**HEX GAME AI**

**MD Asif Hasan**

Date: 03/6/2017

**Alpha Beta**

I employed standard Alpha Beta algorithm with a limit on search depth. At each node, it branches out to all possible moves selecting the next move randomly. When the search depth is reached, the position of that board is scored by an evaluation function. The evaluated score is then backed up to the top according to the rules of alpha beta. The algorithm terminates either when it fully has explored all the nodes bounded by a depth budget or the running time of the whole algorithm reached the time budget. As the last step of termination it selects the next move by comparing the scores of its childs and picking the best one. In the rest of this section I will provide detailed discussion on some of the important aspects of the implementation of Alpha Beta algorithm for Hex.

**Evaluation Function and Search Depth:**

To evaluate a position of a board, a particular heuristic is used which is the maximum number of layers in the board connected for a player. Here layer means the rows or cols in the 8x8 grid. For one player the number of rows are used and for the other the number of columns are used depending on which sides of the grid are they trying to connect. At first this score is calculate for the MAX player and then it is calculated for the MIN player. Then the difference of the two scores is used as the final score for the current position. The whole equation can be expressed as following:

H(Player) = Maximum number of layers connected by the moves of the Player.

Evaluation(board\_state) = H(MAX) - H(MIN)

As per the experience I have got from trying different evaluation function from the very basic to a more informed, it appears that for Alpha Beta algorithm, the evaluation function is of major importance. The kind of evaluation function is used dictates what the AI player would play for. Basically AI Player does not try to win the game as it does not know how to do that, it tries to impress the outcome of the evaluation function by looking up in the future as in Minimax fashion. AI player grows its movements to get good result for when it is evaluated. For example, I have tried with a function which score better if the total nodes in a bridge is more rather than the number of layers the bridge connects. Using that function resulted the AI player to move in a way to make the length of the bridge bigger even though the winning move was at one move next.

The current evaluation function that I am using is not certainly perfect but performs well. Playing against human it does not guarantee to win even if it wins the toss and plays the first move. But it does achieve one thing which is, if the human player makes some mistakes, the chances of winning for the AI player increases very much. Often it wins even if he human player makes no or just one mistake than playing humanely optimal.

To improve on the current evaluation function, I can think of few directions which can be tried out. One is that, often it is seen that when a build-up bridge is blocked by the opponent, the AI player continues to improve upon that bridge even though it can be seen that the number of moves that would be necessary from that bridge to reach the end goals is too much. Due to limit in depth the Minimax can not sense that and also since the evaluation function rewards better for making even a progress on connecting the next layer, it remains happy to increase the length of the chain to connect a next layer in vain. The solution out of this would be, along with the current evaluation function using more knowledge, specifically to calculate how many moves at the minimum would be necessary on the current state of the board to reach the goal layer from that bridge. In that way, it would try to improve another bridge with shorter number of connecting layers which would need less moves necessary to reach the goal. This heuristic has its own problem but certainly improves upon the current. Due to lack of time I have not been able to implement and experiment with it.

Another Important aspect of the evaluation function is the computational efficiency. As in Minimax most of the time is spent for evaluating the leaf nodes, the faster the evaluation function more nodes it can evaluate. I have spent significant amount of time to make the evaluation function efficient. My first implementation of the evaluation function resulted that, the whole algorithm was taking up 90% of its running time to evaluate the leaf nodes. Then I made it efficient by decreasing the array lookups by saving some information(e.g. If a node is expanded) within the nodes. I got it to reduced to 65%-70% of the total algorithm running time. I was able to run AlphaBeta up to depth of 3 with a time limit of 120 seconds. And then I changed the implementation of the evaluation function to use Union-Find algorithm which reduced the time spent on evaluating to 20-30%, which gave the whole algorithm a boost to exhaustively search all the nodes of depth 4. The stats shows that, AlphaBeta with depth limit 4 wins 90% of matches against the one which uses depth limit of 3. Even using a depth limit of 5 with time budget of 120 seconds, the algorithm runs through over 1.4 million nodes searching all the nodes of depth 5 within 100 seconds taking roughly around 50 seconds on average.

Though due to time taken for each move I have not tested many matches with depth 5 as the limit. But according to the few matches I have tried with, it was very much interesting. It seems with the current settings (e.g. current knowledge of the evaluation function), the AlphaBeta algorithm searching exhaustively all the nodes within depth 4 is not performing much worse than which searches all the nodes within depth 5. Here is a stat for their close encounter:

Stat:

Depth 4 vs Depth 5: win ratio = 1 : 2

totalPlay: 3, tossWon[0]: 1, tossWon[1]: 2, Won[0]: 1, Won[1]: 2, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 1, toss\_won\_then\_won[1]: 1, toss\_lost\_then\_won[1]: 1

One thing to note that, with the depth 5 limit, the algorithm actually reached the time budget of 120 seconds. To understand how they face each other when they are at their best, I increased the time budget for depth 5 to 300 seconds. But still depth 4 fought well becoming victorious in some of the matches. It shows the power of Alpha Beta and Evaluation function. With better evaluation function, necessity of searching through the search space exhaustively can be mitigated by a great amount.

**Game Termination:**

Since I have developed a very fast (linear to number of cells occupied) algorithm to evaluate a position to find out how many layers are connected by a player, I have used the same evaluation function to determine if a game is ended. So, when 8 layers are connected, the game is over. I did not have time to invest more on deciding the dead ends by computing as I was more into improving the evaluation function and calculation of the game over as I had found they are taking most of the time of the algorithm running time. But, it is understandable that, having such logic implemented efficiently (e.g. linear to number of cells occupied) can further prune many branches.

**Memory Usage:**

These stats could vary according to the game played. But here is a stat from some games to give a idea what each variations consumes of memory.

Depth Limit 3:

Max total nodes evaluated: 25,251

Max Tree Size: 41,803

Depth Limit 4:

Max total nodes evaluated: 349,873

Max Tree Size: 771,844

Depth Limit 5:

Max total nodes evaluated: 1,860,361

Max Tree Size: 12,011,269

**MCTS**

I employed standard MCTS algorithm with a limit on running time. In this section I will provide detailed discussion on some of the important aspects of the implementation of the MCTS algorithm for Hex.

**Expansion policy:**

At each iteration, the best child with the highest UCT score with the value of constant C which can be set as a parameter to the algorithm is used. Checking the performance stats for different C values I finally have fixed to C = 0.707. At Below, some of the stats are given. At each node the expansion followed by creating one child node with one of the available moves. So, at each node every possible moves could be its child. At the root, if the player is the first player and the board is empty, all 64 cells could be its child. On second layer, they could have all 63 possible cells and so on.

Stats with different C:

C = 1.44 vs C = 0.707 with time limit of 10 seconds:

Win Ratio: 0 **:** 6

Details stat:

totalPlay: 6, tossWon[0]: 2, tossWon[1]: 4, Won[0]: 0, Won[1]: 6, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 4, toss\_lost\_then\_won[1]: 2

C = 1 vs C = 0.707 with time limit of 10 seconds:

Win Ratio: 0 **:** 3

Details stat:

totalPlay: 3, tossWon[0]: 3, tossWon[1]: 0, Won[0]: 0, Won[1]: 3, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 0, toss\_lost\_then\_won[1]: 3

C = 0.5 vs C = 0.707 with time limit of 10 seconds:

Win Ratio: 0 **:** 3

Details stat:

totalPlay: 3, tossWon[0]: 1, tossWon[1]: 2, Won[0]: 0, Won[1]: 3, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 2, toss\_lost\_then\_won[1]: 1

**Simulation Method:**

For the simulation from one node, I used truly random simulation of a play from that position. After running the fixed number of simulations, finally the score for that node is calculated using the following equation:

Score = (number\_of\_win\_player - number\_of\_win\_opponent) / total\_simulation

And if the node being simulated is a terminal node then,

If node is a victory node for the player:

Score = 1

Else,

Score = -1

I have kept the number of simulations as a parameter to the program and compared the performance for MCTS with different number of simulations with each other. Below are the stats for that:

* 2 vs 5: win ratio 0:11 with time limit of 3 secs
* 10 vs 5: win ratio 0:4 with time limit of 20 secs
* 3 vs 5: win ratio 0:5 with time limit of 10 secs

From the stats, it was apparent that, simulating 5 random games were winning over the other tried ones. For 10 vs 5 games, the argument may be when 5’s MCTS was iterating on around 12000 nodes at max depth of 6 per move 10’s MCTS was iterating on around 6000 nodes at max depth of 5. The same argument was not standing when it was played between 3’s MCTS and 5’s MCTS, where 3’s MCTS were running through almost double amount of nodes. But in this case the number of simulations may have been too little for 3’s MCTS to evaluate a node.

Detailed Stats:

Stat 1:

* Player 1:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 3 secs
    - **Simulation Count: 2**
    - BestMove Criterion: Max Score
    - C: 1.44
* Player 2:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 3 secs
    - **Simulation Count: 5**
    - BestMove Criterion: Max Score
    - C: 1.44
* Stats:
  + totalPlay: 11, tossWon[0]: 9, tossWon[1]: 2, **Won[0]: 0, Won[1]: 11**, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 2, toss\_lost\_then\_won[1]: 9

Stats 2:

* Player 1:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 20 secs
    - **Simulation Count: 10**
    - BestMove Criterion: Max Score
    - C: 1.44
* Player 2:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 20 secs
    - **Simulation Count: 5**
    - BestMove Criterion: Max Score
    - C: 1.44
* Stats:
  + totalPlay: 4, tossWon[0]: 2, tossWon[1]: 2, **Won[0]: 0, Won[1]: 4**, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 2, toss\_lost\_then\_won[1]: 2
* Notes:
  + When Player 2 was iterating on around 12000 nodes at max depth of 6 per move Player 1 was iterating on around 6000 nodes at max depth of 5

Stats 3:

* Player 1:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 10 secs
    - Simulation Count: 3
    - BestMove Criterion: Max Score
    - C: 1.44
* Player 2:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 10 secs
    - Simulation Count: 5
    - BestMove Criterion: Max Score
    - C: 1.44
* Stats:
  + totalPlay: 5, tossWon[0]: 4, tossWon[1]: 1, Won[0]: 0, Won[1]: 5, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 1, toss\_lost\_then\_won[1]: 4

**Final Move Selection & Simulation Score:**

Different Methods:

1. Difference of win ratio:
   * Max of (cumulative Score of all visits / total visits), where Score = (number\_of\_win\_player - number\_of\_win\_opponent) / total\_simulation
2. Max Visits:
   * Node with maximum number of visits
3. Highest Win:
   * Max of cumulative Score of all visits, where Score =number\_of\_win\_player
4. Highest Win Rate:
   * Max of (cumulative Score of all visits / total visits), where Score =number\_of\_win\_player
5. highest win rate and highest number of simulations associated with the move
   * Max of (cumulative Score of all visits / total visits), where Score =number\_of\_win\_player / total\_simulation

Stats: Each with a time limit of 3 seconds

(1) vs (2): Win ratio = 8 : 0

totalPlay: 8, tossWon[0]: 4, tossWon[1]: 4, Won[0]: 8, Won[1]: 0, toss\_won\_then\_won[0]: 4, toss\_lost\_then\_won[0]: 4, toss\_won\_then\_won[1]: 0, toss\_lost\_then\_won[1]: 0

(1) vs (3): Win ratio = 4 : 0

totalPlay: 4, tossWon[0]: 1, tossWon[1]: 3, Won[0]: 4, Won[1]: 0, toss\_won\_then\_won[0]: 1, toss\_lost\_then\_won[0]: 3, toss\_won\_then\_won[1]: 0, toss\_lost\_then\_won[1]: 0

* Note: The result can be explained by the fact that, just using the total number of win for the score of evaluation from the simulation, resulted the algorithm to try very few nodes which can be seen from the stats given below (from one game) which made the algorithm to perform worse.

(1) Max Tree Size: 151

(3) Max Tree Size: 1747

(1) vs (4): Win ratio = 4 : 0

totalPlay: 4, tossWon[0]: 3, tossWon[1]: 1, Won[0]: 4, Won[1]: 0, toss\_won\_then\_won[0]: 3, toss\_lost\_then\_won[0]: 1, toss\_won\_then\_won[1]: 0, toss\_lost\_then\_won[1]: 0

* Note: Same as above the algorithm with (4) as the selection method throughout resulted into expanding very few nodes

(1) Max Tree Size: 123

(4) Max Tree Size: 2154

(1) vs (5): Win ratio = 4 : 0

totalPlay: 4, tossWon[0]: 1, tossWon[1]: 3, Won[0]: 4, Won[1]: 0, toss\_won\_then\_won[0]: 1, toss\_lost\_then\_won[0]: 3, toss\_won\_then\_won[1]: 0, toss\_lost\_then\_won[1]: 0

* Note: (5) expanded more than (3) or (4) and looked to be performing better, but still not enough.

(1) Max Tree Size: 658

(4) Max Tree Size: 2077

After comparing different methods, I have decided to use the method (1).

**Memory Usage:**

This stat can vary according to the game played. But here are few stats for the maximum size of the tree during a whole game play:

Time Limit: 5

Max Tree Size: 4315

Time Limit: 10

Max Tree Size: 7776

Time Limit: 20

Max Tree Size: 12624

**Game Termination:**

The termination of a game is detected using the same evaluation function used for AlphaBeta as that is fast and runs in linear time to number of cells occupied. Furthermore, for a node, once computed that if it is a terminal node or not, that information is recorded in order to save recomputation.

**NegaMax:**

For backing up the scores from the simulating node to the top, NegaMax backup function was used. The root of the tree needed a special treatment. When creating the child of the root, essentially, the player moves and opponents moves were kept to be same and from then on, to the deeper child, these data structures were alternated.

**Game Stats:**

Below I have put together some stats of games played in various combinations. To my surprise, the latest version of both of the AI games played well with me. I was beaten most of the times.

**AlphaBeta vs MCTS:**

* Player 1:
  + Algorithm: MCTS with Sim:Evaluation
  + Params:
    - Time limit: 3 secs
    - Simulation: 5
* Player 2:
  + Algorithm: Alpha Beta
  + Params:
    - Depth limit: 4
    - Time limit: 120 secs
* Stats: Win ratio = 4 : 13
  + totalPlay: 17, tossWon[0]: 9, tossWon[1]: 8, **Won[0]: 4, Won[1]: 13**, toss\_won\_then\_won[0]: 4, toss\_lost\_then\_won[0]: 0, toss\_won\_then\_won[1]: 8, toss\_lost\_then\_won[1]: 5
* Note:

MCTS Appears to be dominating AlphaBeta in the games played. Though I have kept the maximum running time of MCTS to be 5 seconds, yet it got success over the Alpha Beta most of the times.

**AlphaBeta vs Human:**

* Player 1:
  + Algorithm: Human
* Player 2:
  + Algorithm: Alpha Beta
  + Params:
    - Depth limit: 4
    - Time limit: 120 secs
* Stats: Win ratio = 1 : 4
  + Player Human - Player AI 2 : totalPlay: 5, tossWon[0]: 1, tossWon[1]: 4, Won[0]: 1, Won[1]: 4, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 1, toss\_won\_then\_won[1]: 3, toss\_lost\_then\_won[1]: 1
* Note:

Apparently my AI has beaten me 4 times out of 5. May be it was that good. I played strongly. But it has beaten me.

**MCTS vs Human:**

* Player 1:
  + Algorithm: Human
* Player 2:
  + Algorithm: MCTS
  + Params:
    - Simulation: 5
    - Time limit: 5 secs
* Stats: Win ratio = 2 : 3
  + totalPlay: 5, tossWon[0]: 1, tossWon[1]: 4, Won[0]: 2, Won[1]: 3, toss\_won\_then\_won[0]: 0, toss\_lost\_then\_won[0]: 2, toss\_won\_then\_won[1]: 2, toss\_lost\_then\_won[1]: 1
* Note:

I felt this configuration was easier than the Alpha Beta, but yet, it has beaten me consecutively in the last 3 games.