Project 2.1

Giulio Fortini, Andrew Gold, Raimondo Grova, Michael Gueye

23 January 2018

Abstract

This paper outlines several Artificially Intelligent agents used to play the board game Ingenious, and compares the efficacy of game tree search algorithms in imperfect information environments to purely greedy and random placements. Indeed, given sufficient knowledge of the domain and current and future states, the search algorithm ExpectiMax outperforms purely greedy or random strategies due to the tactical strength of its evaluation function based on evaluating the probabilities of the opponent's possible moves. This performance comes with significantly slower evaluation times, however, and for every level deeper the game tree is, ExpectiMax can expect a roughly 10 fold increase in computation time.
Keywords: tree-search, AI.

1 Introduction

The purpose of this project is to implement and evaluate several artificial intelligence agents (AI) for the board game Ingenious.
Ingenious was created by Knizia Reiner in 2004. This famous German strategy board game has won several awards such as Schweizer Strategiespiele Preis and the Mensa Select [1]. The broad scope of the game in terms of possible AI implementations, as well as the complexity of the game's branching factor made it a perfect choice for exploring and comparing board game AI efficacy.

1.1 Ingenious Gameplay

Ingenious is a deterministic, zero-sum game with imperfect information that consists of an hexagonal grid where the corners are filled with six different colors of a corresponding shape as shown in Figure 1.

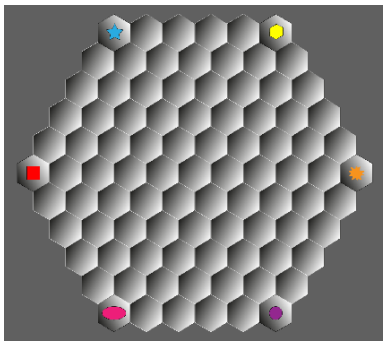


Figure 1: Initial board

It can be played by between two to four players. Each player will receive six tiles from a bag of 120 joint-hexagon tiles. The composition of the bag is shown in Table 1.

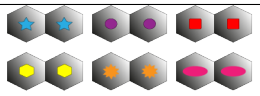
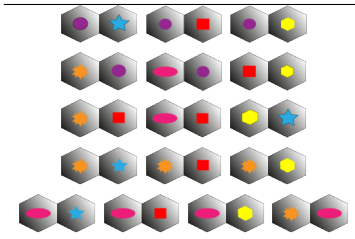
Pieces	Numbers
	30 double color tiles
	90 different color tiles

Table 1: Bag of pieces

At the beginning of the game, the first player will place a tile of his choice on the board. In order for the first move to be valid, the tile has to be placed next to a preset corner. Thereafter, tiles may be placed anywhere on the board. Once the tile is placed, the player will receive points as follows: for each hexagon on the placed tile, the player scores one point in that color

for each similar-color hexagons that lie adjacent to it, extending in a straight line in every directional axis. If a player brings the score of a color to 18, the current player immediately takes another turn [2]. The score of each player is constantly updating so that all the players can scrutinize their actual score or the ones of their opponents. When no other tile can be placed on the board or when a player score 18 in all the six colors, the game ends. The final goal of the game is to have the highest score for the lowest score color. The loser will be the player that has the lowest color score [2]. Therefore the players need to balance their approach such that they prioritize maximizing their score for all colors instead of just a few of them.

The uncertainty of the gameplay is of particular interest. Indeed, players do not have information about the hand of their opponents, yet are aware of their score and therefore are able to reasonably assume which color tiles their opponent will prioritize.

2 Algorithms and Evaluation Functions

The different artificially intelligent AI agents (hereafter referred to as "AI") used in this project and their respective evaluation functions (if applicable) are detailed in this section. Strategic evaluation choices are outlined, and any modifications to canonical algorithm structure is motivated.

2.1 Utility and Evaluation Functions

In this project several strategies to reduce the branching factor of the game tree have been evaluated and implemented. A few utilitarian functions are used to heuristically order or prune sets, moves, or tiles to simplify the evaluation functions. These evaluation functions then inform the search algorithms to maximize efficiency and reduce complexity. Furthermore, due to the imperfect information nature of Ingenious, certain search algorithms such as MCTS and ExpectiMax were adapted to make strong assumptions about the opponent player to reduce complexity and improve runtime.

2.1.1 Point Maximization

Point Maximization as an evaluation function is designed to take a tile from the hand of the player and try to greedily maximize points on the board, prioritizing one of the two colors on the tile (usually the one where the player has a lower score). This strategy is simple and does not take into account the lowest color of the current or opponent player.

2.1.2 Lowest Color Prioritization

Given the hand of the player, tiles that contain at least one of the lowest scoring colors are selected and added to a hashmap. The goal of the game is to maximize the score of every color as evenly as possible, minimizing the chance of the opponent outscoring the gaming player's lowest colors. The rules state that if there exists no tile with the gaming player's lowest color in that player's hand, they may discard their entire hand and redraw a new hand until a tile containing the lowest scoring color is present. This effectively guarantees that each player will hold at least one tile containing the lowest color in their hand at all times. The selection procedure prioritizes doubles of the lowest color, as this would maximize the potential point gain for the player's lowest color. As a whole, the following prioritization heuristics are as follows (in order of importance):

1. Prioritize double color tiles of the lowest color.
2. Choose tile with lowest color + another color.
3. Choose the first tile in the list and place it at a random valid move.

2.1.3 Action Gain

For every action taken such that a tile is placed on the board and a new game state emerges, the action that led to this state change is stored, allowing the Monte-Carlo Tree Search and ExpectiMax algorithms to effectively explore the game tree several ply deep.

2.2 Random

The simplest form of agent implemented is the random algorithm. Indeed, this random will simply try every available placement for each

tile in the hand and place it on the board at the first valid move that it finds. This agent has been implemented as a baseline to compare all other AIs to.

2.3 Greedy

A basic greedy algorithm has been implemented that utilizes the lowest color prioritization scheme described in Section 2.1.2. Based on the highest priority tile in the candidate set, choose the placement that maximizes the score gain of the colors on the chosen tile. Should multiple placements exist with identical score gains, the first placement found is chosen. Should a large number of placements exist with minimal point gains, such as at the beginning of the game, a placement is chosen at random.

Algorithm 1 Greedy Algorithm

```
1: procedure GREEDYSTRATEGY
2:     Action[]actions
3:     for Tile in BestTiles do
4:         actions.add(bestTileAction)
5:     if actions.length == 0 then
6:         actions.add(RandomAction)
7:     for Action a in actions do
8:         if a.gain ≥ bestAction.gain then
9:             bestAction ← a
       return bestAction
```

2.4 ExpectiMax

Ingenious being an imperfect information board game, it becomes impossible for the MiniMax algorithm to be implemented due to its requirement of all knowledge of the game state. A brute-force, depth first search algorithm called ExpectiMax first implemented by Donald Michie in 1996 is implemented that adapts the MiniMax algorithm to be compatible with games of chance based on the 'expected' move an opponent would make to maximize their score [3, 4, 5]. ExpectiMax will average the MiniMax values that are from the chance node that will take into account the probabilities of an event to occur [3, 4]. A simple ExpectiMax game tree is illustrated in Figure 2.

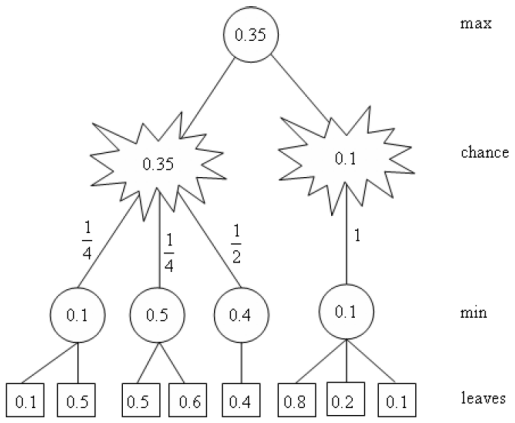


Figure 2: ExpectiMax game tree example [3]

As shown in the figure representing the game tree, the algorithm is represented by four types of nodes.

- The root of the tree is represented by a *max* node which will return the highest value of it's successors. Following the same methodology, the *min* node will return the lowest value of it's children.
- The chance node is the weighted sum of its successors as shown in the illustration of the game tree or can be expressed as mathematical expression as shown in Equation 1. Finally, the last type of node is the leaf node that will contain the values.

$$\sum_{i=1}^{n} Probability(x_i) \times Utility(x_i) \qquad (1)$$

Algorithm 2 ExpectiMax

```
1: procedure EXPECTIMAX(Node, Depth)
2:     if d = 0 or n is a leaf node thenreturn
   Value of node
3:     else if Player plays at n then
4:         α ← −∞
5:         for each c of n do
6:             α ← max(α, expectiMax(c, d −
   1))
7:     elseChance node at n
8:         α ← 0
9:         for each c of n do
10:            α ← α+(P[c]∗expectiMax(c, d−
   1))
       return α
```

2.4.1 Probabilities

In order to determine what the probability of an opponent to have a certain tile in his hand is, the hypergeometric distribution probability mass function is used. The distribution models the total number of successes in a fixed-size sample drawn without replacement from a finite population. The probability mass function is given as:

$$P(X = s) = \frac{\binom{S}{s}\binom{N-S}{n-s}}{\binom{N}{n}} \tag{2}$$

where S is the sample size, s is the number of observed successes from the sample, N is the population size, and n is the number of draws in the population.

Information such as the current bag, the hand of the player and the current board need to be known at all times in order to calculate expectations. The remaining tiles in the bag and the hand of the opponent player are always unknown to the gaming player, however by removing all tiles visible to the player from the set of a full bag, the combined contents of the opponent's hand and the remaining bag is known. From this, the hypergeometric distribution is calculated as follows:

Suppose a hypergeometric distribution is made to find the probability of having a double in the hand. $S = 30$, $s = 1$, $N = 120$, $n = 1$.

$$\begin{aligned} P(X \geq 1) &= \frac{\binom{30}{1}\binom{120-30}{1-1}}{\binom{120}{1}} \\ &= \frac{\frac{30!90!}{(1!(29)!)(0!(90)!)}}{\frac{120!}{1!(119!)}} \\ &= \frac{3}{12} \\ &= 0.25 \end{aligned} \tag{3}$$

Suppose now that the probability for the opponent of having a specific double such as Yellow-Yellow is desired. Assuming that the current turn is the first turn of the opponent and that player one has placed a Yellow-Yellow on the board, the population size of unknown tiles therefore decreases from 114 to 113, and the the sample size is now equal to 4 (the number of a specific double tile in the total population is 5 minus the one already placed on the board, as described in Section 1.1). The resulting probability is:

$$\begin{aligned} P(X \geq 1) &= \frac{\binom{4}{1}\binom{114-4}{1-1}}{\binom{114}{1}} \\ &= \frac{\frac{4!110!}{(1!(3)!)(0!(110)!)}}{\frac{114!}{1!(113!)}} \\ &= \frac{4}{114} \\ &= 0.035 \end{aligned} \tag{4}$$

2.5 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first search algorithm that aims to maximize the overall probability of reaching the goal (winning a game) [6]. It consists of four stages that are repeated until all options have been evaluated, or can be manually stopped after a specific amount of time, returning the most promising action discovered as of yet [6, 7]. Benefits of MCTS over other algorithms such as MiniMax (and ExpectiMax by extension) include the ability to navigate games with large branching factors such as Ingenious [8]. Algorithms such as ExpectiMax by default consider all possible moves from a current state and exhaustively determine the most promising path in a depth-first fashion, however in games with large branching factors such an approach would be severely hindered purely by the amount of computational steps needed to explore all possible options, and therefore in many implementations the game tree is restricted at a certain depth level, or ply [6]. MCTS avoids such pitfalls by examining only the best directly available moves, with several different options for determining these best moves [6]. For the purposes of this project, the best moves are chosen via a method for examining the Upper Confidence bound for Trees (UCT), described in Section 2.5.1.

The four main phases of MCTS are illustrated in Figure 3.

- **Selection**: In this first step, the tree is traversed from the root node to a leaf node via the children of each visited node. How each of these children are selected is a critical element of MCTS, and is explained in Section 2.4.1. Note that the depth (ply) of the leaf node in the tree can vary, but for the purposes of this project the depth is 3.

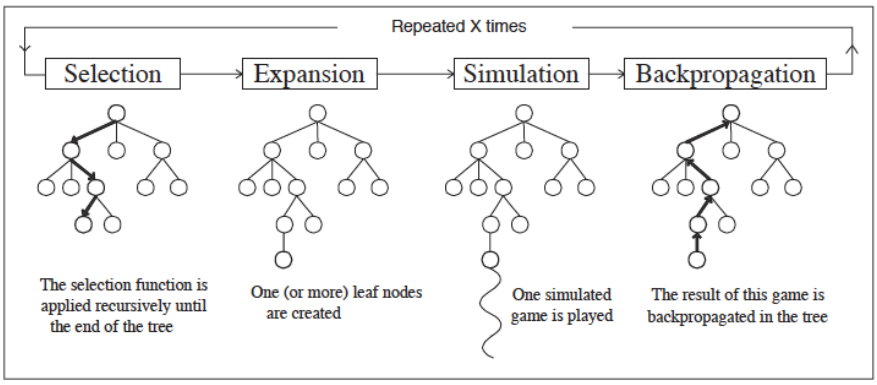


Figure 3: Monte-Carlo Tree Search [6]

- **Expansion**: Once a leaf node has been reached, a child node is created and the roll-out phase begins from this child of the leaf.
- **Simulation (Rollout)**: A complete simulation of the remainder of the game is performed until the point where the end of the game is reached, with most basic MCTS implementations implementing random moves during rollout.
- **Backpropagation**: Finally in the last phase, the result of the simulation is back-propagated to the root of the tree as illustrated in Figure 3.

The results are then compared, and the child of the root with the highest number of wins is chosen as the most promising action. This continues until the end of the game is reached.

2.5.1 Upper Confidence Bound for Trees

Upper Confidence Bound for Trees (UCT) is the most popular evaluation formula in the MCTS family and was introduced by Levente Kcosis and Csaba Szepesvari in 2006 [9]. In MCTS, how the tree is built depends on how nodes in the tree are selected [10]. Let I be the set of nodes reachable from the current node p, then select the child k of the node p that best satisfies the Equation 4:

$$k \in argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{ln\ n_p}{n_i}} \right) \quad (5)$$

where v_i is the value of the node i, C is the exploration parameter (typically defined as $\sqrt{2}$), n_i is the visit count of i, and n_p is the visit [11].

3 Experimental Setup

Listed in this section are the compared results of 100 simulated games between different AIs. First, a win-loss ratio is determined. Second, a runtime analysis is explored. Third, the average score for each algorithm as well as a minimum/maximum overall score is illustrated. Finally, an analytical discussion of the results is given. For the purpose of the experiments, the Random algorithm serves as the baseline to compare all performance against.

Please note that due to time constraints, MCTS was unable to be properly tested and compared to other algorithms.

3.1 Win-loss Ratio

Given below are the results from 100 games from all combinations of agents playing against each other, with the ratio of wins vs losses for each agent.

3.1.1 ExpectiMax vs. Random, Greedy

	Games	Win	Losses
Random	100	100%	0%
Greedy	100	44%	56%

Table 2: ExpectiMax (2-ply) win/loss ratio

ExpectiMax with a depth limit of 2 predictably beats the Random algorithm 100% of the time, however fails to beat Greedy with a win/loss ratio approaching 4:6. This is almost certainly due to the lack of exploration depth, given that

ExpectiMax by nature is a depth-first search algorithm. Considering that the current state exists at level 0, and level 1 represents the possible moves given to the gaming player, ExpectiMax is only able to compare a single future state to each possible option, which severely hinders its innate tactical strength.
This percentage is obtained by computing the mean of 10 sets of 100 hundred games. A statistical approach and calculation is made to provide a 95% confidence interval on how ExpectiMax depth 2 is actually performing against Greedy. The confidence interval is given by a one-sample t-test defined as shown in Equation 6. If the same population is sampled on numerous occasions and interval estimates are made on each occasion, the resulting intervals would contain the true population parameter in approximately 95% of the cases [12].

$$I_c = \left[\bar{x} - t_\alpha \frac{s}{\sqrt{n}}; \bar{x} + t_\alpha \frac{s}{\sqrt{n}}\right] \tag{6}$$

Ten sets of one hundred games are performed and the average result of each algorithm is determined, and a 95% interval is calculated. Results can be seen in Table 3.

WinRate CI	Mean	Wins Interval	CI
Greedy	56.3	[52.843, 59.756]	95%
ExpectiMax ply 2	43.7	[40.244 , 47.156]	95%

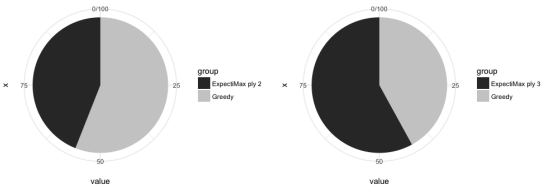
Table 3: ExpectiMax (2-ply) win interval

As it can be seen from Table 3, after the computation of 95% confidence interval, ExpectiMax still loses against Greedy even if the gap between the two algorithm is smaller.

	Games	Win	Losses
Random	100	100%	0%
Greedy	100	58%	42%

Table 4: ExpectiMax (3-ply) win/loss ratio

Using the same methodology of the previous Win/Loss ratio, the mean of 10 sets of 10 games are performed. ExpectiMax with a depth limit of 3 begins to outperform the Greedy algorithm with a win rate of just over half, as it is allowed to explore another level of consequences of its actions, allowing it to regain some of its tactical strength. A confidence interval is also computed. However, due to slow runtime of ExpectiMax at ply 3, fewer simulations are performed and the interval computed in Table 5.



(a) ExpectiMax (2-ply) win/loss (b) ExpectiMax (3-ply) win/loss

Figure 4: 100 Games simulation Win/Loss Ratio

WinRate CI	Mean	Wins Interval	CI
Greedy	4.2	[2.777, 5.622]	95%
ExpectiMax ply 3	5.8	[4.377 , 7.223]	95%

Table 5: ExpectiMax (3-ply) win interval

As there were only 10 games for this particular experiment, the confidence interval is larger. However, what can be deduced from Table 5 is that ExpectiMax ply 3 is playing better than Greedy in most simulations (Appendix Table 13). As expected, due to its exploration depth, the ExpectiMax becomes more effective as it explores deeper in the game tree.
Expanding ExpectiMax to a depth limit of 4 proved impossible within the time constraints of the project deadlines, as there was a tenfold increase in the average time it took to compute and execute a single move between levels 2 and 3 (as detailed in section Section 3.2), and testing 100 games was unrealistic within the given time constraints.

3.1.2 Greedy vs Random

	Games	Win	Losses
Random	100	100%	0%
Greedy	100	53%	47%

Table 6: Greedy win/loss ratio

Greedy predictably beats Random 100% of the time, and just as predictably hovers around a 50% win rate against itself. It is reasonable to suggest that the win/loss ratio for Greedy vs. Greedy would approach 1:1 as the number of iterations grows large.

3.2 Runtime Analysis

Below the minimum, maximum, and average amount of time that take a move to be performed is shown (in milliseconds).

	Games	Avg.	Min.	Max.
Random	100	1.88	1	43
Greedy	100	2.41	1	52
ExpectiMax ply 2	100	196.63	10	919
ExpectiMax ply 3	10	2191.97	10	6069

Table 7: Runtime per moves

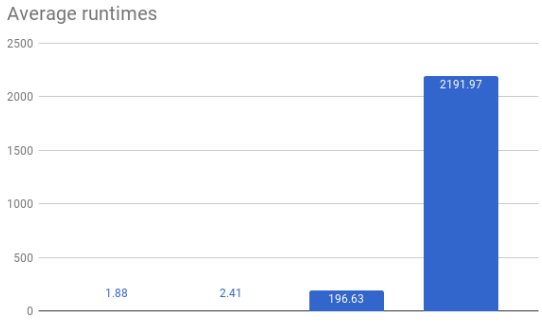


Figure 5: Average Runtimes Chart

Clearly, the random choice is fastest as it relies on no heuristic evaluations to determine the move as seen in Figure 4 and Table 6. Instead, it simply chooses at random from a list of possible moves. Greedy is only marginally slower in determining an optimal move based on point gain calculations, and the results are impressively close to the Random algorithm. ExpectiMax with ply 2 is approximately 100 times slower than greedy, due to its exhaustive exploration of the game tree available to it. By adding another level of depth to ExpectiMax, the average time taken increases by tenfold, which suggests that for the time it takes to explore each node in a single level of ExpectiMax, it takes approximately 10 times that amount to explore the children for each node at the level below.

To illustrate the computation time needed to determine the next action to take, Figure 6 shows a scatter plot of Greedy's decisions and the time it takes to compute them over 100 games. Table 7 demonstrates that the gap between the minimum move runtime and the maximum often varies widely. This might be due to several factors such as an algorithm achieving an Ingenious after a move, or simply represents

Figure 6: Scatter plot of Greedy RunTime

of the beginning of the game where more actions need to be evaluated.

3.3 Gain Testing

In the final experiment, the point gain a player obtains for each move is examined. It is evaluated based on a coefficient assigned to the tiles in the player hand, which are sorted by importance in the current turn.

	Games	Avg.	Min.	Max.
Random	100	9.69	0	60
Greedy	100	9.78	0	54
ExpectiMax ply 2	100	9.63	0	30
ExpectiMax ply 3	10	9.50	0	30

Table 8: Random Gain

The Random agent's average score distribution per move is very closely distributed across all algorithms, which is expected due to the random nature of the choices.

	Games	Avg.	Min.	Max.
Random	100	18.91	0	66
Greedy	100	21.78	0	72
ExpectiMax ply 2	100	20.95	0	60
ExpectiMax ply 3	10	19.91	0	54

Table 9: Greedy Gain

As can be seen above in Table 7, the Greedy algorithm returns the best point gain per move against itself compared to all other algorithms. This is due purely to Greedy's non-tactical point maximization preference. There is also very little difference between the algorithms, as Greedy's evaluation is relatively unimpacted by the opponent agent's choices.

	Games	Avg.	Min.	Max.
Random	100	19.75	6	44
Greedy	100	21.01	6	34
ExpectiMax ply 2	100	21.84	10	46

Table 10: ExpectiMax ply 2 Gain

ExpectiMax with depth 2 averages a relatively normal point distribution as well, however the lower boundary against greedy is noticeably higher than for other moves, with almost double the minimum number of points gained per move than against itself, and more than triple the minimum number of points gained against Random.

	Games	Avg.	Min.	Max.
Random	100	22.10	6	41
Greedy	100	22.77	6	48

Table 11: ExpectiMax ply 3 Gain

As ExpectiMax with depth 3 is significantly slower, only Random and Greedy were compared. With this additional level of depth, ExpectiMax is able to further improve upon its per-turn performance against Greedy and Random. The extra level of depth however seemingly has very little overall effect on the average point gain per turn.

4 Conclusion

In conclusion, it is clear that the more informed an algorithm is of the current and future states, the more effective its search becomes. ExpectiMax outperforms Greedy and Random given that ExpectiMax has at least 3 levels. For a significant tradeoff in efficiency, Greedy still performs well tactically against shallow implementations of ExpectiMax.

5 Future Research

Given the opportunity for further exploration, several augmentations to existing algorithms exist. Namely, a promising adaptation of Monte-Carlo Tree Search would be to integrate the ExpectiMax algorithm to inform both the expansion phase (as a move-selection bias) and as a replacement for the rollout phase, as described by Baier & Winands, however replacing their implementation of MiniMax with ExpectiMax [13]. Due to MCTS' need to balance exploration with exploitation, the strategic ability of MCTS is counterbalanced by a relative lack of tactical ability [13]. By allowing ExpectiMax to inform the selection process of MCTS, short-term tactical ability could possibly be improved. Furthermore, in order to improve upon the random rollout of MCTS, ExpectiMax could be played during the simulation phase which could lead to more significantly improved estimations of playout results, compared to the unpredictable and uninformative nature of a random playout.

In addition to MCTS augmentations, integrating Alpha/Beta pruning with ExpectiMax may lead to significantly improved results. Utilizing the already existing move ordering capabilities of the evaluation functions listed in Section 2.1, further performance improvements could possibly be achieved in addition to A/B pruning.

References

[1] R. Hyde, "Ingenious game." http://ingeniousgame.com, 2008.

[2] Unknown, "Ingenious board game." https://boardgamegeek.com/boardgame/9674/ingenious.

[3] J. Veness, "Expectimax enhancements for stochastic game players," 2006. The University of South Wales School of Computer Science and Engineering, Bachelor Thesis with Honours.

[4] S. Russel and P. Norving, *Artificial Intelligence A Modern Approach*. Pearson, 3 ed., 1994.

[5] D. Michie, "Game-playing and game-learning automata," *L. Fox (ed.), Advances in Programming and Non-Numerical Computation*, pp. 183–200.

[6] G. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," *Computers and Games, 6th International Conference*, pp. 61–71, 2008. Springer.

[7] G. Chaslot, M. H. M. Winands, I. Szita, and H. van den Herik, "Cross-entropy for monte-carlo tree search," *ICGA journal*, vol. 31, pp. 145–156, 9 2008.

[8] J. Bradberry, "Introduction to monte carlo tree search." https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/, 2015.

[9] L. Kocsis and C. Szepesvari, "Bandit based monte-carlo planning," *Computer and Automation Research Institute of the Hungarian Academy of Sciences*, pp. 13–17.

[10] C. Browne, E. Powley, D. WhiteHouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, 2012.

[11] G. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for monte-carlo tree search," 2008.

[12] NIST and SEMATECH, "e-handbook of statistical methods." http://www.itl.nist.gov/div898/handbook/, 2012.

[13] H. Baier and M. H. M. Winands, "Monte-carlo tree search and minimax hybrids with heuristic evaluation functions," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, pp. 167–179, 2015.

6 Appendix

ExpectiMax	Greedy
38	62
43	57
43	57
48	52
44	56
35	65
51	49
47	53
41	59
47	53

Table 12: DataSet ExpectiMax ply 2 vs Greedy

ExpectiMax	Greedy
7	3
5	5
5	5
7	3
10	0
5	5
3	7
4	6
5	5
7	3

Table 13: DataSet ExpectiMax ply 3 vs Greedy