CARLETON UNIVERSITY

Multiplayer Search Algorithms and Strategies

An evaluation of algorithms

By: Benjamin Franks
Supervisor: B. John Oommen, Ph.D.
COMP 4905
4/12/2011

1 Abstract

Multiplayer Artificial Intelligence algorithms are not a well-studied area. The goal of this project was to establish a framework in which algorithms could be compared against one another. Three competing algorithms were selected: max^n, paranoid and best-reply search (BRS). The framework on which these algorithms are designed to compete is a version of Chinese Checkers. Three different boards have been developed to compare results. The framework is written in the Java programming language. The outcomes of some simulated scenarios show that the BRS tends to excel when facing multiple max^n opponents, while also doing very well against paranoid opponents. The board on which the game takes place also appears to have an effect on the outcomes, although more study is needed.

2 Acknowledgements

I would like to thank Dr. John Oommen for supervising this project.

I would also like to thank Kathryn Burnham for proof-reading this paper and being an invaluable resource in emphasizing clarity.

Finally I would like to thank Steve Hutchinson, for his technical consultation during the review process.

3 Table of Contents

4 Table of Figures

5 Introduction and Background

5.1 Introduction

A perfect-information game is one in which all variables are known to all players. For 2-player games there is a well-known and optimal strategy known as min-max [Knuth and Moore, 1975]. The min-max algorithm functions on the basic assumption that one player will move to maximize their position and that their opponent will be attempting to minimize their score; thus, as the player is expecting their opponent to do the best possible move, if they do anything less the player is at an advantage. This method only works well for two player games as with more players, one needs to worry about interaction between those players.

Multiplayer search algorithms allow for a new set of problems to be studied. By expanding the field, a basis for comparison of any strategies employed is created as long as a heuristic can be produced where there are multiple parties competing for a goal. The search algorithms developed can be applied to incomplete information games, games where data is unknown, with adaptation for probabilistic behaviour. This field could be of interest to those modelling games or real-world situations.

An ideal game to study would be Chinese Checkers. As it is a perfect information game (all pieces are visible on the board), and allows for multiple players. As well, the board can be modified to better suit this project.

5.2 Background

5.2.1 Game Rules

Chinese checkers is a fairly simple game; in this version each player will have 6 pieces, as opposed to the standard 10. The boards designed are a square board of variable size for four players, a triangular board for three players, and a smaller star board that 2, 3, 4, or 6 people can play on.

5.2.1.1 Movement

A player can move their pieces in one of the following two ways. A piece can move to an adjacent tile if it is unoccupied. Or, a piece can hop over an adjacent piece if the next tile in that direction is unoccupied. A player may only move one piece per turn, but may hop multiple game pieces in one turn.

5.2.1.2 Winning

In order to win, a player must move all of their pieces to the goal area (the starting position on the opposite side of the board). It is possible to block an opponent from winning by leaving a piece in the goal area. Nothing is in place to deal with this as none of the algorithms abuse this fact, although a simple solution would just be to declare that as long as half the pieces in the

goal area are the player's they win, as was implemented by Schadd and Winands. The game ends when one player has all their pieces in their goal area.

On the triangle board, a player's goal area is the next corner counter-clockwise.

5.2.2 AI Techniques

5.2.2.1 Heuristics

There are three heuristics that have been implemented which are all based off the distance to the goal positions corner. Additionally, the search algorithms can update the heuristic with a number of moves until the game is won, which increases the value assigned to the heuristic. This greatly improves the endgame viability of these algorithms without having to use a lookup table, as the fewer moves the greater the value returned by the heuristic. Even though updating the heuristic is limited by the depth that the search can provide, that is a sensible restriction. This is also a description of the first heuristic.

The second heuristic takes into account the value calculated using the first heuristic for opponents as well. The return value is limited to a minimum of 0. This could lead to a defeatist attitude where if the player is not close to winning and their opponent is, all their moves will be evaluated as 0 using this heuristic. This should actually have no impact on the game play as their opponent should win well before them for this to be an issue anyways.

The third heuristic is again an expansion on the first heuristic described; however it also factors in the number of hops available for each piece. Only hops that move the piece closer to the goal area contribute towards this heuristic, even though all hops are examined. It was decided to look at all hops and not just one's own pieces as this would be more accurate and opponents would have a direct way of minimizing a player's value, which is required for the paranoid algorithm and the best reply search, as described later.

The heuristic used in the literature [Schadd and Winands, 2011] [Sturtevant, 2002] is the minimum number of moves necessary, using only your own pieces, to reach the goal position; this is retrieved from a lookup table. Aside from the problems in the mid-game where this is no longer accurate due to the ability to hop opponents' pieces [Schadd and Winands, 2011], this heuristic does not naturally lends itself to the paranoid algorithm and the best reply search. Any move an opponent makes will not minimize the player's score. If the search tree does not have a player's move as the terminal node, the added depth is wasted as any move they make is not going to impact the value assigned in the search tree.

5.2.2.2 Search Algorithms

5.2.2.2.1 Max^n

The first algorithm to be discussed is the max^n algorithm. This algorithm makes the assumption that every player will make the move that benefits them the most [Luckhardt and Irani, 1986]. This allows for shallow alpha-beta pruning, but limits deep pruning [Korf, 1991]. This algorithm behaves very well with all of the heuristics described above. This is the only algorithm that will actually make informed decisions using the first heuristic, as it does not take into account the opponents' positions. Thus, any move they make does not minimize the value.

```
profit-vector maxn (position v, lower bound on predecessor's profit
        m = Fi−1(predecessor(v)), remaining depth d, player i)
  compute the feasible successor positions v1, . . . , vb of v.
  // Let H(v) be the evaluation function for leaves.
  if (d == 0 or b == 0) return H(v);

  profit-vector a := (−1, . . . ,−1);

  for j := 1 to b
        F(vj) := maxn (vj, ai, d − 1, (i + 1) mod n);
        if (Fi(vj) > ai) a := F(vj) ;
        if ai > 1 − m return a; // shallow pruning
        if j == b return a;
```

Figure 1: Max^n algorithm pseudocode [Lorenz and Tscheuschner, 2006]

5.2.2.2.2 Paranoid

The second algorithm is called the paranoid algorithm. As the name suggests, it assumes that all opponents are attempting to minimize the player's score. This is a logical extension of the min-max algorithm, since now the player is prepared for the worst move that an opponent can make for them and anything else should be more beneficial [Sturtevant, 2002]. In fact, this can be viewed as a min-max algorithm when there are only two players. This method does not interact with the first heuristic since an opponent's turn will not have an effect on the current player's score. The latter two heuristics will function nicely as an opponent's turn will change the player's score.

This algorithm can be applied to any multiplayer game. It has been studied in games such as Hearts, where it has also been modified to deal with probabilistic outcomes [Sturtevant, 2002].

```
value alphabeta (Position v, value α, value β, remaining depth d,
player i)
// Let player 1 be the max-player, let the others be min-players.
compute the feasible successor positions v1, . . . , vb of v.
// Let H(v) be the evaluation function for leaves.
if (d == 0 or b == 0) return H(v);
for j := 1 to b
if (max-player has to move) {
α := maximum(α, alphabeta(vj, α, β, d − 1, (i + 1) mod n));
if α ≥ β return α;
if j == b return α;
} else {
β := minimum(alphabeta(vj,α, β, d − 1, (i + 1) mod n), β);
if α ≥ β return β;
if j == b return β;
}
```

Figure 2: Paranoid algorithm pseudocode [Lorenz and Tscheuschner, 2006]

5.2.2.2.3 Best Reply Search

The third algorithm is called Best Reply Search (BRS). The premise of this algorithm is that it determines which of the opponents' moves minimizes the player's score the most, and picks that move to expand while ignoring the other moves. This allows for a deeper search even though the tree has a much higher branching factor compared to the other two algorithms [Schadd and Winands, 2011], because every other level in the search tree is a move for the player. This can also be viewed as a min-max algorithm if there are only two players. This search does not function well with the first heuristic as the opponents' turns do not have an impact on the player's score. The other heuristics will provide behaviour that is beneficial to this strategy. The trade-off made in this algorithm is the ability to more accurately plan for opponent moves, and instead focus on a longer term plan.

While the BRS excels in a game like Chinese Checkers, it is not suited for games where the scoring is determined by what all players play, like Hearts or any trick-based card games [Schadd and Winands, 2011]. This limitation greatly reduces the areas where this search can be extrapolated.

```
1:  BRS(alpha, beta, depth, turn)
2:
3:  if depth <= 0 then
4:          return eval()
5:  end if
6:
7:  if turn == MAX then
8:          Moves += GenerateMoves(MaxPlayer);
9:          turn = MIN;
10: else
11:         for all Opponents do
12:                 Moves += GenerateMoves ;
13:         end for
14:         turn = MAX;
15: end if
16:
17: for all Moves  m do
18:         doMove(m) ;
19:         v = -BRS(-beta,-alpha, depth-1, turn);
20:         undoMove(m) ;
21:
22:         if v >= beta then
23:                 return v;
24:         end if
25:         alpha = max(alpha, v);
26: end for
27:
28: return alpha;
```

Figure 3: BRS pseudocode [Schadd and Winands, 2011]

6 Motivation, Goals and Objectives

6.1 Motivation

The motivation for this problem is to investigate how well different AI algorithms preform against each other, and to continue to explore different multiplayer AI strategies in a perfect-information game. The knowledge gained could be extended to incomplete-information games where additional features are added to assess probability.

6.2 Goals and Objectives

There are three goals and objectives behind this project. The first objective was to become more familiar with multiplayer AI strategies. The first goal was to analyze the performance of different strategies to determine what situations it is most suited in. The final goal was to develop an environment where new strategies could be tested.

The first objective to become familiar with multiplayer AI strategies for information-perfect games is important for a few reasons. Firstly, the same concepts could be applied to information-incomplete games with probability factored in. Secondly, the topic is interesting.

The second goal of analyzing the performance of different strategies to determine which is better under certain circumstances is important because it allows for informed comparison of the algorithms. By making these comparisons, a player can employ different strategies when it is beneficial to do so.

The final goal was to provide an environment where search strategies can be tested. By having this framework in place, a new algorithm can be written and analyzed to know how it compares to other algorithms. This would allow for any new strategies designed to be easily tested and analyzed.

By achieving these goals and objectives, it is hoped that the knowledge of AI approaches is improved and that a framework where new algorithms can be tested is provided.

7 Software Design

7.1 Board

There are three board types: a square board, a triangle board, and a star-shaped board. The square board is limited to just 4-player games as 3-player games would be not be balanced and the scope of the project is such that there is minimal interest in 2-player games. The triangle board is limited to 3-player games as 3-player games are of the most interest to this project. The star board can be played by 2, 3, 4, or 6 players. A 2-player version was included in this version so that the behaviour could be observed. The most interesting case will be the 3-player game, as fewer players allow the search strategies to efficiently search to a depth that would lead to meaningful results.

The board uses a template pattern, which means there is an abstract class which provides basic common functionality while each implementation can provide different behaviours for specific instances. For example, when limiting to the best x children, behaviour is constant regardless of what the board looks like. However, how pieces move is different between boards and differs between implementations, as well as things like where to check to see if a player has won.

The differences in how each board behaves also impacts how the heuristics could be implemented since they would vary from board to board in implementation. Unfortunately this means that the heuristics implemented are heavily coupled with the board-making, meaning implementing additional heuristics is more difficult as they need to be able to analyze all boards.

7.2 Search Strategies

The search strategies use the strategy pattern. This means that each strategy implements a simple interface, which is passed a board, and determines which move to make according to the strategy. By using this design pattern, it is very easy to add new search strategies since interaction is localized. The only changes that need to be made are to a generic player class, and adding in the new search strategy is fairly simple.

One feature added to ensure that all games differed was that when a strategy encountered another board with the same heuristic value, it used the newly-found value 25% of the time. This was chosen over the alternative, where the best move was picked a large amount of the time (say 90%) with a suboptimal move picked the rest of the time [Schadd and Winands, 2011]. While this additional change would create more variability, it also decreases the accuracy of the analysis of algorithms since the optimal strategy is not being performed.

The search strategies are limited to a maximum depth. This was done to provide a reasonable execution time, as well as limit the resources needed to traverse the tree. In other studies the depth that could be traversed was dependent on execution time, which allows for an analysis of how efficiently they can traverse the search tree created [Schadd and Winands, 2011] [Sturtevant, 2002]. Unfortunately this dynamic feature has not been implemented.

7.3 Configuration

The variables used for running the game simulation can be entered in two ways. In the first method, they are retrieved using a command prompt instead of being passed as arguments, as there are too many for that method to be reasonably used. Alternatively a configuration file may be used and passed as an argument to the program. By using a configuration file (see appendix 1 for more details), a simple batch program may be written to run a series of simulations using different configurations. It should be noted that the error handling in this system is not very robust and generally helpful comments are spit out.

An important variable that needs to be configured is the number of children to evaluate; it was limited to the best x moves, where x can be user specified or -1 to allow for all moves to be evaluated. By setting this variable, it reduces the search space significantly by limiting the branching factor to a maximum of x and allows for more moves to be evaluated [Sturtevant, 2002]. This is a very important step in comparing strategies as it allows for significantly more look ahead, which allows for a more meaningful comparison. Note that by limiting the number of children, all children are still evaluated and the best x are selected.

Another variable that can be configured is the board type. There are three boards currently implemented. The first is a square board which can only be played by 4 players. The square board is also set to be a user specified size. The second board is a slightly modified version of

the traditional Chinese Checkers board, the difference being is it smaller and only has 6 pieces as opposed to the traditional 10. The third board is a triangle for 3-player games. The reason behind limiting all boards to 6 pieces is that it limits the size of the search tree as there are fewer children to evaluate.

A player is configured with three variables: what search strategy they use, the maximum height the search tree can be, and which heuristic they use. Currently the player is limited to the search strategies described earlier – max^n, paranoid and BRS – as well as a default random strategy. The random strategy picks a child at random; when the children are restricted to the best children, this will slowly progress towards winning. The maximum height is something that is currently a hard coded value that was found through some experimentation so that it rarely took more than 5 seconds to occur. Lastly, the heuristic for the player to use is set and can be one of the three methods described earlier.

7.4 General Architecture

To start this program, the Driver class needs to be executed. The Driver class is responsible for loading the properties file or prompting for the needed input, as well as housing the game loop. Once started, the user enters in the configuration data as prompted. The program will then run n games specified and write the winner of each to a CSV file.

The game functions in a simple game loop of players' moves. A player will make a move based upon the strategy it is configured to use by building a tree of a depth that is configured. If this move wins the game, then the game ends. If there are more games to be run in the simulation, then a new game will be started; otherwise, the program terminates.

A general decision was to exclude human playability from these implementations. It would have been a nice feature and could be used to test heuristics, but it doesn't achieve any of the goals of the project.

A simple graphical user interface (GUI) is also present that displays an ongoing game. This is not well developed, as it is outside of the main goals of this project. As such, the display for the star board and triangle board is not correctly displayed as there is an offset that is not applied to alternating rows. The GUI was added to allow for a much easier visual aid than the command line interface.

8 Results

For all experiments, the same heuristic was used (hop based). Each scenario investigated is comprised of 30 games.

8.1 2-Player

For all experiments, all algorithms searched to a depth of 7, which was found to take roughly 5 seconds in the mid-game where the search takes the longest to allow for a reasonably paced game. The matchups looked at are: BRS vs. max^n, BRS vs. paranoid, and max^n vs. paranoid. These results differ from those found by Sturtevant, and Schadd and Winands, because in this experiment the search depth is fixed. By fixing the search depth, the inefficient searching of the max^n algorithm is not penalized.

The first matchup examined was the BRS vs. the max^n. The results obtained here, shown in figure 4, are surprising as it is expected that BRS should regularly beat max^n. BRS should act like a min-max and obtain similar results to those as the paranoid vs. max^n.

Player 1	Player 2	Number of wins	Number of wins	Win %
Max^n	BRS	13	17	53.3 – 46.7
BRS	Max^n	14	16	46.7 – 53.3
Total BRS	Total Max^n	31	29	51.7 – 48.3

Figure 4: BRS vs. Max^n

The second matchup investigated was the BRS vs. paranoid with the results shown below in figure 5. Since both algorithms have similar behaviour in a 2-player game, it is a little surprising that BRS has such a good win ratio.

Player 1	Player 2	Number of wins	Number of wins	Win %
BRS	Paranoid	17	13	56.6– 43.3
Paranoid	BRS	12	18	40.0 – 60.0
Total Paranoid	Total BRS	25	35	41.6 – 58.3

Figure 5: BRS vs. Paranoid

The third matchup examined was the max^n vs. paranoid. In this matchup, paranoid excels when it is the first player with no significant difference when max^n is the first player. Since the paranoid algorithm behaves as a min-max search it is expected to win a large amount of these games.

Player 1	Player 2	Number of wins	Number of wins	Win %
Max^n	Paranoid	14	16	46.7 – 53.3
Paranoid	Max^n	20	10	66.7 – 33.3
Total Paranoid	Total Max^n	36	24	60.0 – 40.0

Figure 6: Paranoid vs Max^n

8.2 3-Player

For all experiments, all algorithms searched to a depth of 6, which was found to take roughly 5 seconds in the mid-game where the search takes the longest to allow for a reasonably paced game. The matchups looked at are BRS vs. max^n, BRS vs. paranoid and max^n vs. paranoid, where

one of the algorithms has 2 players; additionally the case of BRS vs. max^n vs. paranoid was examined. All matchups were run on each of the three-player boards, the triangle and the 3-player version of the star board.

The first matchup of BRS vs. max^n is shown in figures 7 and 8 below. As shown the BRS convincingly beats the max^n players in both situations.

Star					
Player 1	**Player 2**	**Player 3**			
BRS	BRS	max^n	16	8	6
BRS	max^n	BRS	19	8	3
max^n	BRS	BRS	11	13	6
Total max^n:	25	**Total BRS:**	65	**Win Rate:**	27.8 – 72.2
Triangle					
BRS	BRS	max^n	14	9	7
BRS	max^n	BRS	21	2	7
max^n	BRS	BRS	11	3	16
Total max^n:	20	**Total BRS:**	70	**Win Rate:**	37.7 – 62.3
Total max^n:	45	**Total BRS:**	135	**Win Rate:**	25 – 75

Figure 7: 2 BRS vs. 1 max^n

Star					
Player 1	**Player 2**	**Player 3**			
BRS	max^n	max^n	19	8	3
max^n	BRS	max^n	7	16	7
max^n	max^n	BRS	5	11	14
Total BRS:	49	**Total max^n:**	41	**Win Rate:**	54.4 – 45.6
Triangle					
BRS	max^n	max^n	15	6	9
max^n	BRS	max^n	14	14	2
max^n	max^n	BRS	14	2	14
Total BRS:	43	**Total max^n:**	47	**Win Rate:**	47.7 – 52.3
Total BRS:	92	**Total max^n:**	88	**Win Rate:**	51.1 – 48.9

Figure 8: BRS vs. 2 max^n

The next matchup of BRS vs. paranoid is shown in figures 9 and 10 below. As you can see, the BRS tends to do fairly well against the paranoid algorithm.

Star					
Player1	**Player 2**	**Player 3**			
BRS	paranoid	paranoid	18	6	6
paranoid	BRS	paranoid	14	10	6
paranoid	paranoid	BRS	14	10	6

Total BRS:	34	**Total Paranoid:**	56	**Win Rate:**	37.8 – 62.2
Triangle					
BRS	paranoid	paranoid	14	11	5
paranoid	BRS	paranoid	10	10	10
paranoid	paranoid	BRS	11	4	15
Total BRS:	39	**Total Paranoid:**	51	**Win Rate:**	43.3 – 56.7
Total BRS:	73	**Total Paranoid:**	107	**Win Rate:**	40.6 – 59.4

Figure 9: 2 paranoid players vs. a BRS player

Star					
Player 1	**Player 2**	**Player 3**			
BRS	paranoid	BRS	14	10	6
BRS	BRS	paranoid	13	11	6
paranoid	BRS	BRS	13	9	8
Total Paranoid:	29	**Total BRS:**	61	**Win Rate:**	32.2 – 67.8
Triangle					
BRS	paranoid	BRS	12	5	13
BRS	BRS	paranoid	11	10	9
paranoid	BRS	BRS	7	19	4
Total Paranoid:	21	**Total BRS:**	69	**Win Rate:**	23.3 – 76.6
Total Paranoid:	50	**Total BRS:**	130	**Win Rate:**	27.8 – 72.2

Figure 10: A paranoid player facing 2 BRS players

The next pairing is max^n vs. paranoid shown in figures 11 and 12 below. Interestingly there is a very big gap between the performances of the max^n on the two boards in figure 12. However, overall the paranoid algorithm seems to preform slightly better than the max^n.

Star					
Player 1	**Player 2**	**Player 3**			
max^n	paranoid	max^n	12	14	4
max^n	maxn	paranoid	9	17	4
paranoid	max^n	max^n	12	8	10
Total Paranoid:	30	**Total max^n:**	60	**Win Rate**	33.3 – 66.7
Triangle					
max^n	paranoid	max^n	17	11	2
max^n	max^n	paranoid	10	11	9
paranoid	max^n	max^n	14	11	5
Total Paranoid:	34	**Total max^n:**	56	**Win Rate**	37.7 – 62.3
Total Paranoid:	64	**Total max^n:**	116	**Win Rate**	35.6 – 64.4

Figure 11: A paranoid player facing two max^n players

Star					
Player 1	**Player 2**	**Player 3**			

max^n	paranoid	paranoid	12	15	3
paranoid	max^n	paranoid	12	10	8
paranoid	paranoid	max^n	5	11	14
Total max^n:	36	**Total Paranoid:**	54	**Win Rate**	40 – 60
Triangle					
max^n	paranoid	paranoid	11	12	7
paranoid	max^n	paranoid	19	4	7
paranoid	paranoid	max^n	10	15	5
Total max^n:	20	**Total Paranoid:**	70	**Win Rate**	22.2 – 77.8
Total max^n:	56	**Total Paranoid:**	124	**Win Rate**	31.1 – 68.9

Figure 12: 2 paranoid players facing a max^n player

The results that allow us to compare the algorithms in a more broad sense are shown in figure 13. As you can see, the BRS excels on the triangle board, while performing just over average on the star board. On the star board the wins are fairly evenly distributed interestingly.

Star					
Player 1	**Player 2**	**Player 3**			
BRS	max^n	paranoid	13	16	1
BRS	paranoid	max^n	14	8	8
max^n	BRS	paranoid	10	14	6
max^n	paranoid	BRS	10	14	6
paranoid	BRS	max^n	12	11	7
paranoid	max^n	BRS	22	4	4
Total Paranoid:	63	**Total max^n:**	55	**Total BRS:**	62
Win Rate	35	**Win Rate**	30.6	**Win Rate**	34.4
Triangle					
BRS	max^n	paranoid	19	2	9
BRS	paranoid	max^n	17	7	6
max^n	BRS	paranoid	16	9	5
max^n	paranoid	BRS	13	6	11
paranoid	BRS	max^n	5	17	8
paranoid	max^n	BRS	12	6	12
Total Paranoid:	44	**Total max^n:**	51	**Total BRS:**	85
Win Rate	24.4	**Win Rate**	28.3	**Win Rate**	47.2
Total Paranoid:	107	**Total max^n:**	106	**Total BRS:**	147
Win Rate	29.7	**Win Rate**	29.4	**Win Rate**	40.8

Figure 13: Max^n vs. BRS vs. Paranoid

8.3 4-Players

For all experiments, all algorithms searched to a depth of 6, which was found to take roughly 5 seconds in the mid-game where the search takes the longest to allow for a reasonably paced

game. The main matchup looked at was the max^n vs. paranoid, where there are two players for each algorithm and the case where they start opposite each other is not investigated. All matchups were run on the two four-player boards, the square board and the 4-player version of the star board.

Player 1	**Player 2**	**Player 3**	**Player 4**				
max^n	BRS	BRS	max^n	2	12	14	2
BRS	max^n	max^n	BRS	5	4	2	19
Total max^n	10	**Total BRS**	50	**Win Rate**	16.7 – 83.3		

Figure 14: BRS vs. max^n on a square board

Player 1	**Player 2**	**Player 3**	**Player 4**				
BRS	paranoid	paranoid	BRS	4	5	10	11
paranoid	BRS	BRS	paranoid	8	10	6	6
Total BRS	31	**Total paranoid**	29	**Win Rate**	51.7 – 48.3		

Figure 15: Paranoid vs. BRS on a square board

The max^n against paranoid match-up is being looked at. As expected the paranoid algorithm does fairly well against the max^n players, for the same reason it does well in the two player version.

Star							
Player 1	**Player 2**	**Player 3**	**Player 4**				
paranoid	max^n	max^n	paranoid	13	9	5	3
max^n	paranoid	paranoid	max^n	7	12	5	6
Total max^n	27	**Total paranoid**	33	**Win rate**	45 – 55		
Square							
Player 1	**Player 2**	**Player 3**	**Player 4**				
max^n	paranoid	paranoid	max^n	9	8	7	6
paranoid	max^n	max^n	paranoid	12	2	4	12
Total max^n	21	**Total paranoid**	39	**Win rate**	35 – 65		
Total max^n	48	**Total paranoid**	72	**Win rate**	40 – 60		

Figure 16: 4-player max^n vs paranoid results

9 Conclusions

The goals and objectives outlined for in this project were accomplished to differing extents. By implementing the three algorithms – max^n, paranoid and BRS – we became familiar with these algorithms. This allowed the first objective to be met very well. The results section shows a very crude comparison of algorithms. These results are not backed up by statistical analysis as was initially hoped. However, this can be done on future data sets, so this goal, while not fully achieved, is present. Lastly, the framework developed does make it fairly easy to implement new strategies as they are discovered. Thus, the last goal was met successfully.

In general it makes sense that the BRS tends to perform very well especially in games with more players, as the algorithm allows for a much farther look ahead which generally means better planning.

The results are not consistent with those gathered by Sturtevant or Schadd and Winands. One reason for the difference is the fact that a different heuristic was used here than in the two other works. Additionally, the results here are from much smaller runs which make them more susceptible to variability than the larger runs done by other parties. The second major difference is in the approach to the depth of the search tree; in this project it is limited, where in the aforementioned works it is a variable that is studied, which greatly benefits the max^n algorithm. The max^n algorithm benefits by allowing the player a more accurate prediction (all the implemented algorithms will make the best move) to do better.

9.1 Future Work

A better heuristic could be implemented, although the current heuristic works well enough. The improved heuristic would lend itself more to allowing opponents' moves to minimize the player's score, thus getting the full benefit of the paranoid and BRS strategies. This could be a blend of the second heuristic with the hop counting provided in the third heuristic. As such possible refactoring to how heuristics are calculated could be done to minimize coupling with the board, and to allow for easier integration of future heuristics.

Additionally, implementing a time limit per player's turn so that the amount an algorithm can look ahead would provide more in-depth analysis. This is already implemented in the versions by Sturtevant, as well as Schadd and Winands.

As for future work, a few things can be done without any future development. A larger data set should be gathered, and cases that were ignored should be run (for example 4-player games with all 3 algorithms). As well, an actual statistical analysis needs to be done on the data gathered. This would also allow for comparison between matchups on different boards. Also an analysis could be done on whether the player number assigned has an impact on the algorithm's performance. Indications are that player 1 tends to win more often than the other players.

The next step in developing the framework for further use would be to add more games to allow for a comparison across different games. This would be similar to the experiments done by Sturtevant on Chinese Checkers, Hearts, and Spades. This also allows for the ability to easily identify if an algorithm does not work well under different conditions, such as with the BRS described earlier.

A new algorithm for perfect-information games could be designed. It would be limited, similar to the BRS, and would be able to take the maximum move in a probabilistic fashion. This would stop algorithms, like the max^n, from knowing which move you are going to make and including it in their plans. An idea would be to monitor the history of an opponent if they continually make the optimal move. A function would evaluate how continuously the opponent depends on the player to make a move of great mutual benefit, and return a probability for you to not make that move. This would be a departure from search strategies and rely more on game theory to make a competitive adaptive player.

10 References

Korf, R.: Multiplayer alpha-beta pruning. Artificial Intelligence. 48 (1991) 99–11

Knuth, D., Moore, R.: An analysis of alpha-beta pruning. Artificial Intelligence. 6 (1975) 293–326

Luckhardt, C., Irani, K.: An algorithmic solution of *N*-person games. In: AAAI 1986, vol. 1, pp. 158–162 (1986)

Lorenz, U., Tscheuschner, T.: Player Modeling, Search Algorithms and Strategies in Multi-player Games. ACG 2006: 210-224

Schadd, M.P.D.; Winands, M.H.M.: "Best Reply Search for Multiplayer Games," *Computational Intelligence and AI in Games, IEEE Transactions on* , vol.3, no.1, pp.57-66, March 2011

Sturtevant, N.: A Comparison of Algorithms for Multi-Player Games, Proceedings Computers and Games. 2002.

Appendix I: Configuration file

Property Name	Value	Description	Mandatory
stats.file	filename.csv	Enter the filename for the winner of the game to be written to.	Always
stats.runNTimes	n	"n" is an integer for how many games you want to run	Always
board.type	triangle, square or star	This is what type of board you wish to run the simulation on	Always
board.numPlayers	n	"n" is an integer for how many players to play	When board.type=star
board.size	n	"n" is an integer the size of the board to play on (nxn)	When board.type=square
board.limitToNMoves	n	"n" is an integer for how many children to limit the board to. -1 allows for infinite children	Always
player.X.playerType*	maxn, paranoid, bestReply or random	Which algorithm the player should use.	Always
player.X.heuristic	1, 2, or 3	See heuristic section for more detail on each	Always
player.X.lookahead*	n	"n" is an integer for how many players to play	Always

* X is an integer starting at 0 going for the number of expected players.

```
stats.file = T3MMB.csv
stats.runNTimes = 30

board.type = triangle
board.numPlayers = 3
board.limitToMoves = 6

player.0.playerType = maxn
player.0.lookahead = 6
player.0.heuristic = 3

player.1.playerType = maxn
player.1.lookahead = 6
player.1.heuristic = 3

player.2.playerType = bestReply
player.2.lookahead = 6
player.2.heuristic = 3
```

Figure 17: Sample configuration file

Appendix 2: DVD Contents

- HonoursProject – Eclipse workspace containing the project source
- Stats – A directory containing all of the .csv files filled with data for runs
- Config – A directory containing all of the .cfg files used in execution
- HonoursProjectReport.pdf – this document
- Cover_img.png – a screen capture of a 4-player game on a square board