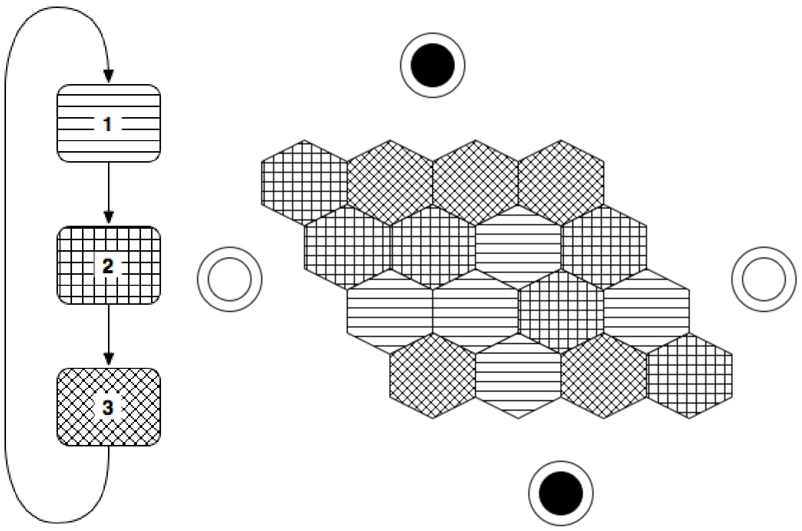
Storing these graphs can be done by use of an adjacency matrix. Each node has both a column and a row showing the links from the node and to the node. As you can see in figure 3.3, The 10th entry corresponding to node 10 has no attachments to other nodes as given in the example. Due to the symmetry that the edges maintain between nodes, the matrix itself is symmetrical. Note that each node shares a link to itself. This is not entirely necessary but is incredibly useful to certain functions that we can perform on this matrix that we shall mention later. The sparseness of the matrix and its symmetry means that we can store it in the implementation as an Upper Symmetrical Sparse Packed Matrix provided by the Java Matrix Toolkit library.

Chapter 4

Proposed Alternate Hex Rules

4.1 Seasonal

The idea of seasonal hex is to limit play to a certain subset of the board for each player's turn. Each subset can be seen as only being playable during a given "season". A position can only belong to one season and all positions must belong to a season. A "year" is a complete cycle of the seasons, in which each season features exactly once. The ordering of the seasons within a year may be different for both players. The changing of the seasons is known to both players and occurs after every move.

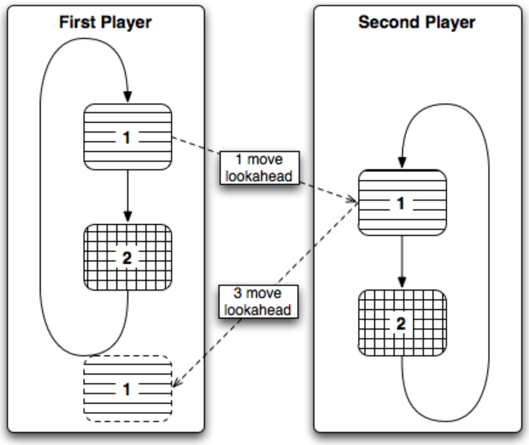


(a) Seasons (b) Seasons allocated to a 4 x 4 board

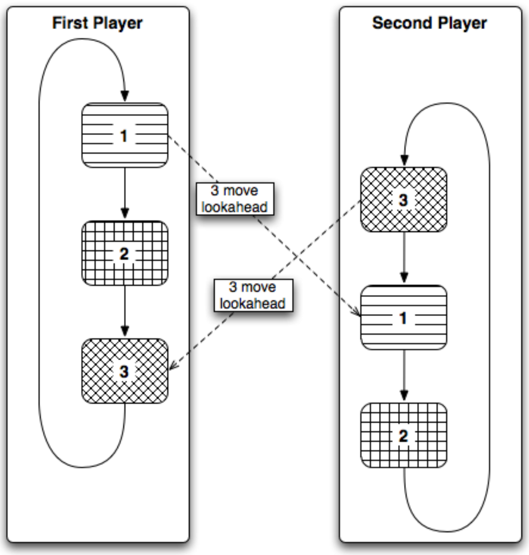
Figure 4.1: 3 Seasons Game

For example, figure 4.1 shows a 4×4 game with 3 seasons. For the first player's first move, they can only play on (3,2), (1,3), (2,3), (4,3) and (2,4). Then, on their second move, they can only play on (1,1), (1,2), (2,2), (4,2), (3,3) and (4,4).

The season ordering within a year must be carefully chosen, as it may provide one player with an advantage over the other. This advantage is caused by noting how far a player must look ahead in the game



(a) Poor season structure



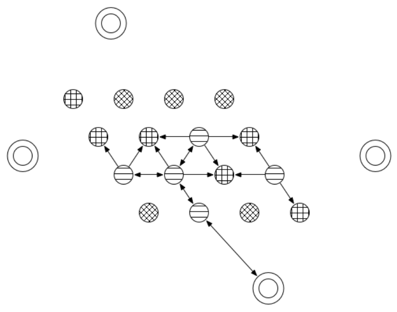
(b) Good season structure

Figure 4.2: Possible season structure for the given yearly cycles

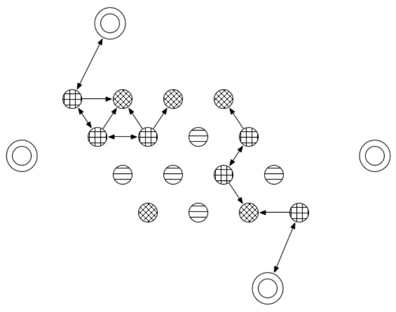
(this process is known as a "lookahead") to note which position their opponent will desire out of the player's potential plays for the current season. In Classic Hex, a player can simply use a lookahead of one move to get an answer, and so can their opponent. However, in Seasonal Hex the selection of positions corresponding to the player's current season may not come into effect for the opponent's next move.

To help explain this, observe figure 4.2a. This (1,2) (1,2) arrangement of two seasons per year gives the first player an advantage over the second as their minimum lookahead required is only one move. However, the second player must use a lookahead of three moves in order to make a judgement on which of his potential moves would be desirable to his opponent. This advantage means that one player would require a more complex evaluation function than their opponent in order to compete.

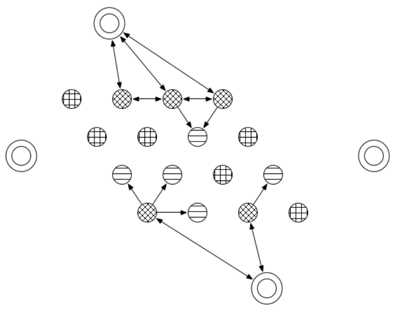
In figure 4.2b we can see a solution to this problem for a 3 season game with yearly cycles of (1,2,3) (2,3,1). This particular arrangement means that both players require a lookahead of 3 moves to judge their opponent's intentions. For Seasonal Hex games, this particular arrangement is recommended.



(a) Season 1



(b) Season 2



(c) Season 3

Figure 4.3: Graph subsets for each of the 3 seasons

To allow players to take advantage of the seasonal structure of a year, we can make note of this data within the graph representation of the game board detailed in chapter 3. When we view the board from the player's point of view, we must take into account the ordering of the seasons. The result of this is that the graph form is split into 3 subsets, each corresponding to the possible current seasons (see figure 4.3). The normal hex adjacency rules still apply (the player's positions are bypassed, the opponent's are completely removed), however only nodes of the same season (both external boarder nodes belong to *all* seasons) share a symmetrical link. Asymmetrical links from the nodes connect to any neighbouring node that belongs to the next season, thus maintaining the order of seasons.

4.2 Random Turn

Random turn is a very simple rule that makes a large change to how the game plays. Instead of the turn-by-turn rules of classic Hex, Random Turn Hex requires that a coin toss decides which of the two players should be allowed to play at a given turn. This means that a player will not know when their next move will occur, leading to interesting games like the one shown in figure 4.4.

This greatly increases the amount of possible moves and outcomes, thus requiring a strong evaluation function. This also makes use of discarding potential unlikely moves (used by the min-max method) almost impossible due to the unpredictable nature of deciding which future move would belong to which player.

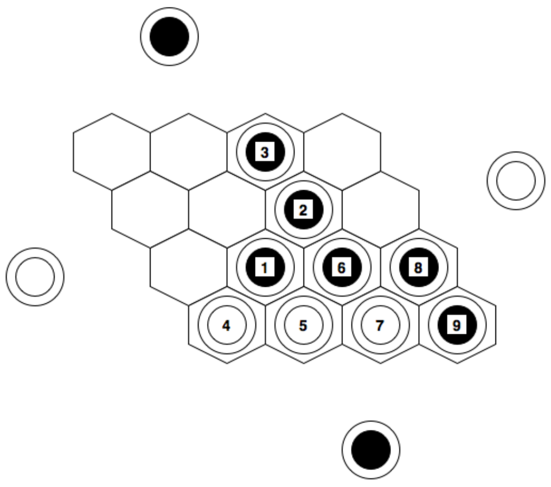


Figure 4.4: example of a 4×4 Random turn game where black has won. Pieces are labeled by move order

Previous work by Peres et al [2] have shown that the best move for one player is also the best move for the opponent. This unique property combined with the loss of the initial move advantage may make for a fairer game. It is also noted that the game of Random Turn Hex holds parallels with real world conflicts in which problems are rarely played out turn by turn.

Chapter 5

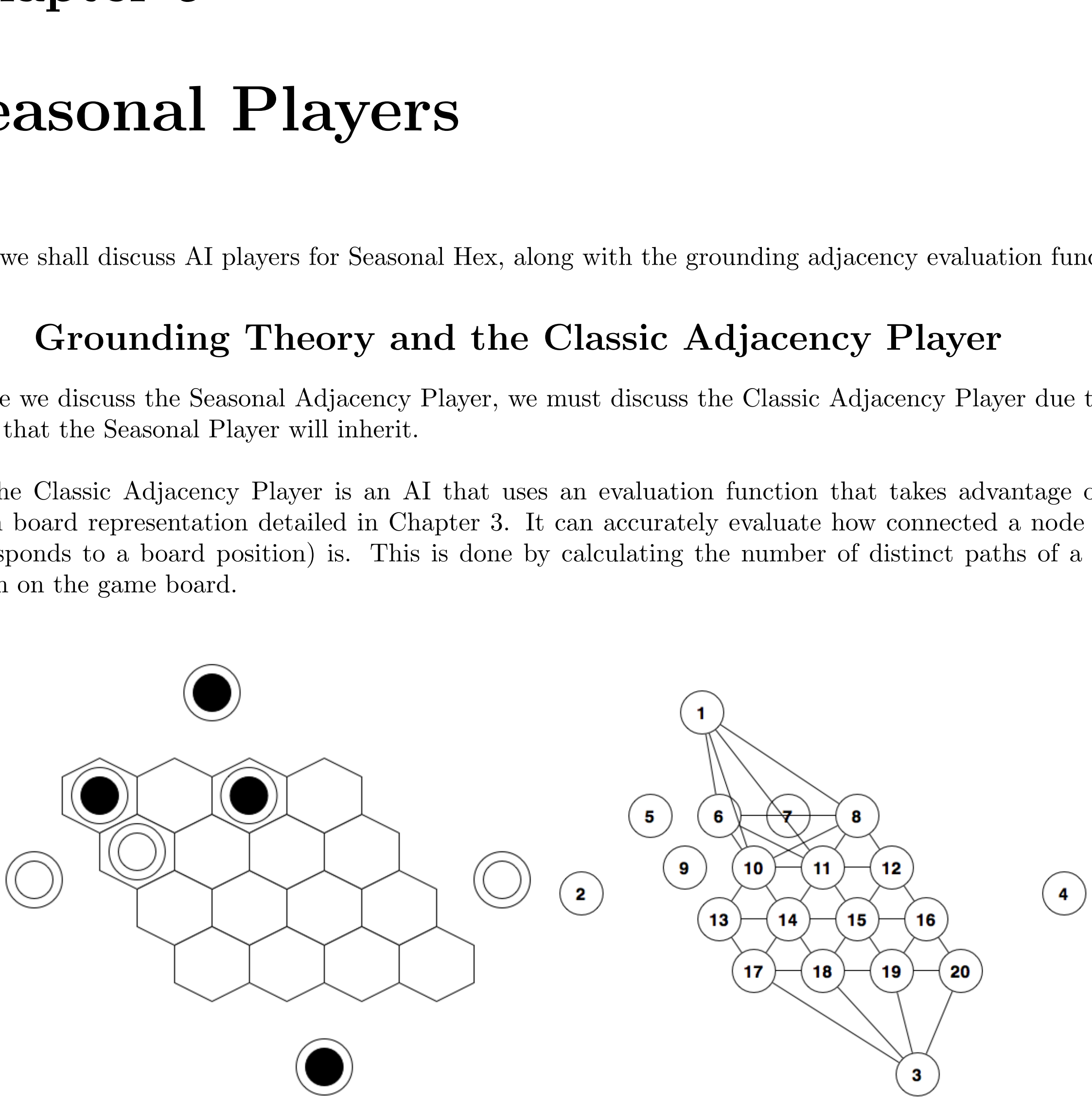
Seasonal Players

Here we shall discuss AI players for Seasonal Hex, along with the grounding adjacency evaluation function.

5.1 Grounding Theory and the Classic Adjacency Player

Before we discuss the Seasonal Adjacency Player, we must discuss the Classic Adjacency Player due to the ideas that the Seasonal Player will inherit.

The Classic Adjacency Player is an AI that uses an evaluation function that takes advantage of the graph board representation detailed in Chapter 3. It can accurately evaluate how connected a node (that corresponds to a board position) is. This is done by calculating the number of distinct paths of a given length on the game board.



(a) Game Board (b) Graph Representation (Red point of view)

Figure 5.1: Example Board with adjacency graph

For example, we shall evaluate the board shown in figure 5.1 for the black player. We need to inspect the paths between the two relevant top and bottom border nodes (1 and 3) in order to evaluate the board for the black player. We can see that the minimum path length between the two nodes is 4, but the *number* of these paths of length 4 is a little harder to count by eye.

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{pmatrix}$$

Figure 5.2: Adjacency Matrix M

What we can do is use the matrix representation of the graph as specified in chapter 3. This adjacency matrix M holds the property that when multiplied by itself l times (M^l), it will hold the number of paths of length l as its values. So for our example, we must calculate M^4 to find that number of paths of length 4 between nodes 1 and 3 is 7 as shown in figure 5.3.

To get a complete view of the game, we also analyse the opponent's board graph using the same techniques. In order to compare different board instances, we must combine these views using the following function as specified by Lu [5]:

$$\begin{aligned}
Score &= (S_1 \& S_2) \\
S_1 &= \Delta \times mylength - (1 - \Delta) \times opplength \\
S_2 &= \Delta \times (-my\#paths) + (1 - \Delta) \times opp\#paths
\end{aligned}$$

Where Δ is a constant between 0 and 1 that defines the aggressiveness of a player. An aggressive player ($\Delta = 1$) will ignore the opponent, whereas a defensive player ($\Delta = 0$) will only consider the opponent's view.

S_1 values a board by the minimum path length of the player (*mylength*) and the opponent (*opplength*), with a higher value given if a board features a large minimum distance between a player's borders and a small minimum distance between the opponent's borders. A smaller value of S_1 is better for the player.

S_2 values a board by the number of paths that are of the minimum path length. We aim to have a large number of paths for the player (*my#paths*) meaning several choices for good moves, but a small number of paths for the opponent (*opp#paths*). A small value of S_2 is better.

$$\begin{pmatrix}
135 & 0 & \mathbf{7} & 0 & 0 & 135 & 0 & 148 & 0 & 157 & 172 & 83 & 53 & 96 & 74 & 27 & 28 & 31 & 17 & 4 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{7} & 0 & 88 & 0 & 0 & 7 & 0 & 11 & 0 & 25 & 37 & 33 & 36 & 74 & 85 & 63 & 72 & 100 & 98 & 64 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
135 & 0 & 7 & 0 & 0 & 135 & 0 & 148 & 0 & 157 & 172 & 83 & 53 & 96 & 74 & 27 & 28 & 31 & 17 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
148 & 0 & 11 & 0 & 0 & 148 & 0 & 166 & 0 & 172 & 196 & 102 & 57 & 109 & 95 & 42 & 31 & 40 & 28 & 10 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
157 & 0 & 25 & 0 & 0 & 157 & 0 & 172 & 0 & 198 & 212 & 100 & 86 & 146 & 107 & 37 & 63 & 65 & 34 & 10 \\
172 & 0 & 37 & 0 & 0 & 172 & 0 & 196 & 0 & 212 & 248 & 136 & 85 & 164 & 150 & 72 & 65 & 86 & 64 & 27 \\
83 & 0 & 33 & 0 & 0 & 83 & 0 & 102 & 0 & 100 & 136 & 97 & 35 & 89 & 112 & 72 & 33 & 61 & 66 & 36 \\
53 & 0 & 36 & 0 & 0 & 53 & 0 & 57 & 0 & 86 & 85 & 35 & 63 & 96 & 61 & 19 & 64 & 63 & 32 & 12 \\
96 & 0 & 74 & 0 & 0 & 96 & 0 & 109 & 0 & 146 & 164 & 89 & 96 & 170 & 140 & 64 & 106 & 124 & 86 & 39 \\
74 & 0 & 85 & 0 & 0 & 74 & 0 & 95 & 0 & 107 & 150 & 112 & 61 & 140 & 168 & 108 & 83 & 128 & 124 & 70 \\
27 & 0 & 63 & 0 & 0 & 27 & 0 & 42 & 0 & 37 & 72 & 72 & 19 & 64 & 108 & 89 & 39 & 79 & 98 & 64 \\
28 & 0 & 72 & 0 & 0 & 28 & 0 & 31 & 0 & 63 & 65 & 33 & 64 & 106 & 83 & 39 & 89 & 100 & 71 & 38 \\
31 & 0 & 100 & 0 & 0 & 31 & 0 & 40 & 0 & 65 & 86 & 61 & 63 & 124 & 128 & 79 & 100 & 134 & 116 & 67 \\
17 & 0 & 98 & 0 & 0 & 17 & 0 & 28 & 0 & 34 & 64 & 66 & 32 & 86 & 124 & 98 & 71 & 116 & 129 & 84 \\
4 & 0 & 64 & 0 & 0 & 4 & 0 & 10 & 0 & 10 & 27 & 36 & 12 & 39 & 70 & 64 & 38 & 67 & 84 & 60
\end{pmatrix}$$

Figure 5.3: Resulting matrix M^4 with the value for the link between nodes 1 and 3 in bold

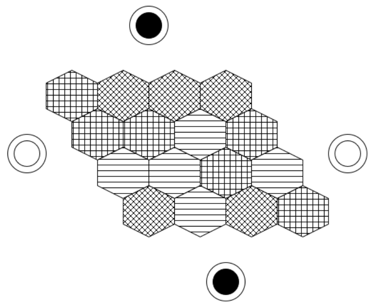
To compare two boards, we first compare the S_1 values, with the smallest value belonging to the better board. If there is a tie, then we look for the smallest S_2 value. For example, (3 & 4) is better than (3 & 6).

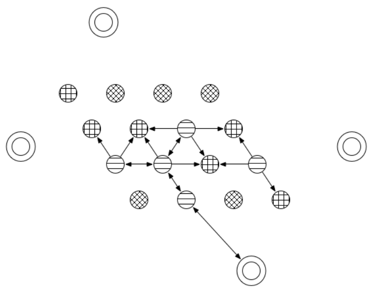
An AI can use this evaluation function by playing virtual moves and using the evaluation function to rate the outcome. The outcome with the best value features the moves that should be played. The AI player that uses this function is called Classic Adjacency Player, and uses a value of delta of 0.5 for simplicity.

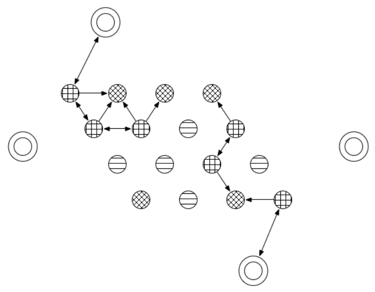
The Adjacency Players operate with the following code:

```
For each position{
  set position as player owned

  calculate mylength and note my#paths

  calculate opplength and note opp#paths

  evaluate Score for position
}
Submit move as position with lowest value of Score
```

When evaluating a possible board layout, it will loop while multiplying M with itself until it finds a path between the border nodes. Care must be taken when evaluating a winning board state, as the lack of a draw means that one player will never connect their borders, and so the multiplication loop could be infinite. We can avoid this by setting a cutoff that is close to the maximum path length (for an $n \times n$ board) of n^2.


(a) Game Board


(b) Graph Representation Season 0


(c) Graph Representation for Season 1

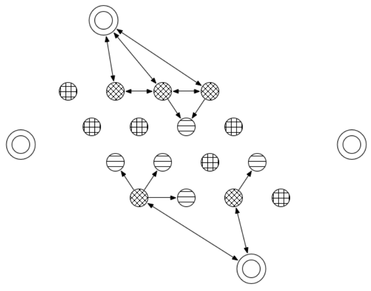

(d) Graph Representation for Season 2

Figure 5.4: Example 3 season board with seasonal subgraphs for the black player

5.2 Seasonal Adjacency Player

The Seasonal Adjacency Player works in a similar manner to the Classic Adjacency Player, with a slight change to how the values for the number of paths are calculated. Instead of raising the power to the matrix, It multiplies the seasonal graph subsets mentioned in chapter 3.

For example, a seasonal board has 3 graph subsets: $M_{season1}$, $M_{season2}$ and $M_{season3}$. If the current season is season 1 (with corresponding graph $M_{season1}$), then to find the number of paths of length 3, we use $M_{season1} \times M_{season2} \times M_{season3}$. The resulting matrix can be used in the same fashion as the Classic Adjacency Matrix uses its own resultant matrix (M^3). The multiplication of the subgraphs must be done in the same order as the yearly loop, so for example paths of length 3 during season 2 would be found by calculating $M_{season2} \times M_{season3} \times M_{season1}$.

The real power of this technique for the Seasonal Adjacency Player is the use of the directed subgraphs. By using directed graphs, we can infer the changing of the seasons. Sadly the implementation of this player only has access to undirected subgraphs, and so may perform poorer than hoped.

5.3 Adapting Other Players For Seasonal Hex Rules

The easiest way to adapt another evaluation function is to simply mask out results for areas that are not members of the current season. This “masking” ensures that no illegal moves are suggested. This is quite a shallow solution as the adapted function will still waste time evaluating impossible moves.

Chapter 6

Random turn players

Here we shall look at the Random Spot Player and our implementation along with our designs for the new Random Path Player.

6.1 Random Spot Player

The Random Spot player was originally defined by Peres et al [2] as the theoretically optimal player for Random Turn Hex. However, in practice, the time required for an optimal evaluation can be seen as unrealistic.

Our implementation of Random spot tests out each playable position by placing a virtual piece and then repeatedly filling out the remains with randomly chosen pieces. Each "random fill" is noted whether it resulted in a win or not by the player. The position that had the highest amount of perceived wins is seen as being better. Our implementation of the player's evaluation function operates in the following way:

```
For each position{
  Set position as player owned

  Repeat{
    For all empty board positions
      Set randomly as either player owned or opponent owned

    If path exists from one border to another
      Increase score for position

    Remove random filled positions
    }
}
```

Note that when specifying the amount of repeating random fills a Random Spot player will use, the amount is divided equally for each position's loop.

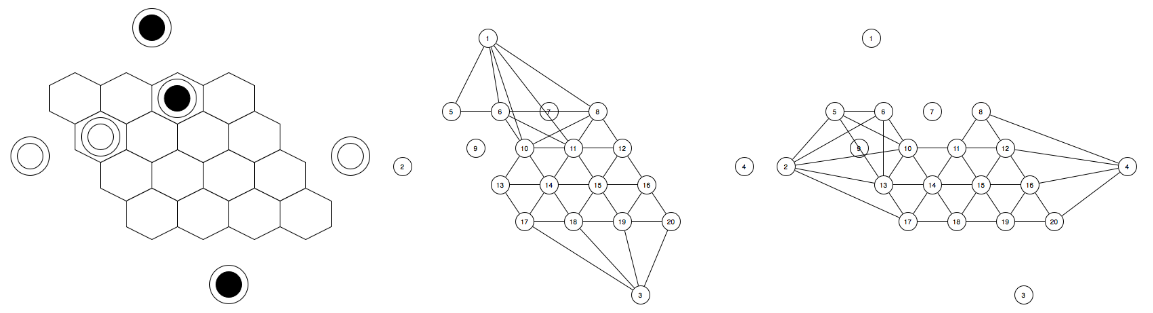
6.2 Random Path Player

Random Path uses the same idea of a "random fill" that the Random Spot player uses, but instead of just noting whether a fill resulted in a win or not, it analyses the win to see which positions were part of the winning path that links the two borders. This is done by the following code:

```
Repeat{
  For all empty board positions
    Set randomly as either player owned or opponent owned

  If path exists from one border to another
    Increase score for available positions that feature in the path

  Restore board to original state
}
```

Note that it does not test each potential move in turn. This means that the amount of repeating random fills specified is used for the entire board. This means that the amount of fills roughly equals the same that a Random Fill Spot player would use given the same arguments.

To find the winning path we use a slightly modified version of the graph board representation specified in chapter 3. To demonstrate this we will use the board layout given in figure 6.1.



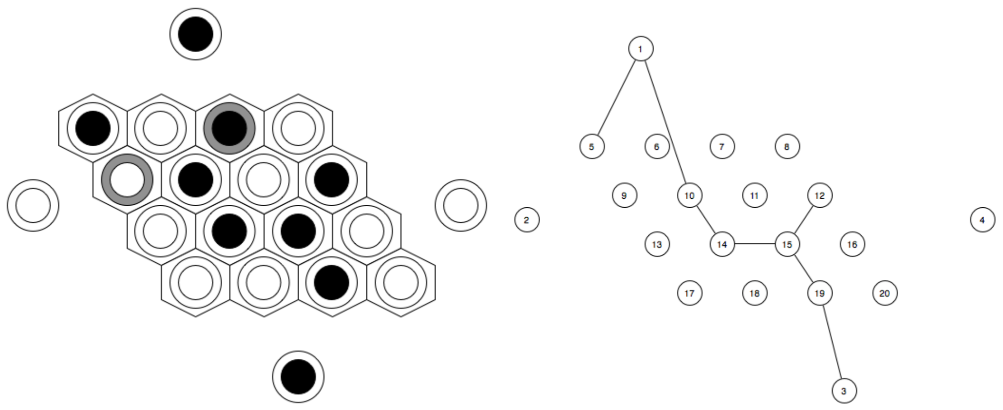
(a) Game Board (b) Graph Representation (Black point of view) (c) Graph Representation (White point of view)

Figure 6.1: Example Board with Black to move

Filling empty spaces with either black or white will be done by visiting each unclaimed node on the graph and choosing with a 50:50 chance whether or not to sever all related edges. By severing a node's edges, we are simulating that the particular node has been chosen by the opponent. Normally in order to simulate the position has been taken instead by the player, we would bypass the node. If the random fill resulted in a win for the player, the method of bypassing nodes would create a direct link between the border nodes. As we want to know about the winning path, we would prefer that the direct link would show the individual nodes as a direct link would be of no use. Instead, we maintain the player owned nodes by leaving the nodes untouched.

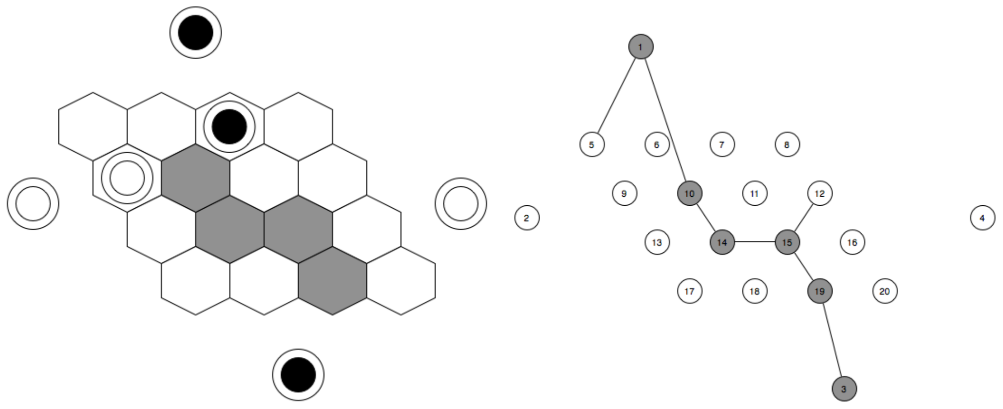
Due to leaving the randomly allocated player nodes intact, we can explore the structure to determine two things: if a path exists that connects the two borders and which nodes feature in this path. This can be done by a breadth first search (note: we maintain that the order that a nodes children is explored at random) initiated at the first border node and searching for the second border node. By recording each parent node, we can determine the path from one border node to the other.

for each member of this "winning path" that is free to play, we raise the score for that position. As the initial board graph only features free nodes (player's nodes are bypassed, opponents nodes are wiped), the only members of the winning path that are unplayable are the border nodes. All other nodes are noted and their individual score is raised.



(a) Game Board after random fill (b) Graph Representation (Black point of view)

Figure 6.2: Results of Random fill



(a) Game Board with path highlighted (b) Graph Representation showing path

Figure 6.3: Path found through breadth first search

6.3 Adapting Other Players For Random Turn Hex

As all the available positions to play are identical to the same game in regular Hex, a regular player needs almost no adaptation to play and so no masking. However, as the game tree (view of potential future moves of a game) is wildly different from a regular player's predictable model, the use of said model is discouraged as it may hinder the player by relying on followup moves that may never occur.

Chapter 7

Experiments

In these experiments we shall be looking at the strength of an AI player by its opening move on a Classic Hex board, along with the results of a round robin tournament (each player plays every other player) for each rule type (Seasonal, Random turn etc). The players to be tested are:

- Random Path Player
- Random Spot Player
- Seasonal Adjacency Player
- Classic Adjacency Player

For the Random Path Player, we will test using 1000 random board fills and 100 random board fills. Sadly, due to the Random Spot Player's inefficiency, we shall only test using 1000 random board fills.

7.1 Opening Move Strength

For each player, we shall note the results of the evaluation of each position as it aims to play to connect the top and bottom borders. In order to visualise this, we can use a heat-map that assigns a colour between black and white to each position. This colour is based on the position's relative score as shown by the following:

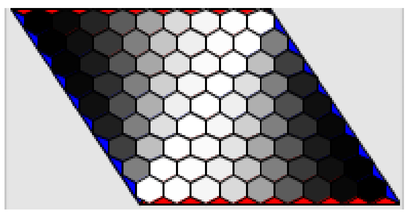
$$ColourIntensity = \frac{Score - lowestScore}{topScore - lowestScore}$$

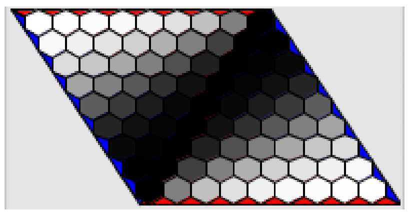
The $topScore$ and $lowestScore$ are the highest and lowest scores found on the board. A high $ColourIntensity$ will be close to white due to a high relative score, and a low $ColourIntensity$ will be closer to black due to a low relative score. Note that the strongest opening move for Classic Hex is the centre position.

Seasonal Adjacency Player / Classic Adjacency Player

As a Classic Hex Board is identical to Seasonal Hex with only one season, the Seasonal Hex Player will have just one graph subset that is the graph itself. Thus, the results of the Seasonal Hex Player will be identical to the Classic Adjacency Player.

As the minimum path lengths for each potential play are identical, we will rely on the S_2 tiebreaker. In figure 7.1 we see the path counts used as a tiebreaker given the uniform path lengths, and the combination of the results. Note that a lower (darker) score is better. The player seems to keep well to a strong central move.


(a) $my\#paths$ = Player's number of paths of the shortest length


(b) $opp\#paths$ = Opponent's number of paths of the shortest length

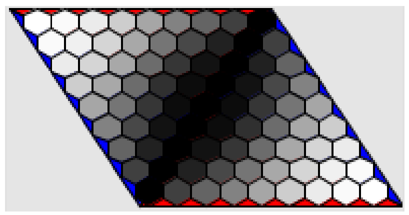

(c) $S_2 = \Delta \times (-my\#paths) + (1 - \Delta) \times opp\#paths$

Figure 7.1: Path counts found by the Adjacency Player

Random Spot

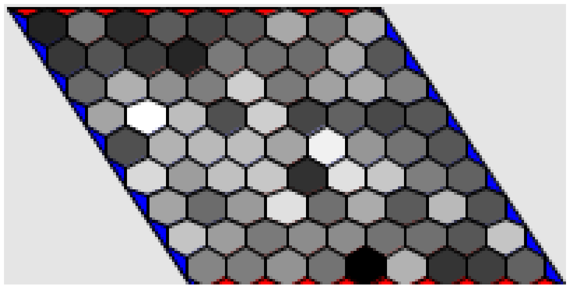


Figure 7.2: Output of Evaluation function for the Random Spot Player

Here we see the evaluation by a Random Spot Player. We have used 50,000 random fills but the results have not converged into anything useful.

Random Path

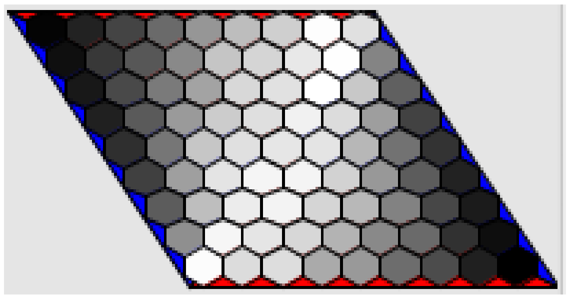


Figure 7.3: Output of Evaluation function for the Random Path Player

Here we see the evaluation by a Random Path Player. We have used 5,000 random fills and note how clear the results are compared to the Random Spot Player, even when using only one tenth of the random board fills. Although the Spot player is the theoretical optimum, this optimum cannot be attained without significant amounts of random fills and so the Random Path results seem to converge faster.

7.2 Player versus Player

For every game, we play on a 7×7 Hex board. Every round consists of two smaller games in which each of the competitors gets to play first once. To win a round, a player must beat their opponent twice, once as black and then again as white. Losing requires both games to have been lost, and a draw means that each player has won one of the two games. Each experiment lasts 10 rounds.

7.2.1 Seasonal Rules

Here we are playing with Seasonal rules in effect, with random season board patterns and a season cycle of (1,2,3) for player one and (3,1,2) for player two. These are the recommended cycles as specified in chapter 4. Unless the player is native to the game (e.g Seasonal Adjacency Player), masking of illegal moves as shown in chapter 5 will be applied to all other players.

Random Path Player (1000 fills)

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	5	2	3
Random Path Player (100 fills)	7	3	0
Random Spot Player	10	0	0
Seasonal Adjacency Player	5	4	1
Classic Adjacency Player	6	4	0
Total	33	13	4

Random Path Player (100 fills)

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	3	7
Random Path Player (100 fills)	3	5	2
Random Spot Player	9	1	0
Seasonal Adjacency Player	4	4	2
Classic Adjacency Player	3	7	0
Total	19	20	11

Random Path appears to perform best in this tournament.

Random Spot Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	0	10
Random Path Player (100 fills)	0	1	9
Random Spot Player	4	4	3
Seasonal Adjacency Player	0	0	10
Classic Adjacency Player	0	0	10
Total	4	5	42

Random Spot Player appears to perform very poorly.

Seasonal Adjacency Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	1	4	5
Random Path Player (100 fills)	2	4	4
Random Spot Player	10	0	0
Seasonal Adjacency Player	2	5	3
Classic Adjacency Player	3	6	1
Total	18	19	13

Classic Adjacency Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	4	6
Random Path Player (100 fills)	0	7	3
Random Spot Player	10	0	0
Seasonal Adjacency Player	1	6	3
Classic Adjacency Player	2	4	4
Total	13	21	16

The Seasonal Player appears to perform better than the Classic Player.

7.2.2 Random Turn Rules

Here we are playing with Random Turn rules in effect. Please note that the random choice of which player can make a move applies to the opening move as well as all subsequent moves.

Random Path Player (1000 fills)

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	1	7	2
Random Path Player (100 fills)	3	3	4
Random Spot Player	9	1	0
Seasonal Adjacency Player	6	4	0
Classic Adjacency Player	6	3	1
Total	25	18	7

Random Path Player (100 fills)

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	4	3	3
Random Path Player (100 fills)	3	4	2
Random Spot Player	9	1	0
Seasonal Adjacency Player	5	4	1
Classic Adjacency Player	4	4	2
Total	25	16	8

Random Path appears to perform best in this tournament.

Random Spot Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	1	9
Random Path Player (100 fills)	0	1	9
Random Spot Player	3	3	4
Seasonal Adjacency Player	1	0	9
Classic Adjacency Player	1	1	8
Total	5	6	39

Random Spot Player appears to perform very poorly, but not as bad as the other rules.

Seasonal Adjacency Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	4	6
Random Path Player (100 fills)	1	4	5
Random Spot Player	9	0	1
Seasonal Adjacency Player	3	3	4
Classic Adjacency Player	3	4	3
Total	16	15	19

Classic Adjacency Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	1	3	6
Random Path Player (100 fills)	2	4	4
Random Spot Player	8	1	1
Seasonal Adjacency Player	3	4	3
Classic Adjacency Player	4	3	3
Total	18	15	17

The Seasonal Player appears to perform similar to the Classic Player.

7.2.3 Classic Hex Rules

Here we are playing with the classic rules.

Random Path Player (1000 fills)

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	10	0
Random Path Player (100 fills)	4	5	1
Random Spot Player	10	0	0
Seasonal Adjacency Player	6	4	0
Classic Adjacency Player	6	4	0
Total	26	23	1

Random Path Player (100 fills)

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	1	5	4
Random Path Player (100 fills)	3	6	1
Random Spot Player	10	0	0
Seasonal Adjacency Player	5	5	0
Classic Adjacency Player	2	8	0
Total	21	24	5

Random Path appears to perform best in this tournament.

Random Spot Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	0	10
Random Path Player (100 fills)	0	0	10
Random Spot Player	3	4	3
Seasonal Adjacency Player	1	0	9
Classic Adjacency Player	0	1	9
Total	4	5	41

Random Spot Player appears to perform very poorly.

Seasonal Adjacency Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	4	6
Random Path Player (100 fills)	0	5	5
Random Spot Player	9	0	1
Seasonal Adjacency Player	2	6	2
Classic Adjacency Player	3	6	1
Total	14	21	15

Classic Adjacency Player

Opponent	Wins	Draws	Losses
Random Path Player (1000 fills)	0	4	6
Random Path Player (100 fills)	0	8	2
Random Spot Player	9	1	0
Seasonal Adjacency Player	1	6	3
Classic Adjacency Player	3	4	3
Total	13	23	14

The Seasonal Player appears to perform similar to the Classic Player.

Chapter 8

Conclusion

The Seasonal Hex game has shown that it provides unique aspects of play, especially the requirement for a deeper lookahead to estimate your opponent's next move. The 3 season pattern of (1,2,3) (2,3,1) appears to show fairness in this regard, but it should be tested with optimised AI players that take advantage of the seasonal lookahead before major conclusions can be drawn.

The randomised board season layouts appear to limit the overpowering success of the opening move, but it is not without its caveats. On rare occasions a board pattern can be generated that has significantly less areas dedicated to one season than another, which can result in a game where at some later point, a player has no area in which to legally play and so the game must be discarded.

It is interesting to see that large areas of the same season are often avoided. This is normally due to the fact that a player must wait an entire 'year' until they can play in that same season again. This can be seen as an area of "classic" hex that can only be played in once a year. Sadly it results in play that is not as nimble as the areas around it, and so is largely avoided by the players.

The ability to 'mask' an AI player's evaluation function has meant that greater amounts of experiments can be run with a larger set of players, however it does put them at a disadvantage

A main issue with the Random Turn Hex game can be classed with its playability. Although the game as an idea is fair, it can sometimes take too long for this to become apparent. It seems that even a one to two move advantage is enough to tip the game in a player's favour. When a player gets a streak of consecutive moves, it can easily have a game winning advantage. This advantage is usually taken away once the luck is given to the other player. However, if the board is too small (usually less than 11×11) the streak can never be countered, and so the player can reach an easy win. For example, a 2×2 board where a player gets a streak of 3 or more ($p = \frac{1}{8}$) will result in the player finding it impossible *not* to win.

Conversely, on large boards these streaks and counter-streaks provide surprisingly exciting gameplay, if a little shallow. It's similarities to the card game of Beggar My Neighbour [9] are notable in the way that the player has little control over the game itself, but it still features climactic situations and reversals of fortune.

Sadly the Seasonal Adjacency Player implementation did not realise every aspect of the theoretical work, namely the directed (as apposed to the classic undirected) graph structure and the seasonal information that it reveals. It must be noted that it did show promise when playing with seasonal rules by beating the Classic player.

The Random Path Player can be noted for it's speed and lack of complexity that allows it to play on large board sizes with little fuss. It can cope with far less random fills than its optimal competitor, the

Random Spot Player. The vague virtual game that it plays when performing a random fill appears to give the AI player a rough lookahead, with the search for a path in the virtual game giving a clearer idea of critical positions.

However, as mentioned in chapter 6, the evaluation function only makes use of board fills that result in a win for the player. Given an empty initial board, this means that statistically only 50% of the random fills can be used for a winning path. It can be noted that if given a game in which the player is in a much stronger position (One or two moves away from winning with multiple options), then the proportion of random board fills that result in a win is much higher. This in turn means more usable path results and so the evaluation results will converge much faster. Sadly, this means that the opposite is also true, whereby a player having to evaluate a particularly weak game state will observe fewer winning states and thus the evaluation may not have enough results to settle.

This tendency for the Random Path player to 'panic' in these poor positions means that it copes poorly trying to evaluate a board where it is loosing at a significant degree. This situation is especially troublesome when it comes to Random Turn Hex rules. As mentioned earlier, Random Turn Hex can produce games where one player has a powerful advantage over the other by making a string of consecutive moves. If the Random Path player witnesses the other player receiving this advantage, giving it in a very poor probability that a random fill will win, then the successive random fills that it performs may never produce a winning state for itself. This means that it has to base its evaluation on no results at all, which is very troubling.

8.1 Further Work

For the Seasonal Hex game, with note to the large season blocks on the board and conversely the areas where the allocated seasons are varied, it may be possible to 'sculpt' the game by giving a specific pattern of season tiles. The idea that a correct pattern on the board could give one player an advantage over the other by making a perfect path is an interesting one. It may even be possible to create a pattern so that one player is guaranteed to win by forcing the player down a predetermined route.

A point to explore in the design of a Seasonal Hex game is the structure of the seasons in the yearly cycles. We have shown how certain cycles are preferred over others, but can it be possible to create a perfect cycle algorithm that forces equal minimum lookaheads for both players at any value?

We must look into finding a solution for randomly allocating seasons to the tiles that does not result in the player running out of legal moves for a particular season, as mentioned earlier.

Sadly the directional implementation of the Seasonal Hex AI could not be completed, but the omnidirectional player that was tested showed strength that could be increased by implementing the entire theoretical system.

Given the issue of wasted random filled states, it would be interesting to see if the Random Path Player could use these to note the opposing player's winning path. The results from these could be combined in a similar to that of the Adjacency Player's scoring function in chapter 5. Although doubling the breadth first searches could be costly, Hex's lack of a draw state could be taken advantage of and only require one search per board fill.

The matter of the evaluation results converging is also another interesting point. It would be worth investigating how quickly this converges in relation to the number of random fills performed.

We may also note that an improvement to the definition of the “winning path” may be possible. By considering the path as being mere than just one route and note the convergence of branches, we may attain a more accurate result.

Appendix A

System Manual

The supplied files on the CD contain all source code and compiled Java code. To run the project from the command line, go to the dist folder and type the following:

```
  java -jar "HexGame.jar"
```

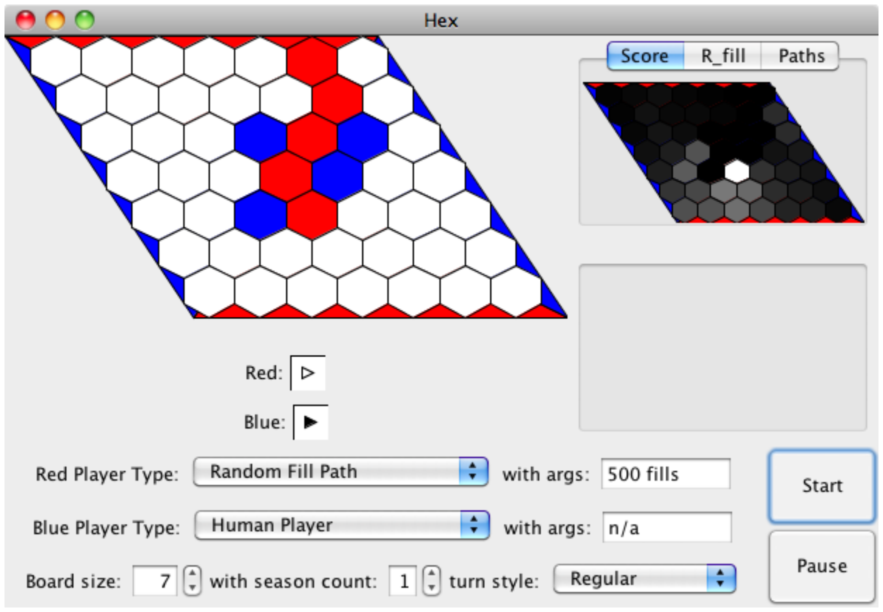
Appendix B

User Manual

To start a game, select the options for both players and the game rules, then press the “Start” button.

When playing, the game will show the season / turn loop with an arrow, where a full arrow meaning that the player is expected to play. For a human player to play, select the correct option from the menu and click where you would like to place a piece.

Tabbed images on the right can be used to monitor each player’s virtual board usage.



Red: ▷

Blue: ▶

Red Player Type: Random Fill Path with args: 500 fills

Blue Player Type: Human Player with args: n/a

Board size: 7 with season count: 1 turn style: Regular

Figure B.1: Hex Program

Appendix C

Project Plan

C.1 Name

Andrew Trevorah

C.2 Project Supervisor

Mark Herbster

C.3 Project Title

Developing and Exploring Computer-Based Random-Turn Hex

C.4 Aim

To develop and explore computer-based Random-turn Hex, Determine an evaluation function and see how it can be applied to other Hex variants.

C.5 Objectives

- Review the Hex game and other related games of perfect information, in particular those games that feature a random-turn variant.
- Develop software tools that can run hex-related algorithms and apply them to a virtual game or board layout.
- Develop an appropriate evaluation function that can use the above games/board layouts as input and return an appropriate score for a board position on which a player could place a move.
- Evaluate the success of the method by comparing it to other solutions, including those of alternate Hex variants (Regular-turn Hex, Simultaneous-play Hex etc.)

C.6 Deliverables

- Literature survey that summarises previous relevant work in Hex, Random-turn Hex, and other related Hex variants.

- A refined algorithm to analyse the board to find if a player has won a game, and more importantly, whether or not a board position was pivotal to winning.
- An evaluation function for Random-turn Hex that uses the above algorithm to correctly assess potential playable moves.
- A software system that is capable of running, testing and comparing various Hex-related algorithms.
- A design specification for the Hex software application.
- Full documentation for the software system.
- A strategy for testing and evaluating the software system.

C.7 Work Plan

Project Start to End of October (4 weeks) Literature search and review

Mid October to Mid November (4 weeks) Coding test programs and prototypes

Mid November to end of January (10 weeks) Develop software with iterative approach in parallel to developing required algorithmic solutions

End of January to mid February (2 weeks) Compare and contrast algorithmic solutions in depth

Mid February to End of March (6 weeks) Work on Final Report

Appendix D

Interim Report

D.1 Project title

Developing and Exploring Computer-Based Random-Turn Hex

D.2 Supervisor

Mark Herbster

D.3 Progress made to date

- Developed software tool that can run hex-related algorithms and apply them to a virtual game or board layout.
- Developed methods to easily visualise output output of algorithms for game layouts.
- Developed appropriate evaluation functions that can use the above games/board layouts as input and return an appropriate score for a board position for random turn hex.
- Implemented appropriate evaluation functions that can use the above games/board layouts as input and return an appropriate score for a board position for regular turn hex.
- Developed methods to run regular turn algorithms on random turn games and visa versa.
- Developed a method of allowing a user to intuitively play hex or random-turn hex against against any implemented algorithm or another local player.

D.4 Further work to be done

- Literature survey that summarises previous relevant work in Hex, Random-turn Hex, and other related Hex variants.
- Full documentation for the software system.
- Implementation of other playable variants of hex in the software tool.
- Development of additional algorithms for new hex variants.
- Expansion of existing algorithms to cope with new hex variants.

- Implementation of other hex algorithms for comparative testing.

Appendix E

Code Listings

Complete listings can be found on the CD in the src folder.

E.1 Game.java

```java
package gameMechanics;

import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import players.Player;
import players.PointAndClickPlayer;
import players.AdjPlayer;
import players.randomTurn.R_Path;
import players.randomTurn.R_Single;
import hexBoards.GameBoard;
import hexBoards.Board;
import players.AdjSeasonPlayer;

public class Game extends Thread implements Runner {

  private GameBoard board;
  private Player red;
  private Player blue;
  private int currentPlayer = Board.RED;
  private boolean finished = false;
  private boolean pause = false;
  private volatile boolean stop = false;
  private SeasonMechanics seasonPicker;
  private int gameType;
  private String commentary = "";

  public Game(int size, int type, int seasoncount, int redPlayer, String[]
     redArgs, int bluePlayer, String[] blueArgs) {
    this.seasonPicker = new SeasonMechanics(seasoncount);
    this.board = new GameBoard(size, seasonPicker);
    this.gameType = type;
    this.red = createPlayer(redPlayer, Board.RED, redArgs);
    this.blue = createPlayer(bluePlayer, Board.BLUE, blueArgs);
  }

  @Override
  public GameBoard getBoard() {
    return board;
  }

  @Override
  public void run() {

    Random coinflip = new Random();

    /*
     * Main running loop
     */
    while (!finished && !stop) {

      if (this.gameType == Runner.RANDOM_TURN)
        if (coinflip.nextBoolean() == true)
          this.currentPlayer = Board.BLUE;
        else
          this.currentPlayer = Board.RED;

      boolean moveAccepted = false;

      Move move = null;
      switch (currentPlayer) {
        case Board.RED:
          seasonPicker.thinkingPlayer(Board.RED);
          move = red.getMove();
          break;
        case Board.BLUE:
          seasonPicker.thinkingPlayer(Board.BLUE);
          move = blue.getMove();
          break;
        default:
          System.err.println("invoking mystery player");
          System.exit(1);
          break;
      }
      try {
        moveAccepted = board.makeMove(move);
      } catch (InvalidMoveException ex) {
        Logger.getLogger(Game.class.getName()).log(Level.SEVERE, null, ex);
      }
      if (!moveAccepted)
        System.out.println("Move was not accepted, passing on...");
      /*
       * move has been accepted
       */

      if (board.checkwin(currentPlayer)) {
        notifyWin(currentPlayer);
        finished = true;
      }


      switch (currentPlayer) {
        case Board.RED:
          seasonPicker.increment(Board.RED);
          this.currentPlayer = Board.BLUE;
          break;
        case Board.BLUE:
          seasonPicker.increment(Board.BLUE);
          this.currentPlayer = Board.RED;
          break;
        default:
          System.err.println("invoking␣mystery␣player");
          System.exit(1);
          break;
      }
    }
  }

  public void notifyWin(int player) {
    this.finished = true;
    java.awt.Toolkit.getDefaultToolkit().beep();
    switch (player) {
      case Board.RED:
        System.out.println("Red␣wins!");
        announce("Red␣Wins!");
        break;
      case Board.BLUE:
        System.out.println("Blue␣wins!");
        break;
    }
  }

  @Override
  public void stopGame() {
    stop = true;
    System.out.println("Stopped!");
  }

  @Override
  public void pauseSwitch() {
    if (pause)
      pause = false;
  else
    pause = true;
  System.out.println("pause = " + pause);
  red.setPause(pause);
  blue.setPause(pause);
}

@Override
public SeasonMechanics getSeasonPicker() {
  return seasonPicker;
}

private Player createPlayer(int type, int colour, String[] args) {
  Player player = null;
  switch (type) {
    case Player.R_PATH:
      player = new R_Path(this, colour, args);
      break;
    case Player.CLICK_PLAYER:
      player = new PointAndClickPlayer(this, colour);
      break;
    case Player.R_POINT:
      player = new R_Single(this, colour, args);
      break;
    case Player.ALL_PATH:
      player = new AdjPlayer(this, colour, args);
      break;
    case Player.SEASON_PATH:
      player = new AdjSeasonPlayer(this, colour, args);
      break;
    default:
      System.out.println("ERROR - no player or exception");
      break;
  }
  return player;
}

@Override
public Player getPlayerBlue() {
  return blue;
}

@Override
public Player getPlayerRed() {
  return red;
}

private void announce(String announcement) {
  this.commentary = announcement;
}
```

```
  @Override
  public String getCommentary() {
    return commentary;
  }
}
```

E.2 AdjMatrix.java

```
package hexBoards;

import java.util.ArrayList;
import no.uib.cipr.matrix.Matrix;
import no.uib.cipr.matrix.UpperSPDPackMatrix;

public class AdjMatrix {

  public static final int NO_LINK = 0;
  public static final int LINK = 1;
  private Matrix data;
  private int size;

  public AdjMatrix(int size) {
    //remember to add the border nodes as well
    this.size = size;
    this.data = new UpperSPDPackMatrix(size);
  }

  public AdjMatrix(Matrix adj) {
    this.data = adj;
    this.size = data.numColumns();
  }

  @Override
  public AdjMatrix clone() {
    return new AdjMatrix(data.copy());
  }

  public Matrix getData() {
    return this.data;
  }

  public AdjMatrix mult(AdjMatrix m2) {
    // both are assumed to be the same size
    if (m2.size() == this.size()) {
      Matrix retMatrix = new UpperSPDPackMatrix(size);
      data.mult(m2.getData(), retMatrix);
      return new AdjMatrix(retMatrix);
    } else
      return null;
  }
```

```
public void write(int a, int b, int value) {
  if (a == b)
    value = LINK;
  internalWrite(a, b, value);
}

public int read(int x, int y) {
  return (int) data.get(x, y);
}

public void wipeNode(int nodeId) {
  for (int i = 0; i < size; i++)
    if (i != nodeId)
      internalWrite(i, nodeId, NO_LINK);
}

private void internalWrite(int x, int y, int value) {
  data.set(x, y, value);
  data.set(y, x, value);
}

public int size() {
  return data.numColumns();
}

public void print() {
  for (int y = 0; y < data.numRows(); y++) {
    for (int x = 0; x < data.numColumns(); x++)
      System.out.print((int) (data.get(x, y)) + "␣");
    System.out.print("\n");
  }
}

public void bypassAndRemoveNode(int node) {
  ArrayList<Integer> neighbours = new ArrayList<Integer>();
  /*
   * populate neighbour list
   */
  for (int i = 0; i < size(); i++)
    if (read(node, i) == LINK)
      neighbours.add(i);
  /*
   * create links between all neighbours
   */
  for (int neighbour : neighbours)
    for (int newNeighbour : neighbours)
      write(neighbour, newNeighbour, LINK);
  /*
   * remove links to and from original node
   */
  wipeNode(node);
  }
}
```

E.3 SeasonMechanics.java

```
package gameMechanics;

import hexBoards.Board;
import java.awt.Color;

public class SeasonMechanics {

  private static final int[] oneSeasonPattern = {0, 0};
  private static final int[] twoSeasonPattern = {0, 0, 1, 1};
  private static final int[] threeSeasonPattern = {0, 2, 1, 0, 2, 1};
  private static final Color thinGreen = new Color(128, 255, 128);
  private static final Color thinMagenta = new Color(255, 128, 255);
  private static final Color thinYellow = new Color(255, 255, 128);
  private static final Color[] oneSeasonColours = {Color.WHITE};
  private static final Color[] twoSeasonColours = {thinYellow, thinMagenta};
  private static final Color[] threeSeasonColours = {thinGreen, thinYellow,
     thinMagenta};
  public static final int MAX_SUPPORTED_SEASON_SIZE = 3;
  public static final int MIN_SUPPORTED_SEASON_SIZE = 1;
  public static final int DEFAULT_SEASON_SIZE = 1;
  private int[] seasonPattern;
  private Color[] seasonColours;
  private int seasonCount;
  private int redPosition;
  private int bluePosition;
  private int[] redSeasonPattern;
  private int[] blueSeasonPattern;
  private boolean redThinking;
  private boolean blueThinking;

  public SeasonMechanics(int numberOfSeasons) {

    this.seasonCount = numberOfSeasons;
    switch (seasonCount) {
      case 1:
        seasonPattern = oneSeasonPattern;
        seasonColours = oneSeasonColours;
        break;
      case 2:
        seasonPattern = twoSeasonPattern;
        seasonColours = twoSeasonColours;
        break;
      case 3:
        seasonPattern = threeSeasonPattern;
        seasonColours = threeSeasonColours;
        break;
      default:
        System.err.println("invalid season count");
        break;


    }

    redSeasonPattern = getSubPattern(Board.RED);
    blueSeasonPattern = getSubPattern(Board.BLUE);

    redPosition = 0;
    bluePosition = 0;
  }

  public int getSeasonCount() {
    return seasonCount;
  }

  public int getAdvanceSeason(int player, int advance) {
    int pos = (getCurrentPos(player) + advance) % seasonCount;
    return getSubPattern(player)[pos];
  }

  public int getCurrentSeason(int player) {
    int season = 0;
    switch (player) {
      case Board.RED:
        season = redSeasonPattern[redPosition];
        break;
      case Board.BLUE:
        season = blueSeasonPattern[bluePosition];
        break;
    }
    return season;
  }

  public int getCurrentPos(int player) {
    int pos = 0;
    switch (player) {
      case Board.RED:
        pos = redPosition;
        break;
      case Board.BLUE:
        pos = bluePosition;
        break;
    }
    return pos;
  }

  public void increment(int player) {
    switch (player) {
```

```
    case Board.BLUE:
      bluePosition = (bluePosition + 1) % blueSeasonPattern.length;
      break;
    case Board.RED:
      redPosition = (redPosition + 1) % redSeasonPattern.length;
      break;
  }
}

public boolean isThinking(int player) {
  switch (player) {
    case Board.BLUE:
      return blueThinking;
    case Board.RED:
      return redThinking;
    default:
      System.err.println(”player␣id␣” + player + ”␣does␣not␣exist!”);
      return false;
  }
}

public void thinkingPlayer(int player) {
  switch (player) {
    case Board.BLUE:
      System.out.println(”blue␣is␣thinking”);
      blueThinking = true;
      redThinking = false;
      break;
    case Board.RED:
      System.out.println(”red␣is␣thinking”);
      blueThinking = false;
      redThinking = true;
      break;
    default:
      System.out.println(”what!!!!”);
      break;
  }
}

public Color[] getColourArray() {
  return seasonColours;
}

public int getNextSeason(int currentSeason, int player) {
  int returnSeason = 0;
  int[] tempSeasons = getSubPattern(player);
  for (int i = 0; i < tempSeasons.length; i++)
    if (tempSeasons[i] == currentSeason) {
      int nextPos = (i + 1) % tempSeasons.length;
      returnSeason = tempSeasons[nextPos];
    }
    return returnSeason;
  }

  public int[] getSubPattern(int player) {
    int[] sublist = new int[seasonPattern.length / 2];
    int startPos;
    if (player == Board.RED)
      startPos = 0;
    else
      startPos = 1;
    for (int i = 0; i < sublist.length; i++)
      sublist[i] = seasonPattern[(i) * 2 + startPos];
    return sublist;
  }
}
```

E.4 R_Path.java

```
package players.randomTurn;

import hexBoards.Board;
import gameMechanics.Runner;
import hexBoards.OpenBoard;
import java.awt.Point;
import java.util.ArrayList;
import hexBoards.GameBoard;
import gameMechanics.Move;
import hexBoards.ScoreBoard;
import java.util.Random;
import players.AbstractPlayer;
import players.Player;

public class R_Path extends AbstractPlayer {

  private final ScoreBoard scoreBoard;
  private final GameBoard mainBoard;
  private final OpenBoard randomFillBoard;
  private final OpenBoard paths;
  private final int sampleLimit;

  public R_Path(Runner game, int colour, String[] args) {
    super(game, colour, args);
    int parsedVal = 0;

    if (args != null && args.length > 0)
      parsedVal = Integer.parseInt(args[0]);

    if (parsedVal <= 0)
      parsedVal = Integer.parseInt(Player.R_DEFAULT_ARGS);
    this.sampleLimit = parsedVal;
    System.out.println("sample␣limit␣set␣at␣" + sampleLimit);
  this.scoreBoard = new ScoreBoard(size);
  this.mainBoard = game.getBoard();
  this.randomFillBoard = mainBoard.openClone();
  this.randomFillBoard.setName("R_fill");
  this.paths = mainBoard.openClone();
  this.paths.setName("Paths");

  auxBoards.add(scoreBoard);
  auxBoards.add(randomFillBoard);
  auxBoards.add(paths);

}
boolean first = true;

@Override
public Move getMove() {
  this.scoreBoard.wipeAll();


  for (int sampleCount = 0; sampleCount < sampleLimit; sampleCount++) {
    while (pause)
      try {
        Thread.sleep(50);
      } catch (InterruptedException e) {

        e.printStackTrace();
      }
    randomFillBoard.setBoard(mainBoard.openClone());
    paths.setBoard(mainBoard.openClone());

    randomFillBoard.randomFill();

    ArrayList<Point> path = randomFillBoard.getData().getWinningPath(player)
        ;

    for (Point p : path) {
      int x = p.x;
      int y = p.y;
      paths.set(x, y, player);
      if (mainBoard.get(x, y) == Board.BLANK && mainBoard.getSeason(x, y) ==
          game.getSeasonPicker().getCurrentSeason(player))
        scoreBoard.raiseScore(x, y);
    }
  }

  Point chosen = getTopScore();
  return new Move(this.player, chosen.x, chosen.y);
}

protected Point getTopScore() {
    Random random = new Random();
    Point chosen = new Point();
    do {
      chosen = new Point(random.nextInt(size), random.nextInt(size));
    } while (!(mainBoard.get(chosen.x, chosen.y) == Board.BLANK && mainBoard.
        getSeason(chosen.x, chosen.y) == game.getSeasonPicker().
        getCurrentSeason(player)));


    for (int row = 0; row < size; row++)
      for (int column = 0; column < size; column++)
        if ((scoreBoard.get(column, row) > scoreBoard.get(chosen.x, chosen.y))
                && mainBoard.get(column, row) == Board.BLANK) {
          chosen.x = column;
          chosen.y = row;
        }
    return chosen;
  }
}
```

E.5 R_Single.java

```
package players.randomTurn;

import gameMechanics.Runner;
import hexBoards.GameBoard;
import hexBoards.OpenBoard;
import java.awt.Point;
import hexBoards.Board;
import gameMechanics.Move;
import hexBoards.ScoreBoard;
import java.util.Random;
import players.AbstractPlayer;
import players.Player;

public class R_Single extends AbstractPlayer {

  private final int sampleLimit;
  private final ScoreBoard scoreBoard;
  private final GameBoard mainBoard;
  private final OpenBoard randomFillBoard;

  public R_Single(Runner game, int colour, String[] args) {
    super(game, colour, args);
    int parsedVal = 0;

    if (args != null && args.length > 0)
      parsedVal = Integer.parseInt(args[0]);

    if (parsedVal <= 0)
      parsedVal = Integer.parseInt(Player.R_DEFAULT_ARGS);
    this.sampleLimit = parsedVal / (int) (Math.pow(size, 2));
    System.out.println("sample␣limit␣set␣at␣" + sampleLimit + "␣per␣position")
        ;

    this.scoreBoard = new ScoreBoard(size);
    this.mainBoard = game.getBoard();
    this.randomFillBoard = mainBoard.openClone();
    this.randomFillBoard.setName("R_fill");

    auxBoards.add(scoreBoard);
    auxBoards.add(randomFillBoard);
  }

  @Override
  public Move getMove() {

    this.scoreBoard.wipeAll();

    for (int y = 0; y < size; y++)
      for (int x = 0; x < size; x++)
        if (mainBoard.get(x, y) == Board.BLANK && mainBoard.getSeason(x, y) ==
            game.getSeasonPicker().getCurrentSeason(player))
          for (int sampleCount = 0; sampleCount < sampleLimit; sampleCount++)
            {
            randomFillBoard.setBoard(mainBoard.openClone());
            randomFillBoard.set_noNewLinks(x, y, player);
            randomFillBoard.randomFill();

            if (randomFillBoard.getData().getWinningPath(player).size() > 0)
              scoreBoard.raiseScore(x, y);

          }
    Point chosen = getTopScore();
    return new Move(this.player, chosen.x, chosen.y);
  }

  protected Point getTopScore() {
    Random random = new Random();
    Point chosen = new Point();
    do {
      chosen = new Point(random.nextInt(size), random.nextInt(size));
    } while (!(mainBoard.get(chosen.x, chosen.y) == Board.BLANK && mainBoard.
        getSeason(chosen.x, chosen.y) == game.getSeasonPicker().
        getCurrentSeason(player)));


    for (int row = 0; row < size; row++)
      for (int column = 0; column < size; column++)
        if ((scoreBoard.get(column, row) > scoreBoard.get(chosen.x, chosen.y))
                && mainBoard.get(column, row) == Board.BLANK) {
```

```
        chosen.x = column;
        chosen.y = row;
      }
    return chosen;
  }
}
```

E.6 AdjPlayer.java

```
package players;

import gameMechanics.Runner;
import java.awt.Point;
import java.util.Random;
import hexBoards.AdjMatrix;
import hexBoards.Board;
import hexBoards.BoardData;
import hexBoards.GameBoard;
import gameMechanics.Move;
import hexBoards.OpenBoard;
import hexBoards.ScoreBoard;

public class AdjPlayer extends AbstractPlayer {

  protected GameBoard mainBoard;
  protected OpenBoard boardClone;
  protected ScoreBoard myPathlength;
  protected ScoreBoard oppPathlength;
  protected ScoreBoard myPathCount;
  protected ScoreBoard oppPathCount;
  protected double aggro = 0.5;
  int myBorder1;
  int myBorder2;
  int oppBorder1;
  int oppBorder2;
  protected ScoreBoard comboLengths;
  protected ScoreBoard comboPathCounts;
  protected int maxPathLength;
  Random random = new Random();

  public AdjPlayer(Runner game, int colour, String[] args) {
    super(game, colour, args);
    this.maxPathLength = size * size;
    this.mainBoard = game.getBoard();
    this.boardClone = mainBoard.openClone();
    this.myPathlength = new ScoreBoard(size);
    this.oppPathlength = new ScoreBoard(size);
    this.myPathCount = new ScoreBoard(size);
    this.oppPathCount = new ScoreBoard(size);
    this.comboLengths = new ScoreBoard(size);
    this.comboPathCounts = new ScoreBoard(size);
    boardClone.setName("clone");
    myPathlength.setName("my_Plength");
    oppPathlength.setName("opp_Plength");
    myPathCount.setName("my_Pcount");
    oppPathCount.setName("opp_Pcount");
    comboLengths.setName("combo_Plength");
    comboPathCounts.setName("combo_Pcount");

    auxBoards.add(boardClone);
    auxBoards.add(myPathlength);
    auxBoards.add(oppPathlength);
    auxBoards.add(myPathCount);
    auxBoards.add(oppPathCount);
    auxBoards.add(comboLengths);
    auxBoards.add(comboPathCounts);

    switch (colour) {
      case Board.RED:
        myBorder1 = BoardData.RED_BORDER1_NODE;
        myBorder2 = BoardData.RED_BORDER2_NODE;
        oppBorder1 = BoardData.BLUE_BORDER1_NODE;
        oppBorder2 = BoardData.BLUE_BORDER2_NODE;
        break;
      case Board.BLUE:
        myBorder1 = BoardData.BLUE_BORDER1_NODE;
        myBorder2 = BoardData.BLUE_BORDER2_NODE;
        oppBorder1 = BoardData.RED_BORDER1_NODE;
        oppBorder2 = BoardData.RED_BORDER2_NODE;
        break;
    }
  }

  @Override
  public Move getMove() {
    /*
     * wipe all scoreboards
     */
    myPathlength.wipeAll();
    oppPathlength.wipeAll();
    myPathCount.wipeAll();
    oppPathCount.wipeAll();
    comboLengths.wipeAll();
    comboPathCounts.wipeAll();

    for (int y = 0; y < size; y++)
      for (int x = 0; x < size; x++)
        if (game.getBoard().get(x, y) == Board.BLANK) {

          /*
           * player related scores
```

```
         */
        int score = 0;
        boardClone.setBoard(mainBoard.openClone());
        boardClone.set(x, y, player);
        AdjMatrix base = boardClone.getData().getAdjMatrix(player).clone();

        while (!(base.read(myBorder1, myBorder2) > 0) && score <
            maxPathLength) {
          base = base.mult(boardClone.getData().getAdjMatrix(player));
          score++;
        }

        myPathlength.set(x, y, score);
        myPathCount.set(x, y, base.read(myBorder1, myBorder2));

        /*
         * opponant related scores
         */
        score = 0;
        boardClone.setBoard(mainBoard.openClone());

        boardClone.set(x, y, player);
        base = boardClone.getData().getAdjMatrix(opponent).clone();

        while (!(base.read(oppBorder1, oppBorder2) > 0) && score <
            maxPathLength) {
          base = base.mult(boardClone.getData().getAdjMatrix(opponent));
          score++;
        }

        oppPathlength.set(x, y, score);
        oppPathCount.set(x, y, base.read(oppBorder1, oppBorder2));

        /*
         * combine scores...
         */
        score = (int) (aggro * myPathlength.get(x, y) - (1 - aggro) *
            oppPathlength.get(x, y));
        comboLengths.set(x, y, score);

        score = (int) (aggro * (-myPathCount.get(x, y)) + (1 - aggro) *
            oppPathCount.get(x, y));
        comboPathCounts.set(x, y, score);
      }

  Point winner = pickRandom();

  int winx = winner.x;
  int winy = winner.y;
```

```
    int winScore = comboLengths.get(winx, winy);


    for (int y = 0; y < size; y++)
      for (int x = 0; x < size; x++)
        // if empty space
        if (mainBoard.get(x, y) == Board.BLANK && mainBoard.getSeason(x, y) ==
            game.getSeasonPicker().getCurrentSeason(player))
          if (beats(x, y, winx, winy)) {
            winx = x;
            winy = y;

          }

    return new Move(player, winx, winy);
  }

  protected Point pickRandom() {
    int x = 0;
    int y = 0;
    do {
      x = random.nextInt(this.size);
      y = random.nextInt(this.size);
    } while (mainBoard.get(x, y) != Board.BLANK || mainBoard.getSeason(x, y)
        != game.getSeasonPicker().getCurrentSeason(player));
    return new Point(x, y);
  }

  @Override
  public void setPause(boolean pause) {
  }

  protected boolean beats(int x, int y, int winningX, int winningY) {
    int length_compettitor = comboLengths.get(x, y);
    int length_champion = comboLengths.get(winningX, winningY);
    if (length_compettitor < length_champion)
      return true;
    else if (length_compettitor == length_champion) {
      int path_competitor = comboPathCounts.get(x, y);
      int path_champion = comboPathCounts.get(winningX, winningY);
      if (path_competitor < path_champion)
        return true;
      else if (path_competitor == path_champion)
        return random.nextBoolean();

    }
    return false;

  }
}
```

E.7 AdjSeasonPlayer.java

```
package players;

import gameMechanics.Move;
import gameMechanics.Runner;
import hexBoards.AdjMatrix;
import hexBoards.Board;
import java.awt.Point;

public class AdjSeasonPlayer extends AdjPlayer {

  public AdjSeasonPlayer(Runner game, int colour, String[] args) {
    super(game,colour,args);
  }

  @Override
  public Move getMove() {
    /*
     * wipe all scoreboards
     */
    myPathlength.wipeAll();
    oppPathlength.wipeAll();
    myPathCount.wipeAll();
    oppPathCount.wipeAll();
    comboLengths.wipeAll();
    comboPathCounts.wipeAll();

    for (int y = 0; y < size; y++) {
      for (int x = 0; x < size; x++) {
        if (game.getBoard().get(x, y) == Board.BLANK && mainBoard.getSeason(x,
            y) == game.getSeasonPicker().getCurrentSeason(player)) {

          /*
           * player related scores
           */
          int score = 0;
          boardClone.setBoard(mainBoard.openClone());

          int future = 0;
          int season = game.getSeasonPicker().getAdvanceSeason(player, future)
              ;
          boardClone.set(x, y, player);
          AdjMatrix base = boardClone.getData().getAdjMatrix(player, season);

          season = game.getSeasonPicker().getAdvanceSeason(player, 1);

          while (!(base.read(myBorder1, myBorder2) > 0) && score <
              maxPathLength) {
            future++;
            season = game.getSeasonPicker().getAdvanceSeason(player, future);
  base = base.mult(boardClone.getData().getAdjMatrix(player,season))
      ;
  score++;
}

if(score >= maxPathLength-1)
  System.out.println("my max path length reached");

myPathlength.set(x, y, score);
myPathCount.set(x, y, base.read(myBorder1, myBorder2));

/*
 * opponant related scores
 */
score = 0;
boardClone.setBoard(mainBoard.openClone());
boardClone.set(x, y, player);


future = 0;
season = game.getSeasonPicker().getAdvanceSeason(opponent, future);
base = boardClone.getData().getAdjMatrix(opponent, season);

while (!(base.read(oppBorder1, oppBorder2) > 0) && score <
    maxPathLength) {
  future++;
  season = game.getSeasonPicker().getAdvanceSeason(opponent, future)
      ;
  base = base.mult(boardClone.getData().getAdjMatrix(opponent,season
      ));
  score++;
}

if(score >= maxPathLength-1)
  System.out.println("opp max path length reached");

oppPathlength.set(x, y, score);
oppPathCount.set(x, y, base.read(oppBorder1, oppBorder2));

/*
 * combine scores...
 */

score = (int) (aggro * myPathlength.get(x, y) - (1 - aggro) *
    oppPathlength.get(x, y));
comboLengths.set(x, y, score);

score = (int) (aggro * (-myPathCount.get(x, y)) + (1 - aggro) *
    oppPathCount.get(x, y));
comboPathCounts.set(x, y, score);
```

```
        }
      }
    }

    Point winner = pickRandom();

    int winx = winner.x;
    int winy = winner.y;

    int winScore = comboLengths.get(winx, winy);


    for (int y = 0; y < size; y++) {
      for (int x = 0; x < size; x++) {
        // if empty space
        if (mainBoard.get(x, y) == Board.BLANK && mainBoard.getSeason(x, y) ==
            game.getSeasonPicker().getCurrentSeason(player)) {
          if (beats(x,y,winx,winy)) {
             winx = x;
            winy = y;

          }
        }
      }
    }
    return new Move(player, winx, winy);
  }
}
```

Bibliography

[1] H. W. Kuhn and S. Nasar, eds., The Essential John Nash, Princeton University Press, Princeton, 2002.

[2] Yuval Peres, Oded Schramm, Scott Sheffield, and David B. Wilson, Random-Turn Hex and Other Selection Games. 2006

[3] D Gale, The game of Hex and the Brouwer fixed-point theorem. American Mathematical Monthly, 1979.

[4] J Yang, S Liao, M Pawlak New Winning and Losing Positions for 7x7 Hex 2003

[5] Zuoshu Lu Tuning the evaluation function for the game of Hex 2008

[6] Alex Lee Developing an evaluation function for computer Hex 2004

[7] Stefan Reisch, Hex ist PSPACE-vollständig, 1980.

[8] Computer Olympiad Hex Tournament.

[9] Marc Paulhus, Beggar My Neighbour. 1999