Exact and Monte-Carlo algorithms for combinatorial games

Anders Leino

February 24, 2014

Abstract

This thesis concerns combinatorial games and algorithms that can be used to play them. Basic definitions and results about combinatorial games are covered, and an implementation of the minimax algorithm with alpha-beta pruning is presented. Following this, we give a description and implementation of the common UCT (Upper Confidence bounds applied to Trees) variant of MCTS (Monte-Carlo tree search). Then, a framework for testing the behavior of UCT as first player, at various numbers of iterations (namely 2,7, . . . 27), versus minimax as second player, is described. Finally, we present the results obtained by applying this framework to the 2.2 million smallest non-trivial positional games having winning sets of size either 2 or 3. It is seen that on almost all different classifications of the games studied, UCT converges quickly to near-perfect play.

Supervisor:
Klas Markström
Department of Mathematics and Mathematical Statistics
Umeå University
Exakta och Monte-Carlo algoritmer för kombinatoriska spel

Denna rapport handlar om kombinatoriska spel och algoritmer som kan användas för att spela dessa. Grundläggande definitioner och resultat som berör kombinatoriska spel täcks, och en implementation av minimax-algoritmen med alpha-beta beskärning ges. Detta följs av en beskrivning samt en implementation av UCT varianten av MCTS (Monte-Carlo tree search). Sedan beskrivs ett ramverk för att testa beteendet för UCT som första spelare, vid olika antal iterationer (nämligen 2, 7, ... 27), mot minimax som andra spelare. Till sist beskrivs resultaten vi funnit genom att använda detta ramverk för att spela de 2,2 miljoner minsta icke triviala positionella spelen med vinstmängder av storlek antingen 2 eller 3. Vi finner att, för nästan alla olika klassificeringar av spel vi studerar, så konvergerar UCT snabbt mot nära perfekt spel.
## Contents

1 **Introduction** 4  
   1.1 What is a combinatorial game? 4  
      1.1.1 Examples of combinatorial games 4  
   1.2 Advances in combinatorial game theory 5  
      1.2.1 Nim-like games and Sprague-Grundy theory 5  
      1.2.2 Probabilistic approach to game theory 5  

2 **Positional games** 5  
   2.1 Definitions 5  
      2.1.1 Reverse games 7  
      2.1.2 Weak games 7  
      2.1.3 Reverse weak games 8  
   2.2 The game tree 9  

3 **Optimal play** 10  
   3.1 Introduction 10  
   3.2 Optimal play 11  
   3.3 The minimax algorithm 11  
   3.4 Alpha-Beta Pruning 14  

4 **MCTS** 16  
   4.1 Introduction 16  
   4.2 MCTS in general 16  
   4.3 The UCT algorithm 17  
      4.3.1 The selection step 17  
      4.3.2 The exploration step 17  
      4.3.3 The backup step 18  
   4.4 An example 18  
   4.5 UCT implementation 19  

5 **Generating games** 23  
   5.1 Introduction and basic definitions 23  
   5.2 Nauty and hypergraphs 23  
   5.3 Choosing the command line switches 24
1 Introduction

1.1 What is a combinatorial game?

Game theory in general is a large field. To give an idea about what kinds of games people have studied, here are two orthogonal categorizations.

Firstly there is the chance versus skill aspect. There are games of pure chance and no skill, games of some chance and some skill, and games of pure skill. Combinatorial games are games of pure skill and no chance, and Poker is a game of some chance and some skill.

Secondly, there are games of complete information and games of incomplete information. Poker is a game of incomplete information, and combinatorial games are games of complete information.

We will be concerned only with combinatorial games played by two players.

1.1.1 Examples of combinatorial games

The game of Nim is a good example of a combinatorial game. Nim is a game for two players, and can be played using nothing more than a bunch of beans (or other small objects). The game is played by arranging the beans in a number of heaps. The players take turns performing moves. A move is made by selecting a heap, and then taking a number of beans from that heap only. The winner is the last person to take a bean.

Another very popular combinatorial game is checkers. (Also known as draughts.) Checkers is played by two players on a checkerboard of some size, say 8-by-8 or 12-by-12 squares. Each player starts out with pieces of a given color in a certain pattern on his side of the board. The players then take turns performing moves on their own pieces. A move consists in selecting a piece of one’s own color, and moving it to an adjacent location. If the opponent has a piece in such a location, it may be captured by jumping over it to the next adjacent location in the same direction, in which case it is removed from the board. If a move captured an opponent piece, it may be extended in case it can capture more of the opponents pieces in the same way, in a subsequent move. A piece may also become a so-called “king”, in which case it can move in more ways.

A good introduction to combinatorial game theory is [Winning Ways, 2001-2004].
1.2 Advances in combinatorial game theory

For a given combinatorial game, it is quite easy to convince oneself that it would be possible, in principle, to exhaustively enumerate every single possible play of that game. In a few of those plays, one player may act ideally in all situations, and in even fewer, both players will act ideally in all situations.

However, playing by investigating all possible plays will quickly introduce a player to the concept of **combinatorial chaos** – the number of possible ways of playing the game, though finite, can be enormous. Therefore it is natural to wonder if one can find an algorithm with reasonable computational complexity, to play a given game perfectly.

Several results along these lines are presented in [Demaine, Hearn]. For instance: on page 10 we learn that checkers has been determined to be a draw if both players play perfectly, but that playing perfectly is a hard computational problem.

1.2.1 Nim-like games and Sprague-Grundy theory

It should be mentioned that, in spite of combinatorial chaos, some very nice results have been found for a certain class of games – so-called “Nim-like” games. In the 1930’s, Sprague and Grundy both independently showed that **impartial games** are equivalent to Nim. An impartial game, roughly, is a game where the players can both make the same moves, i.e. where the allowable moves at any moment only depends on what configuration the game is in, and not on who’s turn it is to move. This is a big discovery because Nim can be considered a solved game – at any moment in a game of Nim, it is possible to quickly say which player would win in perfect play.

1.2.2 Probabilistic approach to game theory

An interesting approach to tackling combinatorial chaos by means of probabilistic methods is presented in [Beck, 2008].

2 Positional games

2.1 Definitions

A positional game is a particular kind of combinatorial game. To define a **positional game**, or **strong game**, or simply **game** from now on, we need a few things. First of all, we need a “board”, $V$, which is just a finite set, as well as a collection of **winning sets**, $\mathcal{F} \subset \mathcal{P}(V)$. The tuple $(V, \mathcal{F})$ constitutes a **hypergraph**. The elements of the
set $V$ are sometimes called the *vertices* and the elements of the set $\mathcal{F}$ are sometimes called the *hyperedges* of the hypergraph $(V, \mathcal{F})$.

**Remark** We often refer to $(V, \mathcal{F})$ as the game.

The idea is that two players, called *First* and *Second* take turns coloring uncolored vertices of the board. Initially, the entire board, $V$, starts out with all vertices uncolored. The object of the game is to be the first to color an entire winning set. The players are named as they are because player First has the benefit of getting the first move.

Note that a vertex which has already been colored cannot be colored again. The word *play* is meant to represent an instance of a correctly played game from start to finish. A given point of the play is called a *position* of the board. More precisely, a position is a (partial or complete) two-coloring of $V$. A *draw* happens when the board is fully occupied, yet neither player occupies completely a winning set.

As an example of a game, look at table 1. It shows an entire play for the game Hex, played on a 3x3 board. The goal is to connect one horizontal side with the opposing one. First wins because he connects the two sides before Second.

Table 1: 3x3 Hex, First wins
Table 2: Reverse 3x3 Hex, Second wins

2.1.1 Reverse games

The reverse of a given game is obtained if the desired outcome is to avoid occupying completely the winning sets from $\mathcal{F}$. In table 2, we can see a play of reverse Hex 3x3. This time, Second wins.

2.1.2 Weak games

In the above definition of a a game, both players (First and Second), strive to occupy the same winning sets, given by $\mathcal{F}$. A player might be interested in settling for a draw. (For instance, if Second knows that he cannot win.) Thus, we have two players: one player is the Maker, and one is the Breaker.

We say that Maker wins if he manages to occupy completely one of the winning sets in $\mathcal{F}$, and Breaker wins if he manages to prevent maker. The notion of who is
“first to win” is moot: a player either wins or doesn’t win. This game is called a the weak version of the original game, or the corresponding Maker-Breaker game. Note that a draw is impossible in a Maker-Breaker game.

Table 3 shows an example of weak 3x3 Hex being played, with First as Maker and Second as Breaker. First (Maker) wins, since he manages to connect the two horizontal sides.

2.1.3 Reverse weak games

The notion of a reverse weak game should now be intuitively clear. The idea is to start with a game, get the corresponding weak game and then reverse that. However, since a weak game is technically a not a game, we should give an explicit definition of what reverse means in reference to a weak game. Suppose we have a game with the hypergraph \((V, \mathcal{F})\). The corresponding weak game has a Maker and a Breaker. The reverse weak game has a player trying to avoid making, and a player trying to avoid breaking. That is: we have an Avoider and an Enforcer. Avoider tries to avoid occupying completely a wining set from \(\mathcal{F}\), and Enforcer tries to prevent Avoider from doing so, that is, tries to enforce Avoider into occupying completely a winning set.
In table 4, we can see an example of reverse weak 3x3 Hex being played. Second (Enforcer) wins, since First has connected the two vertical edges.

### 2.2 The game tree

A good conceptual tool when reasoning about positional games is the so-called *game tree*, corresponding to a given game.

The root node of the game tree is the starting position of the game. That is to say, it corresponds to the empty board of the game. The children of the root node are all possible positions of the board after First has made his first move. The children of *those* nodes are the possible positions after Second has made his move, and so on. Since we are only considering games played on finite boards, the game tree is finite as well, but can be extremely large.
Table 5: A part of the game tree for 3x3 Tic-Tac-Toe

In table 5 you can see an example of a portion of the game tree for 3x3 Tic-Tac-Toe, below a certain position which has been chosen as the root node. Tic-Tac-Toe is a positional game played on a square board of size 3x3, in this case. The winning sets are the rows and the columns, as well as the two diagonals. The large white squares are not part of the tree; they are only in the picture to provide visual balance. The positions for a move are considered from left to right, bottom to top, on the game board. The first free position encountered when traversing the board in this manner is chosen.

3 Optimal play

3.1 Introduction

In this section, we will introduce an algorithm called the minimax algorithm, which can play any positional game (and more general games) in an optimal manner. The natural structure on which algorithmic play takes place is the game tree, as described in section 2.2. A playing algorithm can then be seen as a search algorithm on the game tree.

The minimax algorithm in its most naive implementation will search through the entire game tree. This is not feasible for most games. We therefore also describe a common optimization of the minimax algorithm, called alpha-beta pruning.
Most of this material is covered in [Russell, Norvig], though with some of the propositions given as exercises and with more imperative-style pseudo code.

### 3.2 Optimal play

What do we mean by “optimal play”? In this section, we will give some necessary conditions. Clearly, a player cannot play optimally if he squanders an opportunity to win. More precisely; if the current position is a win for the moving player, then he must make a choice which is also a win for him. Furthermore, if the moving player does not have any opportunity to win in the given position (i.e. none of the leaf-nodes in the tree below the current node contains a winning node for the moving player), but if he does have an opportunity to tie, he must still have an opportunity to tie after he makes his move.

If a player can play as described above, he will win if he can win, and if he can’t win but he can tie, he will do that.

### 3.3 The minimax algorithm

In this section, we will see how the considerations in the previous sections guide us to an algorithm which leads to optimal play. It is intuitively clear that it is always possible to play as described in the previous section. The key in order to find an explicit algorithm is to extend the notion of First player win to be not just for leaf nodes.

In terms of the game tree, what might it mean for any given position, not just a leaf node, to be a First player win? The definition is inductive. In the base case, i.e. we have a leaf node, the definition of winning node is clear from the rules of the particular game. If we are not on a leaf node, we break the definition up into two cases.

**Definition 3.1 (First player win node)**

- **Case 0**: We are on a leaf node. In this case, the rules of the particular game tell us who is the winner, or if there’s a tie.
- **Case 1**: It is First’s turn to move. In this case, the position is a First win if it has a child which is a First win, according to case 0 or case 2.
- **Case 2**: It is Second’s turn to move. In this case, the position is a First win if all of its children are First win, according to case 0 or case 1.
The notion of a position being a Second win is defined similarly. It is easy to see that if a position is a First win, it cannot be a Second win, and vice-versa. This does not mean that a position must be either a First win or a Second win; and we call such positions Neither win positions. We have now defined a kind of coloring of any game tree, with three different colors: First win, Second win and Neither win. Instead saying “the color of a node”, it is more natural to speak of “the winner of a node”, which can be either First, Second or Neither.

The following two propositions follow directly from the definition of optimal play, outlined in section 3.2.

**Proposition 3.2** If the game is in a First win position and First plays optimally, then the path in the tree represented by a play will consist entirely of First win nodes. (And similarly for Second.)

**Proposition 3.3** If the game is in a Neither win position and First plays optimally, then the play path does not contain nodes where Second wins. (And similarly for Second.)

These two propositions are summed up in [Russell, Norvig] (page 197, exercise 5.7). Note that, in the second proposition, the play path might also end with a sequence of nodes of First win color unless Second also plays optimally.

Finally, as a simple corollary of the above two propositions:

**Proposition 3.4** If both players play optimally, then the nodes in the play path will all have the same winner (First, Second or Neither).

From the above results, we have the following nice re-characterization of definition 3.1.

**Theorem 3.5** The winner for a given node is the same as the winner of a leaf node (end position) which results when both players play optimally.

To get something a bit more operational, we define an ordering on the set of colors:

Second win < Neither win < First win

(1)

**Definition 3.6 (Optimal play)** First is said to play optimally if he at all times makes choices which are maximal with respect to the above ordering. Similarly, Second plays optimally if he at all times makes choices which are minimal with respect to the same ordering.
Thus, to find the winner of a position in which it’s First’s time to move, we take the maximum of its children, according to the above ordering. To find the winner of a position in which it’s Second’s time to move, we take the minimum. This leads to the following recursive definition for the winner of a node, which can be used to play optimally:

```
Listing 1: Winner function (no pruning)
winner position =
  | terminal position = terminalWinner position
  | turn position == First =
    maximum $ map winner $ choices position
  | otherwise =
    minimum $ map winner $ choices position
```

All code examples are written in the Haskell\(^1\) programming language. The `minimum` and `maximum` functions know about the ordering defined in inequality 1. Since we know the rules of the game, the functions `turn`, `choices` and `terminal` are assumed to be given. (In turn, they tell who’s turn it is, what his choices are and whether the position is terminal, given any position of the game.)

The code breaks the definition of `winner` up into three cases. In the first case, where the position is terminal, we know the winner from the definition of the game. This is what the `terminalWinner` function computes. In the second case, where the position is not terminal and it is First’s turn to play, the winner is the maximum of `winner` mapped over the choices available from the given position. In the final case, we know that the node is not terminal and it is not First’s turn to move. Thus it is Second’s turn to move, and so the winner is the minimum of `winner` mapped over the choices available from the given position.

With the above function in hand, it is easy for either player to play optimally. The actual code used to run the experiments later on is not quite this simple: as mentioned there’s a common optimization which we will take advantage of. Namely alpha-beta pruning.

\(^1\)Resources for learning Haskell can be found at [http://www.haskell.org/](http://www.haskell.org/). For a precise specification of the language, see [Simon Peyton Jones, 2003](http://www.haskell.org/onlinereport/haskell2010/).
3.4 Alpha-Beta Pruning

Suppose that it is First’s turn to move, and we are evaluating maximum function in code listing 1, and we run into a child which is a First win. We then know that we can stop searching because we cannot do any better than that. Similarly, if it is Second’s turn to move, and we are evaluating the minimum function, and we run into a Second win, we can also stop prematurely.

This is a specific case of alpha-beta pruning, and will allow us to disregard (or prune) big parts of the game tree as we search it. There is a generalization where the leaf node can have a bigger set of values than just First win, Second win or Neither win, but we don’t need it here. The implementation is particularly easy in Haskell. We can even make it look exactly like the implementation above, but we need to take some special care when writing min and max so that they know what the absolute minimum is (Second win) and what the absolute maximum is (First win) and can therefore prune.

This means that we can find the maximum without actually computing all elements in the list (which would require searching the entire game tree below the current position).

Here is our version of minimum which will allow pruning:

```haskell
prunedMinimum ws =
    case find ((==) Only Second) ws of
        Nothing -> minimum ws
        _      -> Only Second
```

The implementation for prunedMaximum is similar:

```haskell
prunedMaximum ws =
    case find ((==) Only First) ws of
        Nothing -> maximum ws
        _      -> Only First
```

Now we can rewrite our pruned minimax algorithm so that it is very similar to listing 1:
Again, the winner function is the interesting part. If one has a winner function, it is easy to fill in the details required to derive a completely generic strategy.

In table 6, we can recognize the game tree example from table 5, with some parts taken out; the gray squares represent positions that do not need to be considered by minimax, thanks to alpha-beta pruning. As in table 5, the white squares represent positions which are not part of the game tree, but are included in the picture for visual balance. Also as in table 5, the moves are chosen by considering the first free position on the board when going trough the board in left to right and bottom to top order.
4 MCTS

4.1 Introduction

Monte-Carlo tree search (MCTS) has become an umbrella term for a class of related algorithms for searching trees probabilistically. This applies directly to games if we decide to search the game tree.

In this section, we will introduce an MCTS algorithm known as UCT. We will mostly follow the exposition in [MCTS Survey, 2012], chapter 3.

4.2 MCTS in general

MCTS studies the game tree as follows. It keeps a record of a subtree of the game-tree containing the nodes that the algorithm has visited so far. It also keeps some extra information about each node, which is supposed to represent an approximation of the “value” of that node. The idea is to somehow find a good expandable node (meaning that it has unvisited children) in the visited part of the game tree, and then to make an excursion from that node, which means doing a quicker kind of search from the node, in order to estimate the value of the node. The information gleaned from this excursion will then contribute to the algorithms knowledge of the game tree.

It is assumed that we have a (reasonably efficient) function that lets us determine the value of a leaf node. Here is a sketch of the steps that will make up our algorithm:

- **Selection**: find a suitable expandable explored node by repeatedly applying a selection function, and select one of its child nodes.
- **Exploration**: run a simulation from the newly found child node and return a score.
- **Back propagation**: use the score found in the previous step to update the visited tree in an appropriate way.

These steps are iterated a number of times in order to make a single move. Each iteration yields a more complete and refined knowledge of the game tree, thanks to the back propagation step. In order to subsequently make a move, a single application of the selection function is made. Note that there are variants of this algorithm which expand and explore multiple nodes instead of just one, but the principle is the same otherwise. Note also that this algorithm is far from complete. There are various appropriate ways of performing each of these steps, depending on the situation. The
next section describes one of the possibilities: the UCT (Upper Confidence bounds for Trees) algorithm.

4.3 The UCT algorithm

In this section, we fill in each of the steps outlined in the previous section, for the special case of the UCT algorithm.

Each node $v$ in the explored part of the game tree has an attached score, which is just a real-valued number, say $S(v)$.

4.3.1 The selection step

Selection takes place in the explored part of the game tree, and can therefore use the score, $S$. We repeatedly pick the “best child” of the current node, in the following sense. If $v$ has children which have not been explored, then pick any of them as the best child. If all children of $v$ have already been explored, then we pick a child, $v'$, which maximizes

$$S(v') = \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

where $v'$ is a child of $v$, $N$ is the visit count and $Q$ is the accumulated score for a node (we will see later how to keep track of $Q$ and $N$, for a given node). The parameter $c$ determines the amount of exploration. We will choose $c = 1/\sqrt{2}$ as per the comments in [MCTS Survey, 2012, p. 9].

This selection process continues until we find either an unexplored node or run into a node without children (i.e. a leaf node), in which case we return that leaf node.

4.3.2 The exploration step

When we have found a node using the selection step, we will explore that node, which will yield a score. If we are “exploring” a leaf node, the outcome is just First, Second or Neither win, according to what outcome the leaf node represents. In case we are not exploring a leaf node, then we are exploring an unexplored non-leaf node, and then simply search randomly from that node until we run into a leaf node, which we know how to evaluate an outcome for. If the node has not previously been explored, it will finally be marked as explored, and its visit count and score will be initialized as appropriate.
The result of the exploration step is a real-valued number – a score. The score depends on who moves in order to arrive at the explored node, as well as what the outcome of the random search was. If First moves in order to arrive at the explored node, then a First, Second or Neither win outcome has a value of 1, −1 or 0, respectively. If Second moves to the explored node, these scores are negated.

4.3.3 The backup step
As explained, the exploration step yields a score which is either 1, −1 or 0. In the backup step we go back up the way we came, all the way to the root node. As we go, we increment the visit count, \( N \). We also update the cumulative score, \( Q \), in the following way. Going up the tree, we bring with us the score corresponding to the outcome of the exploration step. However, in order to reflect the fact that the two players have opposite opinions of the outcome, we make sure to alternate its sign on each step. This alternated score is added to the cumulative score, \( Q \).

4.4 An example
In this section, we give an example of the UCT algorithm described above. To keep things simple, we do only a single iteration. However, in order to get a non-trivial iteration, we assume that five iterations have already been done, and do iteration number six.

In table 7, we have our starting point. We must first carry out the selection step. As can be seen from the figure, there are two cases: one with \( N = 1 \) and \( Q = −1 \), and one with \( N = 1 \) and \( Q = 1 \). Let the choices on the second row of table 7 be \( v_1, v_2, v_3 \) and \( v_4 \), from left to right. Using expression (2) with \( c = 1/\sqrt{2} \), we get

\[
S(v_1) = \sqrt{\ln 5} − 1 \quad \text{for the first choice, and}
S(v_2) = S(v_3) = S(v_4) = \sqrt{\ln 5} + 1 \quad \text{for the other three choices.}
\]

So \( v_2, v_3 \) and \( v_4 \) all maximize \( S \). Suppose we choose to explore \( v_2 \). Since this node has no explored children, we do not need to calculate (2) – we simply select the first child, i.e. the fourth node from the left, on the third row. This node is the result of the selection step.

Now it is time to perform the exploration step. This is just a random search from the selected node, until we hit a leaf node. As can be seen from table 7, there are two possible outcomes: either the seventh node on the fourth row, or the eight node on the fifth row. Suppose that the outcome is the latter. This node is a win for First. Since we are exploring a node which First moves to, the numerical score
of this outcome is 1, as per the explanation in 4.3.2. Thus, the backup step starts with the score 1, and so the explored node gets initialized with $N = 1$ and $Q = 1$. The backup step is not done yet: we should continue all the way up to the root node, remembering to alternate the sign and adjust the accumulated scores and visit counts as we go, as described in section 4.3.3. The complete result after the backup step is shown in table 8.

### 4.5 UCT implementation

It is assumed that we have a function, `value`, which can only be applied to terminal values and which gives a real number representing the value of a given node from the point of view of the player in turn (i.e. the opponent of the player who made the previous move). The value is 1 if the position is a winning position for the player in turn, $-1$ if it is a losing position, and 0 otherwise. We also assume that we have a function named `choices`, which we can apply to a node in order to find the set of possible choices of nodes that the player in turn could move to. Finally, we assume that our nodes may carry “MCTS data” of the following format.

```haskell
data MCTSNodeData = MCTSNodeData { visitCount :: Int, score :: Score }
```

![Diagram showing the tree structure with N=5, Q=-1 at the root, and N=1, Q=-1 at the explored node.](image)

Table 7: Before the iteration
Table 8: After the iteration

A node only carries MCTSNodeData if it has been explored by the explore function, below. The Score type is just a synonym for a real number type, like Float. The function getMCTSData will return the MCTS data for a given node if it has any, and we can set it by means of setMCTSData.

We begin by looking at the recon function, which denotes the score from the point of view of the player in turn, of a random search from the given position.

```haskell
recon position |
| terminal position = return $ value pos |
| otherwise = do |
| c <- Random.fromList [(c,1) |
| | c <- choices position] |
| s <- recon c |
| return $ -s
```

It reads as follows: in case the given position is terminal, then we can return the value of that position. Otherwise we select a random child, apply recon to it, and get a score, s, back. Note that s represents the score from the point of view of the opponent of the player in turn, since recon was applied to a child of position. Therefore we must negate the score before we return it.

Next, we’ll look at the explore function, which is the core of the algorithm. The return value of the function is a tuple of a score together with the explored node. Just like recon, the score it returns is relative to the player who’s turn it is in the
given node. Apart from the node to explore, it also takes \( c_{\text{Exp}} \), which is just the \( c \) parameter from expression (2).

```haskell
explore cExp node =
  case getMCTSData node of
  Nothing -> do
    s <- if terminal node
    then return $ value node
    else recon node
    return ( s,
            setMCTSData node $ MCTSNodeData {visitCount = 1,
                                            score = s} )
  Just (MCTSNodeData {visitCount = vc,
                      score = sc}) -> do
    case terminal node of
    True -> do
      let s = value node
      return ( s,
               setMCTSData node $ MCTSNodeData {visitCount = vc + 1,
                                                 score = sc + s} )
    False -> do
      let (c, cs) = popBestChild cExp node
      (s, c') <- explore cExp c
      let s' = negate s
      node' = setChoices node (c':cs) in
      return ( s',
               setMCTSData node' $ MCTSNodeData {visitCount = vc + 1,
                                                  score = sc + s'} )
```

At the top level, the function is split up into two cases – either our node has no MCTS data, or it does. If the node does not have MCTS data, we obtain a score, \( s \), in either of two ways: using the \texttt{value} function if the node is terminal, or else by applying the \texttt{recon} to the node, i.e. doing a random search. In either case we get a score, and so we can return the explored version of our node.

In case the node has MCTS data, i.e. is explored, we again have two sub-cases: terminal or not terminal. If the explored node is terminal, we again use the \texttt{value} function to obtain a score, and use that to update the MCTS data for the node. If the explored node is not terminal, then we select the best child as described in
section 4.3.1, explore that child recursively with another call to `explore` and use the result to return an updated node.

One important ingredient is the `compareChildren` function. It takes a node and two children, \( a \) and \( b \), of the node, and returns an ordering, which is just a type for representing “less than”, “equal to” or “greater than”. It allows us to sort a set of children, and therefore to write `popBestChild` and `findBestMove` (below), with relative ease. The function is just a straight encoding of the rules mentioned in 4.3.1.

```haskell
compareChildren cExp node a b =
  case (getMCTSData a, getMCTSData b) of
    (Just aData, Just bData) ->
      let Just parentData = getMCTSData node in
        compare (reconScore parentData aData) (reconScore parentData bData)
    (Nothing, Nothing) -> EQ
    (Just _, Nothing) -> LT
    (Nothing, Just _) -> GT

where
  reconScore :: MCTSNodeData -> MCTSNodeData -> Score
  reconScore parentData childData =
    let (vcp, sp) = (visitCount $ parentData, score $ parentData)
        (vcc, sc) = (visitCount $ childData, score $ childData) in
        ( sc / (fromIntegral vcc) ) +
        cExp * sqrt ( 2.0*(log $ fromIntegral vcp) / (fromIntegral vcc) )
```

The function splits into two cases. In the case where both children have been explored (i.e. have got MCTS data), we use expression (2) to produce two numbers which us an ordering in the usual way. In case neither child have been explored, they are equal. In case the first has been explored but not the second, then the first is less than the second, and the final case is just the reverse of this.

With the `explore` function and `compareChildren` in hand, we can easily write the final MCTS strategy. It needs two parameters: a number of iterations and a node to work on.
The base case of zero iterations is listed first. It uses the function `findBestMove` which determines the best move for the player in turn, according to the ordering implied by `compareChildren`. In the other case, one round of `explore` is executed, and the strategy is applied recursively to the resulting node.

There are still a few minor blanks left to fill in. For the complete code, see section 9.

5 Generating games

5.1 Introduction and basic definitions

In the previous section, we showed how to implement MCTS (or more specifically: UCT), which can be used directly to play games in a probabilistic manner. The plan is to try it out on many different games. In order to do that, we have to generate the games themselves. The goal of this chapter is to show how to do just that, given some constraints which we will cover later.

Since we will use a tool called Nauty to generate games, and since Nauty talks about graphs, it is appropriate to talk about hypergraphs in place of positional games.

Definition 5.1 *A hypergraph is a set of vertices, V, together with a set of hyperedges E, which are non-empty subsets of V.*

A positional game is equivalent to a hypergraph, if we take the so-called winning sets to be the hyperedges.

5.2 Nauty and hypergraphs

Nauty is a tool to generate and work with graphs, and can be downloaded at [http://cs.anu.edu.au/people/bdm/nauty/](http://cs.anu.edu.au/people/bdm/nauty/). We are particularly interested in the tools `genbg` and `showg`.
genbg is used to generate (non-isomorphic) bipartite graphs. By default it will output the graphs in the very compact g6 format. showg is used to turn these g6-formatted graphs into more human-readable form.

A bipartite graph corresponds to a hypergraph in the following manner. Let us say that the two colors of the bipartite graph are red and blue. Then we can decide that the red vertices correspond exactly to the set of vertices in our hypergraph. We can make the blue vertices correspond to hyperedges by defining the hyperedge for a given blue vertex as those red vertices which are connected to it. This correspondence is indicated in table 9, where a bipartite incidence graph (aka Levi graph) is displayed on the left hand side, and the corresponding hypergraph is displayed on the right hand side.

5.3 Choosing the command line switches

In this section, we work out as an example all of the hypergraphs with 2 vertices and 3 hyperedges, using Nauty. We will assume that the reader has obtained Nauty, and that the current working directory contains the executables genbg and showg.

Here is the command to generate and display all non-isomorphic bipartite graphs with 2 vertices in the first class and 3 vertices in the second class.

```
$ ./genbg 2 3 | ./showg
```

The output (in tabular form here, for compactness) is as follows:
Each graph is displayed as five rows – one row for each vertex. The rows contain the index of the vertex, followed by a list of it’s neighbours. The first two rows correspond to the two vertices in the first class and thus correspond to the vertices of our would-be hypergraphs. The remaining three rows correspond to the vertices of the second color, and thus correspond to the would-be hyperedges. Clearly, there are some issues to work out. Firstly, note that Graph 1 does not give us an actual hypergraph; both hyperedges would be empty, which we do not allow. The same criticism holds for Graph 2 and Graph 3.

To get around this, we use the command line switch `-dm:n` where the `m` and `n` are the minimum degree of the first and second class of vertices, respectively.

If we pick `-d1:1` we are saying that each vertex in our hypergraph must be in some hyperedge, and that each hyperedge must contain at least one vertex, i.e. must not be empty.

```
$ ./genbg 2 3 -d1:1 | ./showg
```

The output is:
There are still some issues. If we were to try to translate Graph 1 to a hypergraph, we would get the same hyperedge twice. This issue is resolved with the command line switch -z, which ensures that no two vertices in the second class can have the same neighborhood.

```
$ ./genbg 2 3 -z -d1:1 | ./showg
```

This yields the output:

Graph 1, order 5.
0 : 2 4;
1 : 3 4;
2 : 0;
3 : 0;
4 : 0 1;

So in the end we have only a single hypergraph in this class. It has two vertices, each with its own singleton hyperedge plus a hyperedge that contains both of the vertices. Table 9 shows how this bipartite graph and its corresponding hypergraph are related.

Our hypothesis at this point is that a command such as ./genbg m n -z -d0:1 will generate all non-isomorphic hypergraphs of m vertices and n edges.

More precisely, we need to prove the following results.

**Definition 5.2** The neighbourhood $\mathcal{N}(v) \subset V(G)$ of a vertex $v \in V(G)$ is defined as the set of vertices which have edges connected directly to $v$.

**Definition 5.3** A graph $G$ is said to be bipartite, if there are $V_1, V_2 \subset V(G)$ such that $V(G) = V_1 \cup V_2$, such that $V_1 \cap V_2 = \emptyset$, and such that $\mathcal{N}(V_1) \subset V_2$ and $\mathcal{N}(V_2) \subset V_1$. We denote these subsets by $V_1(G)$ and $V_2(G)$, and call them the first and second classes of $G$, respectively.
Lemma 5.4 Let $\mathcal{B}$ be the set of all bipartite graphs where all vertices in the second class have distinct, non-empty neighbourhoods. Let $\mathcal{H}$ be the set of all hypergraphs. Then there is a bijection, up to isomorphism, between $\mathcal{B}$ and $\mathcal{H}$.

Proof Let $G \in \mathcal{B}$. Define $\varphi(G) = (V_1(G), \mathcal{N}(V_2(G))) \in \mathcal{H}$. (This is a member of $\mathcal{H}$ by hypothesis – each element of $\mathcal{N}(V_2(G))$ is non-empty.) In this way we have constructed a mapping from $\mathcal{B}$ to $\mathcal{H}$, which we claim has the desired properties.

Injectivity: To show injectivity up to isomorphism, we should show $G \cong G' \Rightarrow \varphi(G) \cong \varphi(G')$, for any two $G, G' \in \mathcal{B}$. Suppose that there is a graph isomorphism $f : G \to G'$. Define $h : V_1(G) \to V_1(G')$ by $h(v) = f(v)$. In other words, $h$ is just $f$ restricted to $V_1(G)$. We must now show that $h$ yields a bijection between $\mathcal{N}(V_2(G))$ and $\mathcal{N}(V_2(G'))$. Since $\mathcal{N}(V_2(G)) \subset V_1(G)$ we get $h = f$ on $\mathcal{N}(V_2(G))$. Now, since $f$ is an isomorphism, $f$ indeed yields a bijection between $\mathcal{N}(V_2(G))$ and $\mathcal{N}(V_2(G'))$. Therefore, so does $h$. We have shown $G \cong G' \Rightarrow \varphi(G) \cong \varphi(G')$. To show the converse, suppose that we have a hypergraph isomorphism between $h : \varphi(G) \to \varphi(G')$. We define $f : G \to G'$ in two cases: $f|_{V_1(G)} = h$ and $f|_{V_2(G)} = \mathcal{N}^{-1}h\mathcal{N}$. We should check that the second expression is well defined. All vertices in $V_2(G)$ have distinct neighbourhoods, which is just another way of saying that $\mathcal{N}$ yields a bijection between vertices in $V_2(G)$ and their neighbourhoods, for all $G$. We also have that $h$ forms a bijection between the neighbourhoods of $V_2(G)$ and $V_2(G')$, by its definition. These two facts are sufficient to show that $f|_{V_2(G)}$ is well defined. To show that $f$ is an isomorphism, we should also check that whenever $v_1 \sim v_2$ in $G$, we get $f(v_1) \sim f(v_2)$ in $G'$, and vice-versa. Suppose that $v_1 \sim v_2$ in $G$, where $v_1 \in V_1(G)$ and $v_2 \in V_2(G)$. It follows that $v_1 \in \mathcal{N}(v_2)$. Therefore $h(v_1) \in h\mathcal{N}(v_2)$.
Now, \( h(v_1) = f(v_1) \) and \( h\mathcal{N}(v_2) = \mathcal{N}f(v_2) \), so we have \( f(v_1) \in \mathcal{N}f(v_2) \), which implies \( f(v_1) \sim f(v_2) \). This sequence of implications also holds in reverse, and so we get \( f(v_1) \sim f(v_2) \Rightarrow v_1 \sim v_2 \). (Note that \( h(v_1) \in h\mathcal{N}(v_2) \Rightarrow v_1 \in \mathcal{N}(v_2) \) holds because \( h \) is a bijection.)

**Surjectivity:** To show *surjectivity up to isomorphism*, we should show that, given any \( H \in \mathcal{H} \), there is a \( G \in \mathcal{B} \) such that \( \varphi(G) \cong H \). So let \( H = (V, \mathcal{F}) \in \mathcal{H} \) be any hypergraph. Clearly, we can define a set \( W \) such that \( |W| = |\mathcal{F}| \) and \( V \cap W = \emptyset \). Since \( |W| = |\mathcal{F}| \) is finite, there exists a bijection \( \phi : W \to \mathcal{F} \). We define \( G \in \mathcal{B} \) by \( V_1(G) = V, V_2(G) = W, \) and \( v \sim w \iff v \in \phi(w) \), where \( v \in V \) and \( w \in W \). By definition, we then have \( \varphi(G) = (V, \mathcal{N}(W)) \). We can also see that \( \mathcal{N}(W) = \mathcal{F} \). Thus, we get \( \varphi(G) = (V, \mathcal{F}) = H \). 

We need a bit more precision, as provided by the following result, which follows immediately from the definition of \( \varphi \).

We might want to restrict the bijection on the number of vertices and hyperedges.

**Corollary 5.5** Let \( \mathcal{B}_{n,m} \subset \mathcal{B} \) be those bipartite graphs in \( \mathcal{B} \) with \( n \) vertices in the first class and \( m \) vertices in the second class. Let \( \mathcal{H}_{n,m} \) and the set of all non-isomorphic hypergraphs with \( n \) vertices and \( m \) hyperedges. Then there is a bijection between \( \mathcal{B}_{n,m} \) and \( \mathcal{H}_{n,m} \).

We can also restrict on the size of the hyperedges and the number of hyperedges a given vertex can occur in.

**Corollary 5.6** Let \( d, d', D, D' \) be integers. Let \( \mathcal{B}_{d,D,d',D} \subset \mathcal{B} \) be those bipartite graphs \( G \in \mathcal{B} \) which satisfy \( d \leq |\mathcal{N}(V_1(G))| \leq D \) and \( d' \leq |\mathcal{N}(V_2(G))| \leq D' \). Let \( \mathcal{H}_{d,D,d',D} \subset \mathcal{H} \) be those hypergraphs \( (V, \mathcal{F}) \in \mathcal{B} \) with vertices that are in at least \( d \) and at most \( D \) hyperedges (inclusive), and \( d' \leq |\mathcal{F}| \leq D' \). Then there is a bijection between \( \mathcal{B}_{d,D,d',D} \) and \( \mathcal{H}_{d,D,d',D} \).

Or, we can restrict it in both ways.

**Corollary 5.7** There is a bijection between \( \mathcal{B}_{n,m} \cap \mathcal{B}_{d,D,d',D} \) and \( \mathcal{H}_{n,m} \cap \mathcal{H}_{d,D,d',D} \).

### 6 Experiments

#### 6.1 Introduction

In this section we give an overview of the experiments. We want to find out how “good” the UCT strategy is for various numbers of iterations. Of course we expect
that it becomes better at higher numbers of iterations, but how many are necessary? We would also want to know at what number, or even if, it becomes virtually perfect.

We can measure the performance of UCT, for a given game, by playing with UCT at \( n \) iterations as First versus perfect as Second. Since the outcome is random, we’ll want to repeat this many times, and record how many times UCT wins, losses and ties.

We also want to classify the game being played as first player win, second player win or neither player win. As discussed in section 3.3, this means running perfect vs perfect for the game and recording the outcome.

6.2 Classes of games to study

The number of positional games grows rapidly with the number of vertices (or positions). If we want to study interesting games of higher number of vertices, we are going to need to focus on particular classes.

In the experiment outlined below, we will focus on games which have hyperedges (winning sets) containing either two or three vertices. In order to limit the number of vertices in a hyperedge, we can use the arguments \(-\text{dm:n}\) and \(-\text{Dm:n}\) with \texttt{genbg}. The argument \(-\text{dm:n}\) gives lower bounds, \( m \) and \( n \), for the minimum degrees of the first and second classes of vertices, respectively. Recall that the first class of vertices corresponds to the vertices of the hyper-graph, and the second class corresponds to the hyperedges, as described in section 5.3. Similarly, the argument \(-\text{Dm:n}\) specifies upper bounds on the maximum degrees for the two classes. By corollary 5.7, we can list the class of hypergraphs with 4 vertices and 2 hyperedges containing either 2 or 3 vertices, and where each vertex is in at least one hyperedge, by running:

\[
\texttt{$ ./genbg -z 4 2 -d1:2 -D2:3 | ./showg$
}
\]

The output is:

<table>
<thead>
<tr>
<th>Graph 1, order 6.</th>
<th>Graph 2, order 6.</th>
<th>Graph 3, order 6.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 : 4 );</td>
<td>( 0 : 4 5 );</td>
<td>( 0 : 4 5 );</td>
</tr>
<tr>
<td>( 1 : 4 );</td>
<td>( 1 : 4 );</td>
<td>( 1 : 4 5 );</td>
</tr>
<tr>
<td>( 2 : 5 );</td>
<td>( 2 : 5 );</td>
<td>( 2 : 4 );</td>
</tr>
<tr>
<td>( 3 : 5 );</td>
<td>( 3 : 5 );</td>
<td>( 3 : 5 );</td>
</tr>
<tr>
<td>( 4 : 0 1 );</td>
<td>( 4 : 0 1 );</td>
<td>( 4 : 0 1 );</td>
</tr>
<tr>
<td>( 5 : 2 3 );</td>
<td>( 5 : 0 2 3 );</td>
<td>( 5 : 0 1 3 );</td>
</tr>
</tbody>
</table>

6.3 The experiment

This chapter describes our experiment. Mainly the setup and contents – results are dealt with more thoroughly in the next chapter. We are dealing with the set of all
hypergraphs with hyperedges of size either two or three, and where each vertex is in at least one hyperedge, from the following classes:

<table>
<thead>
<tr>
<th>#vertices</th>
<th>#hyperedges</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1...4</td>
</tr>
<tr>
<td>4</td>
<td>2...10</td>
</tr>
<tr>
<td>5</td>
<td>2...20</td>
</tr>
<tr>
<td>6</td>
<td>2...12</td>
</tr>
</tbody>
</table>

That ends up being about 2.2 million hypergraphs, 2,215,838, to be exact.

For each of those hypergraphs, we run 6 tournaments of UCT versus perfect (minimax): UCT with 2,7,12,17,22 and 27 iterations, as First, versus perfect as Second. Each of these tournaments consist of 100 games, since the outcome of a game is random. Within a given tournament, we record the number of First wins, Second wins and the number of ties.

We also classify the games as First, Second or Neither player win, by playing two optimal players against each other and recording the result.

This all ends up in a database, which is discussed in more detail in chapter 8.

6.3.1 A preview of the results

The various data gathered are viewed in several different ways in the next section, in order to get some idea about how well MCTS performs against a perfect opponent.

For instance, in section 8.3 we look at the number of wins, losses and ties at a given number of iterations of for MCTS as First versus a perfect opponent as Second. Since there are many more First-win games than Neither-win games, we look at this table separately for those two classes of games. Note that in First-win games, MCTS wins against a perfect opponent only if it manages to reproduce perfect play, and in Neither-win games MCTS will tie only if it manages to reproduce perfect play. The trend is that in First-win games, MCTS is quite weak at very low numbers of iterations, but then climbs quite rapidly toward approximately 0.96 win rate at 27 iterations. The trend in Neither-win games is different in the way that, even at extremely low numbers of iterations, we get 0.87 tie rate. The rate then proceeds to climb steadily to 0.99 at 27 iterations.

Examples of a different way to split the results up is given in section 8.5. There, the games are classified based on their number of 2-edges and 3-edges. For each such category, we study the win-rates across the number of iterations of MCTS as First versus perfect as Second. In the first table, we quickly notice that in almost all of the cases, increasing numbers of iterations means an increasing win-rate for MCTS. There are exceptions to this in some categories (such as for twelve 2-edges and zero
3-edges), but even in those cases the performance of MCTS is not bad. This section also shows that there are different aspects to MCTS’s performance. For instance, in the case of 0 2-edges and 7 3-edges, MCTS only managed to get up to a win-rate of 0.6 at most. But on the other hand, in the same category and same number of iterations, the win-rate of the perfect opponent is only 0.1, meaning that the games ended up with no winner at a rate of 0.4. This can be contrasted with, for example, the category of 2 2-edges and 10 3-edges, where on the one hand the win-rate of MCTS goes up to 0.89 at most, but at the same time, the win-rate of Second is quite high at 0.7.
7 Bibliography


8 Appendix A: Database queries

This section covers a number of interesting queries that one could make into the database resulting from the experiment described in section 6.3.

The database is about 1.6 gigabytes, and is available to download at:
http://abel.math.umu.se/~klasm/Data/Games/database.sqlite3

It is assumed that you have SQLite 3 installed. To begin making queries into the database database.sqlite3 (in the current working directory), you would issue the shell command:

$ sqlite3 database.sqlite3

You will then be greeted with a prompt like

sqlite>

where you can begin typing the queries and commands covered below.

For the sake of completeness, we will run all the subsequent queries against the database database.sqlite3, which corresponds to our experiment, described in section 6.3.

8.1 Database overview

The following commands are not queries, but they are very important to know.

sqlite> .tables
experiments                    results_Perfect_vs_Perfect
hypergraphs                    results_UCT_vs_Perfect

These are the tables in database.sqlite3. The hypergraphs table contains the games we want to play, along with some meta-information about the games. (See next command for details.) The table results_Perfect_vs_Perfect contains the outcome for each game when perfect First plays against perfect Second. (The perfect strategy is implemented using minimax, as covered in section 3.3.) The table results_UCT_vs_Perfect contains a number of sample outcomes when UCT (section 4.3) plays as First against a perfect opponent as Second.

To get more precise information about what’s contained in the above tables, we issue the following command:
This command not only tells you the names, types and constraints of the columns making up the table, but it does so by telling you the exact command that was issued to create the table. The important information here is the names and types. We can see that `hypergraphs` has a column named `hypergraph`, which stores the hypergraph as a non-null string (in the graph6 format). The `hypergraphs` table also contains the number of vertices and edges as well as a more human-readable representation. Even though the last three columns of `hypergraphs` can easily be derived from the first column, they are nice to have there for convenience when making queries, as will be seen below.

2 NULL is used to denote ‘nothing’, and is not appropriate here, which is why it is explicitly disallowed.
8.2 Overall database structure

This section presents various ways of querying the overall structure of the database.

8.2.1 Query: Vertices and edges

Suppose that we are interested in finding out, roughly, the structure of the table results_UCT_vs_Perfect in our database. We might first be interested in knowing which “classes” of hypergraphs are in the table, in the sense that two hypergraphs are in the same class iff they have the same number of vertices and hyperedges.

The following command will print out all such classes in the format #vertices | #edges.

\[
\text{sqlite} > \text{SELECT DISTINCT numvertices, numedges} \\
\quad \text{FROM results_UCT_vs_Perfect NATURAL JOIN hypergraphs;}
\]

The output is suppressed here, since it is not ordered. If we want ordered results, we extend the previous query a little bit:

\[
\text{sqlite} > \text{SELECT DISTINCT numvertices, numedges} \\
\quad \text{FROM results_UCT_vs_Perfect NATURAL JOIN hypergraphs} \\
\quad \text{ORDER BY numvertices, numedges;}
\]

| 3|1   | 4|7   | 5|7   | 5|16  |
|---|-----|-----|-----|-----|-----|-----|
| 3|2   | 4|8   | 5|8   | 5|17  | 6|7   |
| 3|3   | 4|9   | 5|9   | 5|18  | 6|7   |
| 3|4   | 4|10  | 5|10  | 5|19  | 6|8   |
| 4|2   | 5|2   | 5|11  | 5|20  | 6|9   |
| 4|3   | 5|3   | 5|12  | 6|2   | 6|10  |
| 4|4   | 5|4   | 5|13  | 6|3   | 6|11  |
| 4|5   | 5|5   | 5|14  | 6|4   | 6|12  |
| 4|6   | 5|6   | 5|15  | 6|5   |       |

Now we can see why the hypergraphs table exists, and contains redundant information: we simply need to do a \text{NATURAL JOIN} with it in order to get the number of vertices and edges for our hypergraphs in results_UCT_vs_Perfect.

As can be seen above, some hypergraphs from the class 4 | 7 are present in results_UCT_vs_Perfect. How many?

\[
\text{sqlite} > \text{SELECT DISTINCT COUNT (*)} \\
\quad \text{FROM ((SELECT DISTINCT hypergraph FROM results_UCT_vs_Perfect) NATURAL JOIN hypergraphs)} \\
\quad \text{WHERE numvertices = 4 AND numedges = 7;}
\]

11
Note that we add the qualifier **DISTINCT** when pulling hypergraphs from \texttt{results\_UCT\_vs\_Perfect}, since each hypergraph in this table occurs three times. (See the experiment structure in 6.3.)

If we want to know this number for all combinations of \texttt{numvertices} and \texttt{numedges} stored in \texttt{results\_UCT\_vs\_Perfect}, we can issue the following query:

```sql
sqlite> SELECT DISTINCT numvertices, numedges, COUNT (hypergraph)
...> FROM ((SELECT DISTINCT hypergraph FROM results\_UCT\_vs\_Perfect)
...> NATURAL JOIN hypergraphs)
...> GROUP BY numvertices, numedges;
```

<table>
<thead>
<tr>
<th>numvertices</th>
<th>numedges</th>
<th>COUNT (hypergraph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>174</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>414</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>795</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1254</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1642</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1805</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>1644</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1259</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>806</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>431</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>192</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>75</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>88</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>523</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>2527</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>10205</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>35018</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>103426</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>265716</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>598486</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>1189151</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>806</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>431</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>192</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>

The attentive reader will note that the number of hypergraphs listed in the above table adds up to 22,158,38 – the number of hypergraphs in the database:

```sql
sqlite> SELECT COUNT (*) FROM hypergraphs;
2215838
```

### 8.3 Results at large

The following query shows the percentage of First wins, Second wins and Neither wins, respectively, for the games classified as a first player win.

```sql
sqlite> SELECT SUM(num\_first\_wins), SUM(num\_second\_wins),
...> SUM(num\_neither\_wins), num\_iterations\_first
...> FROM results\_Perfect\_vs\_Perfect NATURAL JOIN results\_UCT\_vs\_Perfect
...> WHERE winner = "First"
...> GROUP BY num\_iterations\_first;
```

<table>
<thead>
<tr>
<th>num_iterations_first</th>
<th>SUM(num_first_wins)</th>
<th>SUM(num_second_wins)</th>
<th>SUM(num_neither_wins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15900100</td>
<td>15900100</td>
<td>15900100</td>
<td>15900100</td>
</tr>
<tr>
<td>1205213500</td>
<td>1205213500</td>
<td>1205213500</td>
<td>1205213500</td>
</tr>
<tr>
<td>95947824</td>
<td>95947824</td>
<td>95947824</td>
<td>95947824</td>
</tr>
<tr>
<td>122110522</td>
<td>122110522</td>
<td>122110522</td>
<td>122110522</td>
</tr>
<tr>
<td>13347654</td>
<td>13347654</td>
<td>13347654</td>
<td>13347654</td>
</tr>
<tr>
<td>170292845</td>
<td>170292845</td>
<td>170292845</td>
<td>170292845</td>
</tr>
<tr>
<td>197396923</td>
<td>197396923</td>
<td>197396923</td>
<td>197396923</td>
</tr>
<tr>
<td>21355856</td>
<td>21355856</td>
<td>21355856</td>
<td>21355856</td>
</tr>
<tr>
<td>2653221</td>
<td>2653221</td>
<td>2653221</td>
<td>2653221</td>
</tr>
<tr>
<td>10629080</td>
<td>10629080</td>
<td>10629080</td>
<td>10629080</td>
</tr>
<tr>
<td>2472675</td>
<td>2472675</td>
<td>2472675</td>
<td>2472675</td>
</tr>
<tr>
<td>213217957</td>
<td>213217957</td>
<td>213217957</td>
<td>213217957</td>
</tr>
</tbody>
</table>

Or, in normalized form and rounded:
Here is the same query, but investigating games where neither player can win if both play perfectly:

```sql
sqlite> SELECT SUM(num_first_wins), SUM(num_second_wins),
    ...> SUM(num_neither_wins), num_iterations_first
    ...> FROM results_Perfect_vs_Perfect NATURAL JOIN results_UCT_vs_Perfect
    ...> WHERE winner = "Neither"
    ...> GROUP BY num_iterations_first;
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22900</td>
<td>154900</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>9516</td>
<td>168284</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>12010</td>
<td>165790</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>7051</td>
<td>170749</td>
<td>17</td>
</tr>
<tr>
<td>0</td>
<td>3315</td>
<td>174485</td>
<td>22</td>
</tr>
<tr>
<td>0</td>
<td>1107</td>
<td>176693</td>
<td>27</td>
</tr>
</tbody>
</table>

Again, in normalized form:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.13</td>
<td>0.87</td>
<td>2</td>
</tr>
<tr>
<td>0.00</td>
<td>0.05</td>
<td>0.95</td>
<td>7</td>
</tr>
<tr>
<td>0.00</td>
<td>0.07</td>
<td>0.93</td>
<td>12</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.96</td>
<td>17</td>
</tr>
<tr>
<td>0.00</td>
<td>0.02</td>
<td>0.98</td>
<td>22</td>
</tr>
<tr>
<td>0.00</td>
<td>0.01</td>
<td>0.99</td>
<td>27</td>
</tr>
</tbody>
</table>

### 8.4 Results in perfect play

Here is a query which will summarize the results from our games when two perfect players play against each other:
SELECT hc.numvertices, hc.numedges, IFNULL(numfirstwins, 0), IFNULL(numsecondwins, 0), IFNULL(numneitherwins, 0) FROM (SELECT numvertices, numedges FROM hypergraphs GROUP BY numvertices, numedges) hc LEFT OUTER JOIN (SELECT numvertices, numedges, COUNT(*) AS numfirstwins FROM hypergraphs NATURAL JOIN results_Perfect_vs_Perfect WHERE winner = "First" GROUP BY numvertices, numedges) fwc ON (hc.numvertices = fwc.numvertices AND hc.numedges = fwc.numedges) LEFT OUTER JOIN (SELECT numvertices, numedges, COUNT(*) AS numsecondwins FROM hypergraphs NATURAL JOIN results_Perfect_vs_Perfect WHERE winner = "Second" GROUP BY numvertices, numedges) swc ON (hc.numvertices = swc.numvertices AND hc.numedges = swc.numvertices) LEFT OUTER JOIN (SELECT numvertices, numedges, COUNT(*) AS numneitherwins FROM hypergraphs NATURAL JOIN results_Perfect_vs_Perfect WHERE winner = "Neither" GROUP BY numvertices, numedges) nwc ON (hc.numvertices = nwc.numvertices AND hc.numedges = nwc.numedges);

The output is as follows:

<table>
<thead>
<tr>
<th>numvertices</th>
<th>numedges</th>
<th>numfirstwins</th>
<th>numsecondwins</th>
<th>numneitherwins</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>11</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>16</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>41</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>155</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>402</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>791</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1253</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1642</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1805</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>1644</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1259</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>806</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>431</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>41</td>
<td>0</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>388</td>
<td>0</td>
<td>135</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>2266</td>
<td>0</td>
<td>261</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>9858</td>
<td>0</td>
<td>347</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>34673</td>
<td>0</td>
<td>345</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>103155</td>
<td>0</td>
<td>271</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>265551</td>
<td>0</td>
<td>165</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>598403</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>1189119</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

The first two columns contain the number of vertices and edges, respectively. The following three columns contain the number of times First, Second and Neither won in that category.

8.5 More detailed results

The following tables show percentages of First, Second and Neither wins, respectively, for a given number of 2-edges and 3-edges. (Recall that all edges are of size either 2 or 3, as described in section 6.3.) Note that the number of 3-edges increase along rows, and the number of 2-edges increase along columns. The query is split up into First player win games (games where First wins if both players play perfectly), and Neither player win games (games where neither player wins if both play perfectly). Tables 10 and 11 shows the query for First and Neither player win games, respectively.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 10: Results for First player win games. 2, 7, ... 27 iterations.
Table 11: Results for Neither player win games. 2, 7, ... 27 iterations.
9 Appendix B: Code

9.1 Overview

In this section, we give an overview of the overall structure of the code and the PoGa library.

Some parts of the PoGa library, such as the implementations of the games Tic-Tac-Toe and Hex, are not relevant and are thus not brought up here. The same goes for the entire Graphics submodule, which implements support for a human player for the cases of Tic-Tac-Toe and Hex.

The full code is available in a Git repository at:

https://github.com/leino/thesis_msc_engphys.git

Note that the code may have been updated since it was used to generate the results you can read in this thesis. Therefore, the exact version used has been tagged in the repository as final_code.

9.2 Positional game library

In this section, we introduce the PoGa library. PoGa is mainly an interface used for representing and playing arbitrary positional games. However, PoGa also provides the two strategies discussed earlier: minimax with alpha-beta pruning, and the UCT variant of MCTS.

PoGa also contains implementations of arbitrarily sized \((m \times n)\) Tic-Tac-Toe and Hex.

9.2.1 Game.hs

This module defines the Position type-class, which is the central interface in the PoGa module. It also defines the notion of a Strategy and how to play two strategies against each other in playGame.

Note that a strategy is just a move function, but that the result of the move is a monadic position value. If we use the IO monad, we can represent a human player as a strategy. If we use a monad in the MonadRandom type-class, we can represent a random strategy. (Though, typically not completely random, of course.)

module GameTheory.PoGa.Game
  (Player(First, Second),
   Game(..),
   Winner(..),
Position(choices, winner, terminal, turn),
playGame,
playTournament,
opponent)
where

data Player = First | Second
deriving (Eq, Show)

type Strategy m p = p -> m p

data Winner = Neither | Both | Only Player
deriving (Eq, Show)

newtype Game p = Game {position :: p}

-- The typeclass Position provides an abstract
-- view of positional games
class Position p where
  choices :: p -> [p]
  winner :: p -> Winner
  terminal :: p -> Bool
  turn :: p -> Player

opponent :: Player -> Player
opponent First = Second
opponent Second = First

playGame :: (Position p, Monad m) =>
  Game p -> Strategy m p -> Strategy m p -> m Winner
playGame (Game pos) firstStrategy secondStrategy
  | terminal pos = return $ winner pos
  | otherwise = do
    pos’ <- firstStrategy pos
    playGame (Game pos’) secondStrategy firstStrategy

playTournament :: (Monad m, Position p) =>
  Int -> Game p -> Strategy m p -> Strategy m p -> m [Winner]
playTournament n game fststrat sndstrat = do
  let g = playGame game fststrat sndstrat
  ws <- sequence $ replicate n g
9.2.2 SetGame.hs

A typeclass works well as a general interface, but we need to have a concrete data-type to represent our games.

For many purposes, the following one works quite well. When it doesn’t, a user of the library can always define his own data-type and use most of the functionality provided by the PoGa library, as long as the data-type is an instance of the Position typeclass, as described above.

```
module GameTheory.PoGa.SetGame
    (SetGame(..), makeMove, fromWinningSets)
where

import Data.Set as Set
import qualified GameTheory.PoGa.Game as G

data SetGame v = SetGame { 
    board :: Set v, -- all vertices of board
    turn :: G.Player, -- who's turn is it?
    firstChoices :: Set v, -- vertices that First occupy
    secondChoices :: Set v, -- vertices that Second occupy
    firstWin :: Set v -> Bool, -- is First winner?
    secondWin :: Set v -> Bool} -- is Second winner?

instance Show v => Show (SetGame v) where
    show sg = show $ board sg

    -- all unoccupied vertices
    availableVertices :: Ord v => SetGame v -> Set v
    availableVertices sg =
        ((board sg) \ (firstChoices sg)) \ (secondChoices sg)

    -- current player makes move corresponding to the given vertex
    makeMove :: Ord v => SetGame v -> v -> SetGame v
    makeMove sg vtx =
        case turn sg of
            G.First -> sg{firstChoices = Set.insert vtx (firstChoices sg),
                           turn = G.Second}
```

return ws
G.Second -> sg{secondChoices = Set.insert vtx (secondChoices sg),
   turn = G.First}

-- construct a game given a board and winning sets
fromWinningSets :: (Ord v) =>
   Set.Set v -> Set.Set (Set.Set v) -> Set.Set (Set.Set v) -> SetGame v
fromWinningSets board wssFirst wssSecond =
   SetGame {board = board,
             turn = G.First,
             firstChoices = Set.empty,
             secondChoices = Set.empty,
             firstWin = win wssFirst,
             secondWin = win wssSecond}

   where
   -- Make a win function from a set of winning sets
   win :: (Ord v) => Set.Set (Set.Set v) -> Set.Set v -> Bool
   win wss s =
      or [ws 'Set.isSubsetOf' s | ws <- Set.toList wss]

-- All possible moves that can be made for the given game.
instance Ord v => G.Position (SetGame v) where
   choices sg =
      Set.toList $ Set.mapMonotonic (makeMove sg) (availableVertices sg)
   winner sg =
      case (firstWin sg $ firstChoices sg,
            secondWin sg $ secondChoices sg) of
         (True, False) -> G.Only G.First
         (False, True) -> G.Only G.Second
         (False, False) -> G.Neither
         (True, True) -> G.Both
   terminal sg =
      (Set.null $ availableVertices sg) || (G.winner sg /= G.Neither)
   turn sg = turn sg

9.2.3 The Strategy module

In this section, we list the strategy submodule of PoGa. It contains implementations of minimax with alpha-beta pruning as well as UCT. The implementations of these were discussed in less detail in sections 3.4 and 4.3, respectively.

45
As mentioned, strategies take a position to a monadic position, where the monad typically is in `MonadRandom` or is the `IO` monad.

A strategy representing a human player would require some kind of user interface, which is not provided here. It is provided in the module named `GraphicalGame`, and the required user interfaces are specified for the cases of arbitrarily sized Tic-Tac-Toe and Hex games. Since it is not an important for our purposes here, that entire module is left out of this section for space reasons.

Below follows, `Perfect.hs`, the submodule containing the algorithm for perfect play, i.e. the minimax algorithm with alpha-beta pruning.

```hs

import GameTheory.PoGa.Game
import Data.List (find, sortBy, maximumBy, minimumBy)
import Data.Function (on)

perfectStrategyFirst :: (Monad m, Position p) => p -> m p
perfectStrategyFirst pos = return $ perfectStrategy First pos
perfectStrategySecond :: (Monad m, Position p) => p -> m p
perfectStrategySecond pos = return $ perfectStrategy Second pos

prunedMaximumBy :: (a -> a -> Ordering) -> (a -> Bool) -> [a] -> a
prunedMaximumBy compare isMax xs =
  case find isMax xs of
    Just x -> x
    Nothing -> maximumBy compare xs

alternatingStrategy :: Position p =>
  (Winner -> Winner -> Ordering) ->
  Player -> p -> Winner
alternatingStrategy compareWinners player pos |
  terminal pos = winner pos
  otherwise =
    let ws = map (alternatingStrategy
                  (flip compareWinners)
                  (opponent player)) (choices pos) in
    prunedMaximumBy compareWinners ((==) (Only player)) ws

perfectStrategy :: Position p => Player -> p -> p
perfectStrategy player pos =
  let cs = choices pos
```

46
ws = map (alternatingStrategy
         (flip compareWinners)
         (opponent player)) cs
wcs = zip ws cs in
snd $ prunedMaximumBy (compareWinners 'on' fst) ((==) (Only player) . fst) wcs

where
winnerValue w
    | w == (Only player) = 1
    | w == (Only $ opponent player) = -1
    | otherwise = 0
compareWinners w w' = compare (winnerValue w) (winnerValue w')

Next is MCTS.hs, containing our implementation of the UCT variant of MCTS.

module GameTheory.PoGa.Strategy.MCTS
    (mctsStrategyFirst, mctsStrategySecond, unexploredMetaDataNode, MCTSNodeData(..))

where

import qualified GameTheory.PoGa.Game as Game
import qualified Control.Monad.Random as Random
import Test.QuickCheck
import Data.List (find, sortBy, maximumBy, minimumBy)
import Data.Maybe

data MetaDataNode a b p = MetaDataNode {
    firstData :: Maybe a,
    secondData :: Maybe b,
    choices :: [MetaDataNode a b p],
    terminal :: Bool,
    turn :: Game.Player,
    winner :: Game.Winner
} deriving (Show)

data MCTSNodeData = MCTSNodeData {visitCount :: Int, score :: Score}
newtype MCTSNodeFirst b p = MCTSNodeFirst (MetaDataNode MCTSNodeData b p)
newtype MCTSNodeSecond a p = MCTSNodeSecond (MetaDataNode a MCTSNodeData p)
mctsStrategyFirst :: (Game.Position p, Random.MonadRandom m) =>
   Int -> MetaDataNode MCTSNodeData b p ->
   m (MetaDataNode MCTSNodeData b p)
mctsStrategyFirst numIterations node = do
   MCTSNodeFirst node' <- mctsStrategy numIterations (MCTSNodeFirst node)
   return node'

mctsStrategySecond :: (Game.Position p, Random.MonadRandom m) =>
   Int -> MetaDataNode a MCTSNodeData p ->
   m (MetaDataNode a MCTSNodeData p)
mctsStrategySecond numIterations node = do
   MCTSNodeSecond node' <- mctsStrategy numIterations (MCTSNodeSecond node)
   return node'

-- make an empty node with no metadata, given a node
unexploredMetaDataNode :: Game.Position p => p -> MetaDataNode a b p
unexploredMetaDataNode p =
  MetaDataNode {
    firstData = Nothing,
    secondData = Nothing,
    choices = map unexploredMetaDataNode $ Game.choices p,
    turn = Game.turn p,
    terminal = Game.terminal p,
    winner = Game.winner p
  }

class MCTSNode p where
  getMCTSData :: p -> Maybe MCTSNodeData
  setMCTSData :: p -> MCTSNodeData -> p
  setChoices :: p -> [p] -> p

explored :: MCTSNode p => p -> Bool
explored = isJust . getMCTSData

instance MCTSNode (MCTSNodeFirst b p) where
  getMCTSData (MCTSNodeFirst node) = firstData node
  setMCTSData (MCTSNodeFirst node) md = MCTSNodeFirst $ node {firstData = Just md}
  setChoices (MCTSNodeFirst node) cs =
    let unwrap (MCTSNodeFirst node) = node in
    MCTSNodeFirst $ node {choices = map unwrap cs}
instance MCTSNode (MCTSNodeSecond b p) where
getMCTSData (MCTSNodeSecond node) = secondData node
setMCTSData (MCTSNodeSecond node) md = MCTSNodeSecond $ node {secondData = Just md}
setChoices (MCTSNodeSecond node) cs =
  let unwrap (MCTSNodeSecond node) = node in
  MCTSNodeSecond $ node {choices = map unwrap cs}

instance Game.Position p => Game.Position (MetaDataNode a b p) where
choices mdn = choices mdn
winner mdn = winner mdn
terminal mdn = terminal mdn
turn mdn = turn mdn

instance Game.Position p => Game.Position (MCTSNodeFirst b p) where
choices (MCTSNodeFirst mdn) = map MCTSNodeFirst $ choices mdn
winner (MCTSNodeFirst mdn) = winner mdn
terminal (MCTSNodeFirst mdn) = terminal mdn
turn (MCTSNodeFirst mdn) = turn mdn

instance Game.Position p => Game.Position (MCTSNodeSecond a p) where
choices (MCTSNodeSecond mdn) = map MCTSNodeSecond $ choices mdn
winner (MCTSNodeSecond mdn) = winner mdn
terminal (MCTSNodeSecond mdn) = terminal mdn
turn (MCTSNodeSecond mdn) = turn mdn

type Score = Double

compareChildren :: MCTSNode p => Double -> p -> p -> p -> Ordering
compareChildren cExp node a b =
case (getMCTSData a, getMCTSData b) of
  (Just aData, Just bData) ->
    let Just parentData = getMCTSData node in
    compare (reconScore parentData aData) (reconScore parentData bData)
where
  reconScore :: MCTSNodeData -> MCTSNodeData -> Score
  reconScore parentData childData =
    let (vcp, sp) = (visitCount $ parentData, score $ parentData)
        (vcc, sc) = (visitCount $ childData, score $ childData) in
        ( sc / (fromIntegral vcc) ) +
        cExp * sqrt ( 2.0*(log $ fromIntegral vcp) / (fromIntegral vcc) )

49
popBestChild :: (MCTSNode p, Game.Position p) => Double -> p -> (p, [p])
popBestChild cExp node
    | explored node = popMaximumBy (compareChildren cExp node) (Game.choices node)
    | otherwise = error "popBestChild: un-explored node given"

findBestMove :: (MCTSNode p, Game.Position p) => p -> p
findBestMove node =
    case filter explored (Game.choices node) of
      [] ->
        error "cannot find a best move: no children are explored"
      explored ->
        maximumBy (compareChildren 0.0 node) explored

mctsStrategy :: (MCTSNode p, Game.Position p, Random.MonadRandom m) =>
    Int -> p -> m p
mctsStrategy 0 node = do
    return $ findBestMove node
mctsStrategy numSteps node = do
    (_, node') <- explore cExp node
    mctsStrategy (numSteps-1) node'
    where
      cExp :: Double
      cExp = 1.0 / (sqrt 2.0) -- amount of exploration

explore :: (MCTSNode p, Game.Position p, Random.MonadRandom m) =>
    Score -> p -> m (Score, p)
explore cExp node =
    case getMCTSData node of
      Nothing -> do -- not explored
        s <- if Game.terminal node then return $ value node else recon node
        return (s, setMCTSData node $ MCTSNodeData {visitCount = 1, score = s})
      Just (MCTSNodeData {visitCount = vc, score = sc}) -> do
        case Game.terminal node of
          True -> do
            let s = value node
            return (s, setMCTSData node $ MCTSNodeData {visitCount = vc + 1,
                                                          score = sc + s})
False -> do
    let (c, cs) = popBestChild cExp node
    (s, c') <- explore cExp c
    -- score is negated since it is the score of a
    -- child node relative to this node
    let s' = negate s
    node' = setChoices node (c':cs) in
    return (s', setMCTSData node' $ MCTSNodeData {visitCount = vc + 1,
        score = sc + s'})

recon :: (Random.MonadRandom m, MCTSNode p, Game.Position p) => p -> m Score
recon pos
| Game.terminal pos = return $ value pos
| otherwise = do
    c <- Random.fromList [(c,1) | c <- Game.choices pos]
    s <- recon c
    -- note that score changes sign, since this is the childs score
    return $ -s

-- helper function: takes a non-empty list and
-- extracts its maximum
popMaximumBy :: (a -> a -> Ordering) -> [a] -> (a, [a])
popMaximumBy _ [] = error "popMaximumBy: empty list"
popMaximumBy _ [x] = (x, [])
popMaximumBy cmp (x:xs) =
    let (m, xs') = popMaximumBy cmp xs in
    if cmp m x == LT then (x, m:xs') else (m, x:xs')

value :: (Game.Position p, MCTSNode p) => p -> Score
value pos
| Game.turn pos == Game.First = valueFirst pos
| otherwise = negate $ valueFirst pos

valueFirst :: (Game.Position p, MCTSNode p) => p -> Score
valueFirst node =
case Game.winner node of
    Game.Neither -> 0.0
    Game.Both -> 0.0
    Game.Only Game.First -> -1.0
    Game.Only Game.Second -> 1.0
The **Strategy** submodule also contains **Random.hs**, a strategy for a completely random player.