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.

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 release by the firm Parker Brothers, Inc [?]. 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 $\times$ 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 very 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 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 exsists 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 impliemented in the algorithms descriped in this paper, as both algorithms will have equal time as player one and player two.

3 Algorithms

3.1 Alpha-beta Hex Players

The alpha-beta player consists of a player which uses a minimax game tree and applies $\alpha - \beta$ pruning as it builds the tree.

Minimax algorithm

The minimax algorithm tries to find the best next possible action by fully expanding each node and seeing all possible moves in a certain depth. After the tree is fully expanded the leaf nodes are evaluated and given a value.

3.2 Electric circuit

Explanation of the electric circuit of Anshelivic

3.3 Monte Carlo Tree Search

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

The game tree 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 $a \in A$.

- 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

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 1):

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

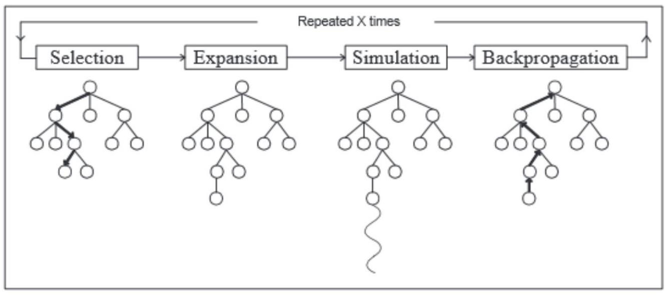

Figure 1: The four steps of Monte-Carlo Tree Search [?]

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 1 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 1 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.

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

$$UCT = \bar{X_v} + C_p\sqrt{\frac{\ln n}{n_v}} \tag{1}$$

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 $C_p >= 0$ can be used to weight the importance of exploration. If a child is not visited ($n_v = 0$) the exploration term is set to ∞.

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. [?]

Algorithm 2 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
        $s \leftarrow$ 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
```

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.

$$\beta = \frac{\tilde{n}}{n + \tilde{n} + 4n\tilde{n}\tilde{b}^2} [?] \qquad (2)$$

To calculate the β value AMAF uses the equation denoted above in (2). 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. [?]

Algorithm 3 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 3

Parallelisation

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

In this paper we only consider leaf parallelisation since this is a straightforward to implement. To select

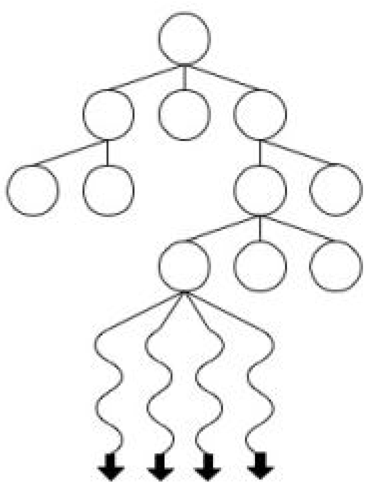

Figure 2: Visual explanation of leaf parallelisation, the down arrows are the threads [?]

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.[?] Leaf parallelisation is depicted in figure 2.

A problem with leaf parallelisation 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.[?]

To reduce the cost of using threads we implemented both the *Java ExecutorService* interface and *Thread-Groups* and we compared them

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

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