# AI Implementations for Hex

Group 5: A. Steckelberg, N. Vaessen, J.L. Velasquez, J. Vermazeren, T. Wall, X. Weber

January 24, 2017

## Abstract

Hex is a classic board game invented in 1942 by Piet Hein and independently by John Nash in 1948. In this paper there is research into Alpha-Beta and Monte-Carlo Tree Search (MCTS) Hex players, as well as comparisons of different evaluation functions which includes Dijkstra and Electric Circuit. In addition to this Monte-Carlo Tree Search specific heuristics, Upper Confidence Threshold (UCT) and All Moves As First (AMAF), are compared. The most optimal versions of every implementations were found, showing that the UCT policy in the MCTS algorithm has a better performance over all the other techniques tested.

## 1 Introduction

Hex is a well-known board game first created in 1942 by Piet Hein, a Danish physicist, with later improvements being made by John F. Nash in 1948. It gained the name Hex in 1952 when a version of the game was released by the firm Parker Brothers, Inc [8]. The basic idea of Hex is to create a bridge across a diamond shape board of hexagons. A sample of a Hex board can be seen in figure 1.

Shannon and Moore have developed the first Hex playing machine in 1953.[13] Subsequent approaches were based on Shannon's Hex playing machine and a tradition of AI for playing Hex using Alpha-Beta search started with Anshelevich and Hexy in 2000.[1] Alpha-beta search based bots have dominated international competitions until the Monte-Carlo Tree search based player MoHex was developed in 2007 by Arneson, Hayward and Henderson.[2]

This paper investigates both Alpha-Beta based Hex players and Monte-Carlo based Hex players. Both Alpha-Beta and Monte-Carlo based Hex players have been implemented by the authors of this article based on Hexy and MoHex. The aim was to first improve the performance of both players individually and then compare them with each other.

To demonstrate this first a short overview of rules of Hex is given. Second, several algorithms to play Hex are described. Third, the experiments conducted are explained and the choice of algorithms and experiments is motivated. Then, the results are discussed and conclusions are drawn subsequently. In the last section further research topics are suggested.

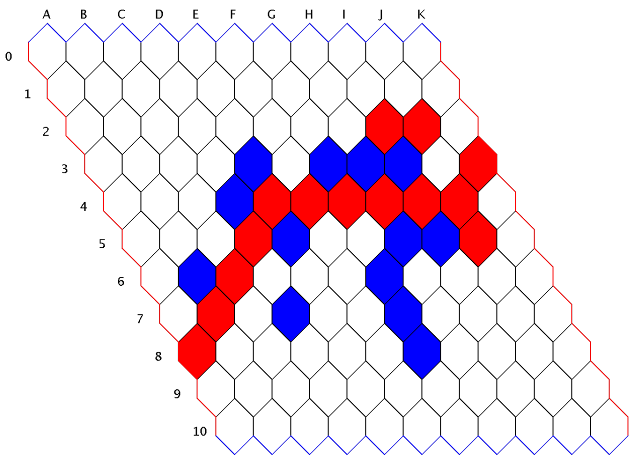

Figure 1: A sample hexboard

## 2 Rules of Hex

The rules of Hex are simple. Each player is given a colour, typically blue and red or black and white. The players then take turns placing tiles down on the board to claim a space. The aim is to create a path from one side of the board to the other, either top to bottom or left to right.

The generally accepted normal size of the board is 11 × 11, however it can be played on boards of differing sizes. Nash has shown that there is always a winning strategy for the first player. The first player is at a distinct advantage over the second player.[11] Hence, the Swap Rule was introduced, which states that after the second player has made their first move they may swap the positions of the first two tiles.

# 3 Algorithms

## 3.1 Game tree

Game trees closely follows a *Markov Decision Process* (MDP) in which every node of the tree is represented by a State $s \in S$ and the connection between the nodes is represented by an action $a \in A$. [4]

- $S$: a set of all possible states of the board
- $A$: a set of all possible actions that can transition from one state to the other
- $P_a(s, s')$: the probability of reaching state $s'$ by using action $a$ on state $s$
- $R_a(s, s')$: the reward for reaching state $s'$ by using action $a$ on state $s$

## 3.2 Alpha-beta

MiniMax search with $\alpha$ - $\beta$ pruning is a standard approach to deterministic games with perfect information and Hex is such an environment. MiniMax search investigates future positions of the board and evaluates them. It was used by the first Hex playing machine by Shannon and Moore [13] and Alpha-Beta based Hex players have dominated Hex competitions until 2008.[2]

MiniMax search determines what the best move in each position is using a game tree. MiniMax evaluates the game tree using an evaluation function from the perspective of one player, referred to as MAX. The opponent is referred to as MIN. [12] MiniMax is a depth-first search. The game tree is evaluated from the leaf towards the root. The root of the tree is associated with MAX. Each level of the tree is associated with a player in an alternating way. Nodes with an odd depth are associated with MIN and nodes with an even depth are associated with MAX.

MAX nodes receive the maximal evaluation of the nodes one level below it as MAX aims to maximize utility. MIN nodes receive the minimal evaluation of the nodes one level below it as MIN aims to minimize the utility of MAX. [12]

$\alpha$ - $\beta$ pruning is a strategy to simplify the search. The idea behind Alpha-Beta pruning is to prune subtrees for which we know that they do not yield a better strategy. "Consider a node n somewhere in the tree (...) such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n, or at any choice point further up, then n will never be reached in actual play. So, once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it." [12]. This strategy increases the efficiency of the search. Pseudocode for the Minimax algorithm with Alpha-Beta pruning is provided below. (see Algorithm 1)

**Algorithm 1** Alpha-beta [12]

```
function MAX-VALUE($stategame, \alpha, \beta$) returns the minimax value of $state$
    inputs: $state$, current state in game
    $game$ , game description
    $\alpha$, the best score of MAX along the path to $state$
    $\beta$, the best score for MIN along the path to $state$

    if CUTOFF-TEST($state$) then
        return EVAL($state$)
    end if
    for each $s$ in SUCCESORS($state$) do
        $\alpha \leftarrow$ MAX($\alpha$, MIN-VALUE($s$, $game$, $\alpha$, $\beta$))
    end for
end function

function MIN-VALUE($state, game, \alpha, ft$) returns the minimax value of $state$

    if CUTOFF-TEST($state$) then
        return EVAL($state$)
    end if
    for each $s$ in SUCCESSORS($state$) do
        $ft \leftarrow$ MIN($ft$, MAX-VALUE($s, game, \alpha, \beta$))
        if $ft < a$ then return $a$
        end if
    end for
end function
```

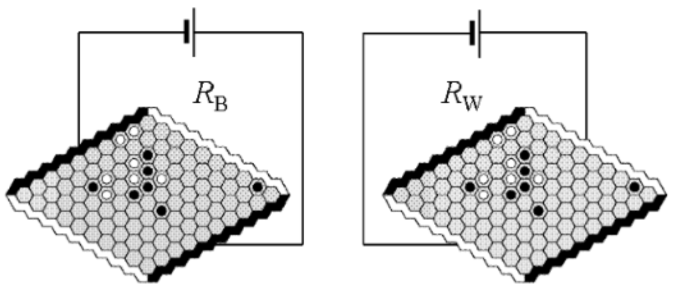

Figure 2: Black's and White's circuits[1]

## 3.3 Evaluation Functions

Shannon and Moore[13] model the board as an electrical circuit which overcomes this limitation as it considers all paths in parallel. An illustration of that by Anshelevich can be seen in figure 2. Their ideas have been used by Anshelevich[1] to develop the Hex player Hexy. The board is represented as a graph. Positions on the board are vertices and adjacent positions are connected by edges. Edges of the graph are resistors and resistance is assigned in the following way:

For Black's circuit:[1]

$$r_B(c) = \begin{cases} 1, & \text{if } c \text{ is empty,} \\ 0, & \text{if } c \text{ is occupied by a black piece,} \\ +\infty & \text{if } c \text{ is occupied by a white piece.} \end{cases}$$

And likewise, for the circuit of the other player . One can see that positions occupied by the opponent are not a part of the circuit as they have an infinite resistance. A voltage is then applied to the two sides that the player needs to connect. [1].

By calculating the equivalent resistance of the board an evaluation of the board is achieved. The lower the resistance, the better the evaluation. The equivalent resistance can be calculated using a system of linear equations based on Kirchhoffs Current Law. [1]

H. Challup, Mellor and Rosamund [5] also used Dijkstras algorithm for shortest path finding in a graph. In contrast to the previous evaluation this evaluation only considers a single path and not the whole board. The board is represented as a graph. Positions on the board are vertices and adjacent positions are connected by edges. The edges are weighted using the following evaluation. (1. Dijkstra evaluation [5]) Positions occupied by the opponent do not belong to the graph.

$$W(p_1, p_2) = \begin{cases} 0, & \text{if } p_1 \text{ and } p_2 \text{ are both occupied} \\ & \text{by the player,} \\ 1, & \text{if one out of } p_1 \text{ and } p_2 \text{ is} \\ & \text{occupied by the player,} \\ 2 & \text{if both positions are unoccupied.} \end{cases}$$
(1. Dijkstra evaluation [5])

The shortest path between the two sides of the board that need to be connected by the player is calculated. Shorter paths are preferred by the player.

## 3.4 Virtual Connection and H-Search

Another interesting technique for Hex playing is the search fo virtual connections devised by Anshelevich and used by Hexy.[1] This article only touches upon virtual connections as they were not used in the implementation of the Alpha-Beta algorithm. Hexy combines both Alpha-Beta search and the previously mentioned electrical circuit evaluation function with this approach. Virtual connection on a Hex board are positions occupied by one of the players which can always be connection with one another i.e. the opponent is able to prevent these positions or paths from being connected.[1]

H-search is a search that finds virtual connections using the AND- and OR- deduction rule devised by Anshelevich.[1] H-search first creates a first generation of virtual connections which are all neighbouring cells and subsequently applies the deduction rules to find new virtual connections.[1] It is important to note that H-search is not a game tree search but rather searches for virtual connections in sub-games.[1]

Hexy uses H-search to create a hierarchy of virtual connections which are then evaluated by the evaluation function above.[1] This increases the efficiency of the Alpha-Beta search search, because H-search narrows the space for Alpha-Beta search to connections which are achievable to matter the interference of the opponent. Hexy with H-search considers the play of the opponent to a greater extend than without H-search and speeds up the Alpha-Beta search.

## 3.5 Monte-Carlo Tree Search

A powerful AI technique for board games is the Monte-Carlo Tree Search(MCTS). It applies optimization techniques of random simulations to the game tree.

For Hex, the probability of reaching a state for any action is 1 as long as it is a legal move. The MCTS algorithm can be divided into four main parts (see figure 3):

1. Selection: In this part the algorithm looks for the most urgent node with a state $s$ that has not being visited before.

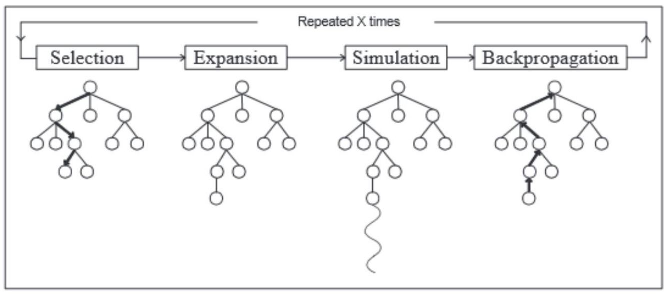

Figure 3: The four steps of Monte-Carlo Tree Search [6]

2. Expansion: one or more nodes are expanded by adding leaf nodes according the values left in $A$.
3. Simulation: a simulation takes place in order to assign a reward value to the newly added node/s.
4. Backpropagation: the statistics of the tree are updated through backpropagation.

These four simple parts of the algorithm can be divided into two policies that simplify the process.

- *Tree Policy*: in this policy the stages of selection and expansion take place. The random nodes are selected and expanded, ready to receive their reward value.
- *Default Policy*: in this policy the simulation takes place and a reward value is given to the node

The backpropagation stage of the MCTS does not occur within any policy as it just represents the update of the tree after the reward, which is used in later cycles for the tree policy to select and expand new nodes.

**Algorithm 2** Monte-Carlo Tree Search [4]

```
procedure MCTS($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TreePolicy($v_0$)
        $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
        Backup($v_l, \Delta$)
    end while
    return $a$(BestChild($v_0$))
end procedure
```

In Algorithm 2 we see the general structure of the MCTS. The root of the tree will be the board state at the beginning current turn. After the tree has fully expanded according to computational budget using the repeated MCTS process explained before, the best child is selected. We return the action $a$ which connects the current state $s$ to the best child.

## 3.6 Improvements on General MCTS

Different heuristics can be used on top the MCTS algorithm to enhance its performance.

**Simulations per Iteration (SPI)**

In MCTS algorithm(Algorithm 2) the default policy does 1 simulation of the expanded child. Using more than one simulation on the expanded note might be viable way to improve the results for Hex because the default simulation for hex is straightforward.

**Upper Confidence Bounds for Trees (UCT)**

The UCT is an algorithm which adapts the Upper Confidence Bounds (UCB) exploitation and exploration of trade-offs as a tree policy. Selecting a child node to traverse in the selection part of MCTS can be seen as a multi-armed bandit problem.[3]

The UCT tree policy works by choosing the node $v$ that maximizes the independent multi-armed bandit problem:

$$UCT = \bar{X}_v + C_p\sqrt{\frac{\ln n}{n_v}}[4]$$

In the equation above $n$ represents the amount of time the current node has being visited, $n_v$ the amount of time a child node has been visited, and the constant $C_p \geq 0$ can be used to weight the importance of exploration. If a child is not visited ($n_v = 0$) the exploration term is set to $\infty$.

In the UCT policy we see the general MCTS algorithm structure of *selection, expansion, simulation* and *backpropagation.* However, as we can see in Algorithm 3, the UCT policy affects the Tree policy of the algorithm. The heuristic used to get the best child $v'$ to expand from $v$ is done via multi-armed bandit problem. What this creates is an asymmetrical tree that expands only in what the algorithm's exploration value ($C_p$) allows it to. On one hand, if the $C_p$ value is too small the expansion would not have enough children. On the other hand, if the $C_p$ value is too large the search will not reach a desirable depth, making also under perform.

**Algorithm 3** Upper Confidence Threshold [4]

```
function UCTSearch($s_0$)
    $v_0 \leftarrow$ create root node from $s_0$
    while within computational budget do
        $v_l \leftarrow$ TreePolicy($v_0$)
        $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
        Backup($v_0, \Delta$)
    end while
    return $a$(BestChild($v_0, 0$))
end function

function TreePolicy($v$)
    while $v$ is non-terminal do
        if $v$ not fully expanded then
            return Expand($v$)
        else
            $v \leftarrow$ BestChild($v_0, Cp$)
        end if
    end while
    return $v$
end function

function Expand($v$)
    $a \leftarrow$ any random unused action in $A(s(v))$
    add new child $v'$ to $v$ where:
    $s(v') = \text{Result}(s(v), a)$
    $a = a(v')$
    return $v'$
end function

function BestChild($v, c$)
    return $\underset{v' \in children(v)}{\text{argmax}} \frac{Q(v'))}{N(v')} + c\sqrt{\frac{\ln N(v)}{N(v')}}$
end function

function DefaultPolicy($s$)
    while $s$ is non-terminal do
        choose $a \in A(s)$ uniformly at random
        $s \leftarrow$ Result($s, a$)
        return reward for state $a$
    end while
end function

function Backup($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of $v$
    end while
end function
```

**All Moves As First (AMAF)**

Originally the AMAF method has been introduced to improve the learning process inside MCTS trees. The selection of the UCT algorithm is based on an estimated value obtained by simulating the move in the node a couple times. This can lead to a problem if there is a large state space since the algorithm has to do many simulations before it can sample all the moves in a node. To conquer this issue AMAF, also known as RAVE, is introduced. For every node the algorithm stores for all legal moves the following values

- The average UCT result obtained from all simulations in which move $a$ is performed in state $s$
- The average AMAF result, obtained from all the simulations in which move $a$ is performed further down the path that passes by node $s$

When backpropagating the result of a simulation in a certain node $t$ in the tree, the UCT result is updated for the move $a$ that was directly played in the state, and the AMAF value is updated for all the legal moves in node $t$ that have been encountered at a later stage in the simulations. AMAF will encounter more samples and will use them to make better predictions. However, the disadvantage of this information is that AMAF information is more global whilst the UCT information is more local. This makes AMAF scores useful for less visited nodes, but when the number of visits increases, the UCT values should become more important. [14]

To solve this issue the AMAF algorithm keeps track of the two scores separately and uses a weight $\beta$ to reduce the importance of the AMAF score over time.

$$\beta = \frac{\tilde{n}}{n + \tilde{n} + 4n\tilde{n}\tilde{b}^2} [9]$$

To calculate the $\beta$ value AMAF uses the equation denoted above. Where $\tilde{n}$ is the visits of AMAF and $n$ is total number of visits or amount of simulations. The equation also includes a bias $\tilde{b}$ which is in this algorithm a parameter which can be adjusted by testing the performance. [9]

To understand AMAF there is pseudocode above in algorithm 4. In comparison with normal back propagation it also updates the AMAF values for certain actions.

**Parallelization**

There are three different types of parallelisation, depending on in what stage Monto-Carlo Tree Search is parallelised. The three options are: leaf parallelization, root parallelization and tree parallelization.

In this paper only leaf parallelization was considered since this is a straightforward to implement. To select a leaf node this method uses one main thread, which is

**Algorithm 4** All Moves As First (AMAF)

```
function BACKPROPAGATE(node, reward, times)
    innerBackpropagate(node, reward, times)
    for every child of the parent of node do
        amafForwardPropagation(child, reward, times, action)
    end for
end function

function INNERBACKPROPAGATE(node, reward, times)
    $visits \leftarrow visits + 1$
    $reward \leftarrow reward + 1$
    if $node \neq root$ then
        innerBackpropagate(parent, reward, times)
    end if
end function

function AMAFFOWARDPROPAGATION(node, reward, times, action)
    if $nodeaction = action$ then
        $visits \leftarrow visits + 1$
        $reward \leftarrow reward + 1$
    else
        for every child of the parent of the node do
            amafForwardPropagation(child, reward, times, action)
        end for
    end if
end function
```

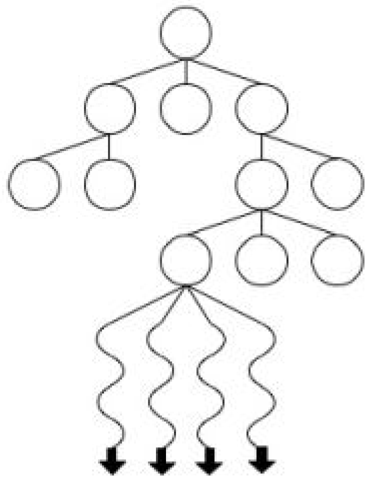
Figure 4: Visual explanation of leaf paralellization, the down arrows are the threads [7]

the same thread that runs the game logic. From this leaf node it simulates independent games for each available thread. When all games are finished the main thread backpropagates all the values.[7] Leaf parallelization is depicted in figure 4.

A problem with leaf parallelization is that threads have to wait for each other which could slow then the process a way to solve this is end simulations for a move if a certain amount of games is won. For instance, if 16 threads are available, and 8 (faster) finished games are all losses, it will be highly probable that most games will lead to a loss. Therefore, playing 8 more games is a waste of computational power. This will enable the program to traverse the tree more often.[7]

To reduce the cost of using threads, both the *Java ExecutorService* interface and *ThreadGroups* were implemented and compared.

### Tree reuse

The ordinary implementation of Monte-Carlo Tree Search builds a new tree for every move a player has to make. An obvious improvement to speed up the algorithm would be that the program does not build a tree every move but it tries to keep the tree it made in the previous move by searching through the children of the previous move and check if the node already exist. Since the program does not know the move of the opponent it could be that the move does not exist This strategy enables the algorithm to have already some reward values for certain moves so the predictions can be done more accurately.

## 4 Experiments

The algorithms described in the previous section were tested against each other in a Hex game engine with the algorithmic frameworks. Different experiments were conducted on separate machines. The machines used to conduct the experiments are listed below:

- Machine 1, Intel i5 2600 2.4GHz, 8GB ram
- Machine 2, Intel i7 4720HQ 2.6GHz, 8GB ram
- Machine 3, Intel i5 5257U 2.7GHz, 8GB ram
- Machine 4, Intel i5 3210M 2.5GHz, 6GB ram

When two algorithms were tested against each other they played 20 games against each other. After 10 games, the algorithm allowed to play first was reversed.

### 4.1 Experiment 1 Alpha-beta comparison

In the first experiment, the two evaluation functions for the Alpha-Beta bot were compared: Dijkstra's shortest path algorithm vs. Electrical circuit analysis. A random element was added to the first move of the bots, so every tested game would be different. The chosen depth for both bots was 3. The test was run on machine 4.

### 4.2 Experiment 2 Finding the best MCTS-UCT player

The UCT algorithm has two main parameters that can be tested in order to get the best possible player; these are the exploration parameter and the number of simulations per iteration in every visited node.

Time was not used as the computation budget because games were played on machine 1, 2 and 3 to safe time.

**Exploration parameter $C_p$**

The exploration parameters that were tested were $C_p$ ={ 0.0, 0.2, 0.4, 0.6, 0.8, 1, 3, 5}. The test consisted in getting the best exploration parameter from each subset $X$ = {0.0, 0.2, 0.4} , $Y$ ={0.6, 0.8, 1}, and $Z$ ={3, 5} . Once the best exploration parameters from each subset were found, they were played against each other to find the best one of all the tested exploration parameters. Each individual $C_p$ was tested in with SPI = 1 and SPI =10 in order to find the best $C_p$.

**Simulations per Iteration (SPI)**

As the best $C_p$ value was found, the next step is to find the best SPI. The SPIs that were tested are $SPI$ = {10, 50, 100, 500, 1000, 10000}. They were tested against a player using $spi = 1$.

### 4.3 Experiment 3 leaf-parallelization

For the third experiment, the use of leaf parallelization in MCTS was tested to see if this yielded any performance or efficiently improvements. The mutltithreading was tested on 2, 4 and 8 cores with SPI values of 10, 100 and 1000. The opponent was normal MCTS-UCT without $C_P = 0.6$ and SPI = 1. The tests were conducted on machine 3.

### 4.4 Experiment 4 Multi-thread techniques

Leaf-parallelization makes use of Threads objects in the *Java* Standard Library. To implement multi-threading the standard library provides different ways to manage multiple threads. This experiment tests whether the new *ExecutorService* interface for multi-threading performs better than using *Threadgroup*. MCTS-UCT with $C_p$ = 0.6, SPI = 0.6 and leaf-parallelization using Threadgroup with 4 cores was played against the same MCTS-UCT with leaf-parallelization using *ExecutorService*

### 4.5 Experiment 5 AMAF bias $b$

AMAF relies on a bias parameter $b$. Different biases were tested against the best UCT player to see how it affects the performance of the AMAF player. The biases tested were {0, 0.25, 0.5, 0.75, 1, 1.5, 2}.

### 4.6 Experiment 6 Alpha-beta vs MCTS

Using the previous results the best Alpha-Beta bot was compared with the best MCTS bot. 20 games were played with each bot being the first to move 10 times with both players having 10 seconds per move. This makes the results comparable since as mentioned previously the first player has a distinct advantage It was expected that MCTS outperforms Alpha-Beta search because currently the strongest Hex players are using MCTS and the MCTS implementation used is closer to the best MCTS bots that the Alpha-Beta bot is to the best Alpha-Beta bots used.

## 5 Results

### 5.1 Experiment 1

- Electrical circuit as Player 1: won 10 out of 10 games.
- Electrical circuit as Player 2: won 9 out of 10 games.

The results are as expected, since the Dijkstra Evaluation only considers one path, and the Electrical Evaluation considers all possible paths. The only time the Dijkstra bot won, it got a good opening move, while the Electrical bot got a bad opening move.

Table 1: Experiment of subset $X$

| $C_p$ | 1 SPI | 10 SPI |
|---|---|---|
| 0 | 2.5 | 10 |
| 0.2 | 50 | 57.5 |
| 0.4 | 97.5 | 82.5 |

Table 2: Experiment of subset $Y$

| $C_p$ | 1 SPI | 10 SPI |
|---|---|---|
| 0.6 | 55 | 52.5 |
| 0.8 | 45 | 45 |
| 1 | 50 | 52.5 |

Table 3: Experiment of subset $Z$

| $C_p$ | 1 SPI | 10 SPI |
|---|---|---|
| 3 | 70 | 45 |
| 5 | 30 | 65 |

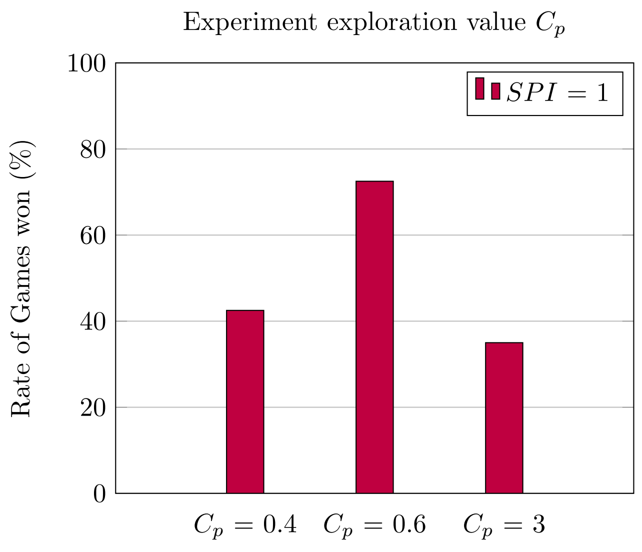

Figure 5: Experiment exploration value $C_p$, comparison between different $C_p$

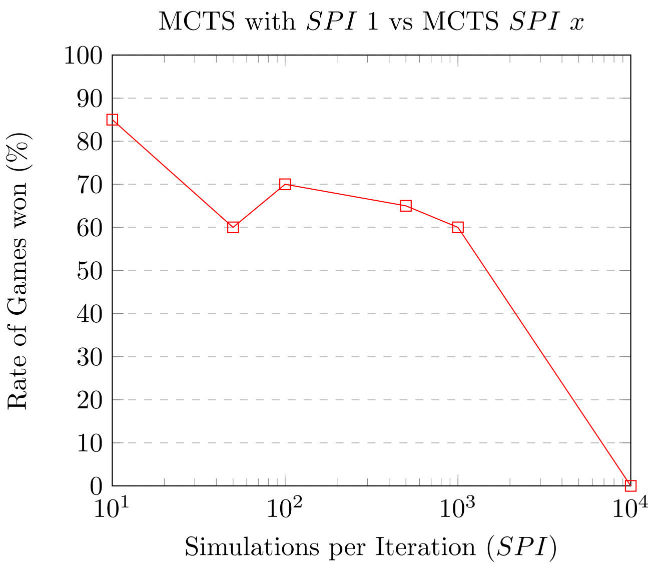

Figure 6: Experiment finding the best SPI

Table 4: Results of leaf-parallelization with different parameters

| core \ spi | 10 | 100 | 1000 |
|---|---|---|---|
| 2 | 65 | 75 | 60 |
| 4 | 70 | 55 | 70 |
| 8 | 55 | 55 | 65 |

## 5.2 Experiment 2

### Exploration parameter $C_p$

As seen in the figure 5 and Table 2 the best exploration parameter is $C_p = 0.6$ . Although as shown in Table 1, $C_p = 0.4$ out performed in the subset $X$ and $C_p = 3.0$ barely won in subset $Z$(Table 3), when tested against 0.6 both were outperformed. After the second part of this experiment took place, the exploration parameter of $C_p = 0.6$ is considered the best.

### Simulations per Iteration (SPI)

As can be seen in figure 6, as the SPI increases its performance decreases. However, if the SPI of the player $1 \leq spi \leq 1000$ it outperforms an SPI of 1. Therefore, the best SPI-value would be 10.

## 5.3 Experiment 3

The table 4 shows that every parameter has a higher win-rate against MCTS-UCT without leaf-parallelization. It is also important to note that using 8 cores is less effective than using 2 or 4 cores. The amount of SPI used does not seem to be very important for gaining better results

Table 5: My caption

| | ThreadPool | ExecutorService |
|---|---|---|
| win % | 9 | 11 |

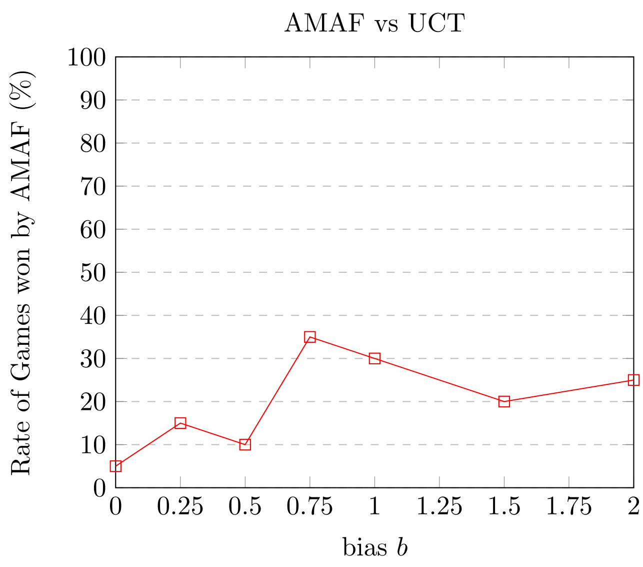

Figure 7: Experiment with bias $b$ of AMAF to find the optimal $b$ value

### 5.4 Experiment 4

Table 5 shows that *ExecutorService* won 1 more game than *ThreadPool*. This is a 10 % increase in winning rate. It would be interesting to do more experiments on this because 20 games is a small sample size.

### 5.5 Experiment 5

The result of the test in the figure 7 show that although a UCT policy outperforms all AMAF players adding a bias affects the AMAF player's performance. The best bias would be $b = .75$ because as the bias gets bigger than 0.75 the player's performance is affected negatively.

### 5.6 Experiment 6

The results of the final test shows clearly that MCTS is the superior algorithm, with a 100% win rate against alphabeta electrical circuit. This remains the same even when playing as the second player. Results can be seen in Table 6

Table 6: MCTS vs Alphabeta

| | MCTS | Alphabeta |
|---|---|---|
| Won(%) | 100 | 0 |

## 6 Conclusion

After having conducted all the experiments above, three main conclusions can be drawn. All conclusions apply to our implementation of the game and the algorithms. Firstly, the Alpha-Beta algorithm was tested with two different evaluation functions: Dijkstra's algorithm and an electrical ciruit simulator. As one can see in the results,the electrical circuit is the best evaluation function out of the two. This is due to the fact that the electrical circuit simulation considers the whole board while the other evalution only considers a single path. Secondly, for the MCTS algorithm, the best UCT and AMAF policies were found, and tested against each other. The best UCT policy, with a $C_p = 0.6$ and an $spi = 10$, out performed the best AMAF policy with a $c = 0$ and bias $b = .75$. Lastly, the best Alpha-Beta and MCTS algorithms were tested against each other. The best MCTS algorithm had complete dominance over the best Alpha-Beta algorithm. It is believed that under the used constrains exploring a partial game tree is a stronger strategy than to explore the whole game tree.

## 7 Future Research

Overall, this article answers several interesting questions about Hex algorithms. However, there are still quite a few research questions that could be addressed in the future. For instance, it would be very interesting to see how H-search affects the performance of Alpha-Beta search. To what extend would H-search improve the overall performance and efficiency of Alpha-Beta search? Furthermore, it would be compelling to see how all used algorithms compare using different time constraints to achieve more comparable results. Does Alpha-Beta perform better than MCTS with more time or not? In addition, the MCTS search could be further edited with the evaluation functions used for Alpha-Beta search. One could compare how this new MCTS performs against MCTS used in this article.

Other intriguing techniques are Q-learning and Artificial Neural Networks (ANN). For ANNs it would be refreshing to see how they compare to the approaches used in this report. Similar research has been conducted previously in 2007[10] and recently in 2016[15]. The later research suggested that their network is likely to improve in performance with more training time.[15]

## References

[1] Anshelevich, Vadim V (2002). A hierarchical approach to computer hex. *Artificial Intelligence*, Vol. 134, No. 1-2, pp. 101–120.

[2] Arneson, Broderick, Hayward, Ryan B, and Henderson, Philip (2010). ”monte carlo tree search in hex”. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 251–258.

[3] Auer, Peter (2002). Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, Vol. 3, No. Nov, pp. 397–422.

[4] Browne, Cameron B, Powley, Edward, Whitehouse, Daniel, Lucas, Simon M, Cowling, Peter I, Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). ”a survey of monte carlo tree search methods”. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–43.

[5] Chalup, Stephan K, Mellor, Drew, and Rosamond, Fran (2005). The machine intelligence hex project. *Computer Science Education*, Vol. 15, No. 4, pp. 245–273.

[6] Chaslot, Guillaume M JB, Winands, Mark HM, HERIK, H JAAP VAN DEN, Uiterwijk, Jos WHM, and Bouzy, Bruno (2008a). Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, Vol. 4, No. 03, pp. 343–357.

[7] Chaslot, Guillaume MJ-B, Winands, Mark HM, and Den Herik, H Jaap van (2008b). ”parallel monte-carlo tree search”. *International Conference on Computers and Games*, pp. 60–71, Springer.

[8] Gardener, Martin (1959). ”hexaflexagons and other mathematical diversions - the first scientific american book of puzzles and games”. pp. 73–75.

[9] Gelly, Sylvain and Silver, David (2011). ”monte-carlo tree search and rapid action value estimation in computer go”. *Artificial Intelligence*, Vol. 175, No. 11, pp. 1856–1875.

[10] Kahl, Kenneth, Edelkamp, Stefan, and Hildebrand, Lars (2007). Learning how to play hex. *Annual Conference on Artificial Intelligence*, pp. 382–396, Springer.

[11] Nash, John (1952). ”some games and machines for playing them”.

[12] Russell, Stuart, Norvig, Peter, and Intelligence, Artificial (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, Vol. 25, p. 27.

[13] Shannon, Claude E (1953). Computers and automata. *Proceedings of the IRE*, Vol. 41, No. 10, pp. 1234–1241.

[14] Sironi, Chiara F and Winands, Mark HMComparison of rapid action value estimation variants for general game playing.

[15] Young, Kenny, Hayward, Ryan, and Vasan, Gautham (2016). Neurohex: A deep q-learning hex agent. *arXiv preprint arXiv:1604.07097.*