AI Implementations for Hex

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

January 23, 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 Hex players, as well as comparasions of different evaluation functions which includes Dijkstra, Electric Circuit. In addition to this Monte Carlo Tree Search specific heuristics are compared.

GIVE RESULTS!!!!

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. The generally accepted normal size of the board is 11 × 11, however it can be played on boards of differing sizes. One player will try to create a bridge from the top of the board to the bottom, while the other player tries to make one from left to right. The game is fully deterministic, as by finishing one players bridge will always block the other player. As the first player is always at an advantage, Hex has a unique rule to address this. Called the Pie Rule, it states that after the second player has had their first move, they may switch the placement of the first two tiles.

The history of creating algorithms to play Hex is rich with successful attempts, all using a different approach. In this paper, several of these attempts will be implemented and tested against both each other, while also seeing if limiting factors in these algorithms (time per move, tree depth, etc.) have major effects on the results.

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. Since the first player is always at distinct advantage there exists the Pie Rule, which states that after the second player has made their first move they may swap the positions of the first two tiles. This rule is not implemented in the algorithms described in this paper, as both algorithms will have equal time as player one and player two.

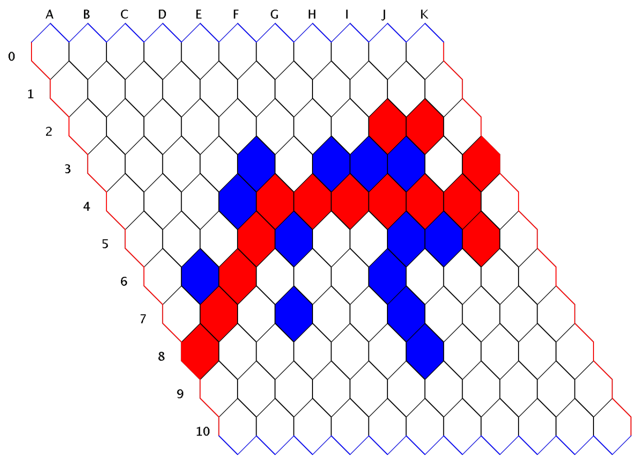

Figure 1: An example of a hexboard

BACKGROUND JOHN NASH STEALING ARGUMENT SWAP RULE [10]

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 α - β 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 [12] 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. [11] 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. [11]

α - β 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." [11]. 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

```
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
```

3.3 Evaluation Functions

Shannon and Moore [12] model the board as an electrical circuit which overcomes this limitation as it considers all paths in parallel. 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:

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

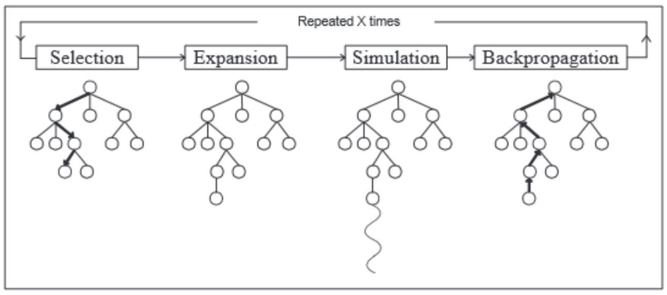

Figure 2: The four steps of Monte-Carlo Tree Search [6]

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.

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

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 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 2):

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

```
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.

3.5 Improvements on General MCTS

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

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:

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

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 ∞.

XXXX NEEDS TEXT ABOUT ALGORITHMS

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 a 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. [13]

To solve this issue the AMAF algorithm keeps track of the two scores separately and uses a weight β 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 β value AMAF uses the equation denoted above. Where $\tilde{n}$ is the visits of AMAF and n

Algorithm 3 Upper Confidence Threshold

```
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
        $s \leftarrow \text{Result}(s, a)$
        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
```

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]

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
```

In algorithm 4 **XXXX NEEDS TEXT ABOUT ALGORITHMS**

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 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 3.

A problem with leaf parallelization is that threads have to wait for each other which could slow then the

Figure 3: Visual explanation of leaf paralellization, the down arrows are the threads [7]

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

All experiments were run on the following machines:

- 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

Every test was run 20 times, with each algorithm being player one 10 times. For the first experiment, simulations per iteration (SPI) and the exploration coefficient (C) were changed to find the best combination of the two. These results will be used when testing the MCTS evaluation function comparison and MCTS parallelization tests.

In the second test, the different evaluation functions for MCTS were compared to see which one is superior. This will be used when MCTS is compared against the Alpha-Beta algorithm.

For the third experiment, the use of leaf parallelization in MCTS was tested to see if this yielded any performance or efficiently improvements. If it does, this will also be included when MCTS is compared against Alpha-Beta. The fourth experiment compared the evaluation functions for the Alpha-Beta algorithm. Whichever was more successful will be used when comparing against MCTS. (Machine 4) Finally, the last experiment was to do a direct comparison of Alpha-Beta and MCTS with their most optimal settings to determine which algorithm is the best.

Every test was run 20 times, with each algorithm being player one 10 times.

XXXX Some settings stuff, fun times

5 Results

Explanation of results of the experiments

6 Conclusion

Conclusion on the explanation of the result of the experiments

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] Nash, John (1952). "some games and machines for playing them".

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

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

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