Reinforcement Learning 2021
Practical Assignment 1: Search

Xinrui Shan(s2395738) Pengyu Liu(s2604531) Zhe Deng(s2696266)

1 Introduction

Hex is a connection strategy game for two players, and players attempt to connect opposite sides of a hexagonal board either horizontally (blue), or vertically (red). The first player who succeeds in doing so, wins. In this assignment, we aim to build an AI that determines the placement position to help play the game, and we implemented it with two algorithms, **Alpha Beta, Monte-Carlo Tree Search(MCTS)**.

1.1 Tree structure

Since we are working on implement search algorithms, we construct the tree structure to store information with the following attributes: (i)board with virtual movements of two players, (ii)parent node, (iii)children, (iv)a list of all possible untried move, i.e. empty lattice, (v)visited, (vi)win, (vii)position, (viii)color, (ix)g value.

2 Alpha-Beta

Alpha-beta is an adversarial search algorithm as for its name alpha vs. beta, which usually used for machine play in two-person games (tic-tac-toe, chess, go, etc.). It is developed from the *minimax* algorithm with *cutoff*, which reduces the number of nodes evaluated in its search tree. When it detects one event that is worse than the action previously checked, this algorithm will be pruned, and no further tree will be constructed, which will save a fair amount of time.

The key idea of the *Alpha-beta* algorithm is that there are two type of nodes, *max* and *min* representing two opponents, alternately from top to bottom forming the tree. Player 1 starts with initial large negative *g value=-99999*, its goal is to find the largest evaluation *g value* and *best-move* position by the comparison of branches thorough the tree. While the optimal direction for player 2 is updating *g value=99999* from a large positive number to the smallest one.

Figure 1 is an example sketch showing the *depth=3* search tree used in *Alpha-beta*, the root is the max type node, the second level whose type is the *min* presents there are three possible movements for opponent, third and forth levels are again *max*, *min* repeatedly belonging to player 1 and player 2 respectively. The last row of numbers represents the evaluation toward that certain strategy. The search direction inside the algorithm is from root to leaf node all the way down, then transfer *a,b,g* upward from bottom to root. For example, the order of travel the left branch tree in figure 1 is $A \to B \to E \to L \to M \to E \to F \to N \to O \to F \to B$.

Another feature in *alpha-beta* is *cutoff* relevant to the visited order. Take the first branch - second branch $F \to N \to O \to F$ for example, F is a *max* node, the *g value* of later coming in branch O is smaller than the former coming in node $N(3 < 9)$, hence the algorithm prunes the branch O; while for the first left - second left branch, the later coming branch M is better than previous $L(8 > 4)$. Eventually, root A will pick the maximum *g value* from children B, C, D and place corresponding (x, y) in the board.

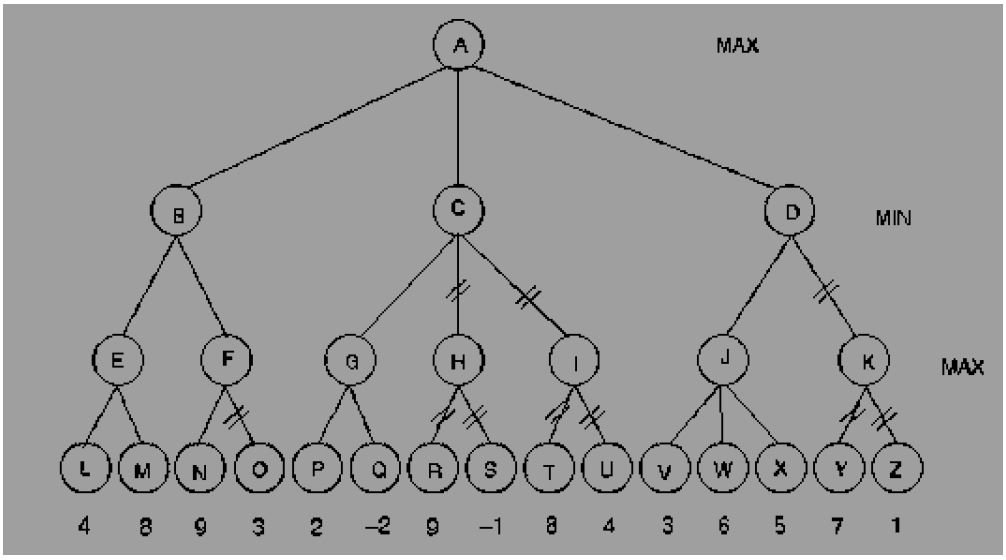


Figure 1: credit: author C. R. Dyer, CS 540 Lecture Notes, Formulating Game Playing as Search, University of Wisconsin - Madison.

2.1 Implementation

In our code, we use tree structure.

1. Firstly, we transfer the root node into the function, add all possible children nodes.
2. Use the info of *depth*, if reaching the leaf $d = 0$, based on method (*alpha-beta* or *dijkstra*, will be discussed in section 2.2), return evaluation result of that certain path to *g value*.
3. If not reaching the leaf $d \geq 0$, decide run which turn(*max*:computer aims to maximize *g* / *min*:opponent aims to minimize *g*) based on node.color, recursively call alphabeta

itself with input (n=all children) in a loop going downward. Then send information (*node*) upward in the tree, the parent will get it. With this evaluation number, update and store g in *node.gvalue*, shrink *a,b* which is a kind of boundary, a starts from minus infinity while b from positive infinity, the algorithm does cutoff once they meet each other $a \geq b$. The equivalent text explanation is if the later coming in children's parameter value is not better compared to existed one, the algorithm does cutoff action.

4. If $d = depth - 1$ meaning at the second level of the tree which contains the possible placement decisions that computer could do. Since we choose computer color=my color being the *max* turn, we pick the largest g *value* among nodes in the second level, store in global variable best move. When complete travel all children in the second level, the best move at this moment is the eventual result.

5. If reaching the root $d = depth$, return the best move node recorded from its children.

2.2 Leaf evaluation function

For the evaluation function, we start by implementing a dummy random evaluation with a random number generator. By doing so, our program is a blind search in order to test whether the alpha-beta tree cutoff and the hex game interface work fine.

We mainly focus on the Dijkstra evaluation method afterward, since it could return a figure after evaluating the state of the hexboard according to the distance between the current existed "bridge" and the edge, thus the computer could do better decisions. The heuristic method does not traverse the final position of the tree, but traverses it to the depth of the end of the game, but creates artificial leaves at a shallower search depth. The deeper search is artificially restricted, and a heuristic is invoked to statically evaluate any location that occurs at that depth.[1]

Dijkstra algorithm aims to find the shortest path between nodes in a graph. For example, the graph can be a road network. It is a greedy search algorithm and dynamic programming especially for finding the shortest path on graph data. Recently, it can fix a single node as the source node, and find the shortest path from the source to all other nodes in the graph, thereby generating the shortest path tree. In this way, it leaves more space for the combination with other methods, which can be used to solve more complex problems in life. The standard baseline of the Dijkstra algorithm is following

1. Store all the nodes and mark them "unvisited;

2. Initialize the distance of the starting point to "0" and others to "INF";

3. Pick the node with the smallest distance from the "unvisited" node-set, and make it as the current point now;

4. Find neighbors from "unvisited" set for the current node, and calculate their distances between the current node;

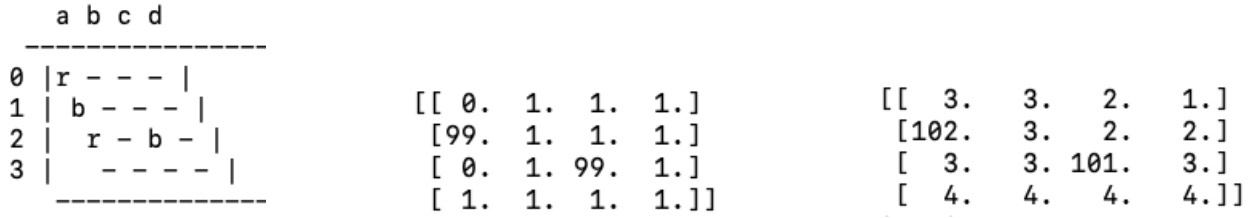
5. Compare the newly calculated distance with the assigned one and memorize the smaller one;

6. Mark the current node as "visited" and remove it out of the "unvisited" set;

7. Check whether the destination node has been labeled "visited" or whether the smallest distance among the "unvisited" set is "INF". If not, repeat steps 3-7.

2.2.1 Dijkstra implementation

In our code, we don't strictly follow the guidelines, we create our own algorithm but mainly follow the key idea of *dijkstra* which is efficiently finding the shortest path. We first derive one step cost array named *Graph* in our code which is the cost of one point if we want to go through it. Then calculate the *Distance* which stores the accumulated cost starting from one of the edge points to all other points in the board, like a cumulative distribution function if we regard *Graph* as a possibility distribution function. Basically, there is one *Graph* and size number *Distance* for one board since there are size number starting points.



(a) A Board (b) Graph array (c) Distance array

Figure 2: An example of three output arrays in our *dijkstra* algorithm, left panel: a board; center panel: the *Graph* array if computer color is red, which agrees with the rule stated above, the red cost is 0, the blue cost is 99, the empty cost is 1; right panel: calculated *Distance* array starting from the corner right point the last row [4,4,4,4] is the required win accumulated cost going from the start point.

Here is the way to generate cost *Graph* in our code, if the position(x,y) is empty, the cost is 1; if consistent with computer's color, cost is 0; occupied by opponent, cost is 99. Then is the method to derive the *Distance* array. Initially we set the start point in line with its value in *Graph* array since accumulated distance of the start point is just the one step cost; set all other elements to a large number. We update the accumulative distance of center position's neighbor by adding the *distance[center]* and *Graph[neighbor]* if is smaller. During the process, we keep calling recursion function repeatedly, that is set current neighbor being center, get the distance of its neighbor. Figure 2 shows one example board and corresponding *Graph* and

Distance arrays. The center panel is the *Graph* array if computer color is red, which agrees with the rule stated above, red cost is 0, blue cost is 99, empty cost is 1. The right panel is the calculated distance array starting from the corner right point the last row [4,4,4,4] is the required win accumulated cost going from the start point.

During the game, we find if we keep visiting the neighbors without any flag since we are also our neighbor's neighbor, the function will be stuck into a loop (as shown in figure 3), and the calculation will not end. Thus we create an array named *index* to store if that lattice is can be updated or not. It is slightly different from the visited array in the standard practice. When the *Distance* value in that position can be updated, that is the new path has a better distance value, set index[position] to True, and only if index[position]=True, we would continue getting its neighbor by recursively calling the function. Figure 4 shows an example about the index array change during the process of finding the neighbor.

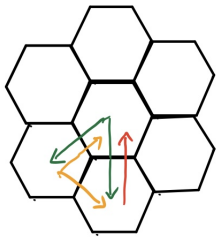


Figure 3: Left bottom, center bottom, center grids are the neighbor of each other, if without any visited flag array, the function will be stuck into a loop.

Once we have accumulated *Distance* array, we have to decide to take which property as evaluation result. We eventually choose the sum of all elements in *Distance* array, not the smallest number in edge row or column though they are the shortest-path definition. Take the example in figure 2, the output is $\sum(Distance) = 244$ instead of $min([4, 4, 4, 4]) = 4$. Here is our reason for choosing it. Because the board size is small(=3 or 4), the possible return results are $[0, 1, 2, 3, 99(INF)]$ if using the minimum, which indicates the little distinction. Moreover with search *depth*=3 or 4, there are a great number of boards simulated under the same guess movement in the second level of the tree. One thing that is likely to happen is the returned *g value* in the second level will all have distance="0" this option. Then the best move will always be the first added child node leading to a regular movement pattern which is no more what "Dijkstra"'s goal. Although our practice have large distinction, we realize that this robust practice that takes the sum of distance array would introduce some bias, eventually cause limitation in our search algorithm.

Note, in *Alpha-Beta*, we keep find max g value for computer turn, thus the return evaluation shortest distance of computer should meet this criterion: the better, the smaller criterion, then we return the minus one times the original result, that is $-sum(Distance)$.

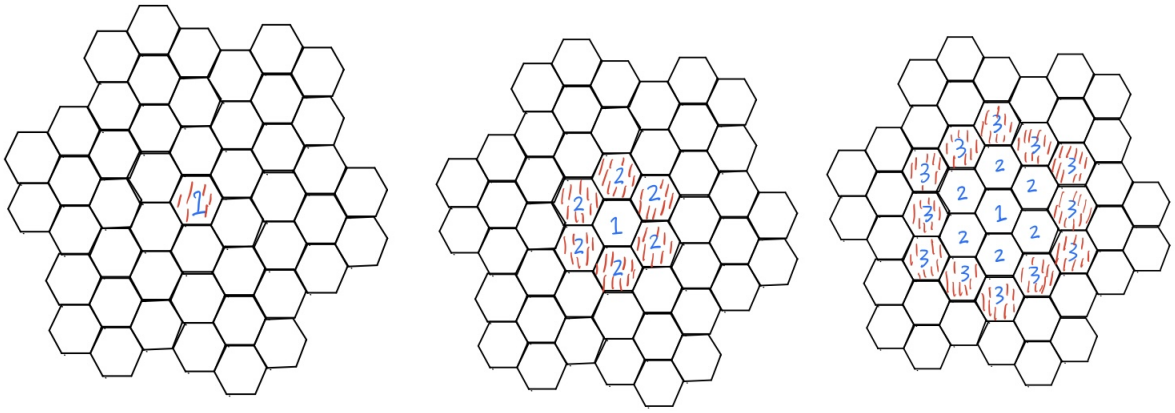


Figure 4: The *index* array changing on an empty board, blue value is the distance, red filled indicates the index=True. When the program starts, the center marked as True; as the program runs, find the center's neighbor, their index[neighbor] become True since the newly calculated distance is smaller than the initial infinity value; next turn the ring begin to find their neighbor, they won't find back to center because current center's index value is smaller than 2+1, this true-false judgment turns the index[center]=false.

One last thing to note, in the board game, when we make a move, we have to consider the opponent's advantages as well, hence we calculate the opponent's distance array with the same routine but this time output with a plus sign. Eventually, the output evaluation is opponent sum(distance) minus computer's sum(distance), i.e. $\sum(Distance)_{computer} - \sum(Distance)_{opponent}$.

2.3 Enhancement: IDTT

Iterative deepening and transposition tables are important enhancements of *Alpha-Beta* programs. After applying them, Alpha-beta could have three advantages.

1. It becomes an algorithm that can be stopped at any time, and according to the last search depth completed, an approximation of the best strategy and value function can be obtained.

2. It can be applied with good move ordering. As mentioned in section 2, *Alpha-beta* does cutoffs based on the previous branch returned(a,b). With adjustment on the order of visit, the (a,b) would shrink earlier leading to more cutoffs. It has been noticed that even if a given depth is about to be searched, iterative deepening is faster than searching the given depth.

3. The transposition table(TT) is a cache of previously seen positions and related evaluations in the game tree generated by the computer game program. If the position is repeated through a different movement sequence, the value of the position will be retrieved from the table to avoid re-searching the game tree below the position. The use of the transposition table is essentially the memory applied to the tree search, and is a form of dynamic programming.[2]

2.3.1 IDTT implementation

The key idea of iterative-deepening is it keeps searching the deeper tree until the time is up or reaches the given depth. Although it seems the previous efforts are in vain, the required search time grows exponentially. Thus algorithm actually spend most of time on the last depth search.

In our code, outside the *Alpha-Beta*, we write an *iterativedeepening* function that aims to search deeper to realize the first benefit; the core is *Alpha-Beta* mainly, apart from two more functions *lookup* and *store* to achieve second and third benefits. When implementing *store* function, we use dictionary type data, take *node* as the key and *gvalue*, *depth*, *bestmove* as the value. *lookup* function will find if there is the same node stored in *transpositiontable*. When going downward the tree, the normal order is traveling every node from left to right. Here, with *transpositiontable*, we could move shallow searched best move option front with the command *children.insert(0,ttbm)*. Recall from section 2, the cutoff happens when *g value* of later entered node is worse than current best *g value*. It has been noticed that shallow-search best movement is always the best choice for deepening search. With order alternation, the possible best movement is searched first so that *(a,b)* shrinks rapidly, making more prune being cutoff.

3 MCTS

The second algorithm we implemented is Monte-Carlo tree search (MCTS). Different from heuristic planning, MCTS uses sampling to play games. In each simulation, it expands a leaf and plays against itself (self play) by randomly making moves until the game is over, which is called a path. MCTS simulates many paths and calculates the winning rate of moves. The move with the highest winning rate is our next move. In our code, we still use the UCT selection rule to calculate the winning rate, but c_p for this calculation is always 0.

3.1 MCTS algorithm for Hex game

We use a Tree structure for MCTS. The root node is the current state of a game, and all the rest empty places on the board are its potential children. Each node stores the number of win times and the number of visited times. The algorithm mainly consists of four parts:

1. select: In each simulation, MCTS traverses from the root node until a leaf. For a fully expanded node, MCTS uses the UCT rule to select a child node that has the highest UCT. The UCT selection rule is:

$$UCT(j) = \frac{w_j}{n_j} + c_p * \sqrt{\frac{ln(n)}{n_j}} \tag{1}$$

It combines exploitation and exploration: if c_p is larger, it tends to explore a node that is less visited.

2. expand: When we reach a node that is not fully expanded, we randomly pick one child from its unexplored children list and add this leaf to the tree. We add one leaf in each simulation

3. play out(simulation): From the leaf, MCTS makes a move randomly in self-play until the game is over. Hex is a zero-sum game and does not exist a draw. If it wins, the reward is 1; if it loses, the reward is 0.

4. back propagation: The reward is propagated back upwards in the tree. The visited number +1 for the leaf and its parents. If it's a win, the win number of the leaf and its parent node +1. We don't need to store information for moves(simulated) after current leaf, because they are not in the tree.

Though it seems strange to let the agent self-play randomly, the results(winning rate for a move) become statistically stable and are close to the real world after enough simulations or sampling. MCTS is an adaptive sampling because we use the selection rule(UCT): it visits nodes with relatively high winning rate, but still have chances to explore nodes with relatively low winning rating. This setting ensure MCTS to work efficiently and are not stuck in some local optima. We referred to the pseudo code on the text book (Listing 5.1) and implemented the algorithm. MCTS has two parameters we can tune: C_p and the simulation number N.

4 Experiment

4.1 Text interface: human VS computer

Before the start of the game. Players need to choose their favorite in-game configurations to complete the initial settings of the game: First, the board size needs to be decided; Then, color (1 for blue, 2 for red) is chosen. As well as the direction of the bridge you are ought to built to win is determined, since it is bound to the chosen color (blue: left to right, red: top to bottom). Next, you determine whether you would like to play first or not.

During the game, Player and AI play the Hex game in turns. When it comes to player's turn, the coordinate referring to the location where you want to place the piece is input. At first, we intend to enter in the coordinate "(x,y)" format. However, the correspondence between

coordinates and the number of rows and columns is a bit puzzling for players. For this reason, the input is designed separately along with the descriptions, and consists with the output format in the hex_skeleton. x refers to the column and labelled by letters (a,b,c...), while y refers to the raw and marked by figures (0,1,2...). After entering the value of x and y, the program will check: 1) whether the location is inside the board, 2) whether there is a piece at this location. If the coordinate is invalid, the player will be told to enter again.
The program will do check_win for every single step. If either of two succeed, the game will be over and the winner will be put on the screen.

4.2 Computer VS computer

The board size for experiments is 3*3 and the two agents take turns to move first to get rid of the first-move priority. In games, blue is from left to right and red is from up to down. Blue is also the color for the first move. We used TrueSkill (mu and sigma) to assess the Elo rating(in this report, Elo rating = mu) and performance of two agents. The code is in main.py.

4.2.1 Alphabeta

Three experiments were done for alphabeta with random or Dijkstra evaluation:
1. random evaluation in depth 3 vs. Dijkstra in depth 3;
2. random evaluation in depth 3 vs. Dijkstra in depth 4;
3. Dijkstra evaluation in depth 3 vs. Dijkstra in depth 4.

Figure 5 shows the Elo rating of experiment 1: r1 is the Elo rating of random evaluation in depth 3, while r2 is the Elo rating of Dijkstra in depth 3. Left subplot only shows mu from calculated trueskill, and right subplot includes sigma for mu. We can see that the Elo ratings of these two agents become stable after about 40 games and their sigmas become smaller. The required game number from trueskill document to assess 2 players' skills is 12.[3] After 12 games, we can clear see that Dijkstra is better than random. After 80 games, Elo rating for random evaluation in depth 3 is 22.92 and Elo rating for Dijkstra in depth 3 is 27.07. The higher Elo rating means that our Dijkstra evaluation function works.

Figure 6 shows the result of experiment 2: Elo rating curves converge after about 20 games. After 80 games, Elo rating for random evaluation in depth 3 is 22.96 and Elo rating for Dijkstra in depth 4 is 27.04. Dijkstra in depth 4 is better than random.

Figure 7 shows the Elo rating of experiment 3: we found that Dijkstra in depth 4 is not better than Dijkstra in depth 3. After 80 games, Elo rating for Dijkstra evaluation in depth 3 is 24.93 and Elo rating for Dijkstra in depth 4 is 25.07. The reason might be that the board size is only 3*3, so depth 4 cannot improve the performance significantly. Furthermore, it could be found that our Dijkstra alphabeta was not super smart from the testing result of human

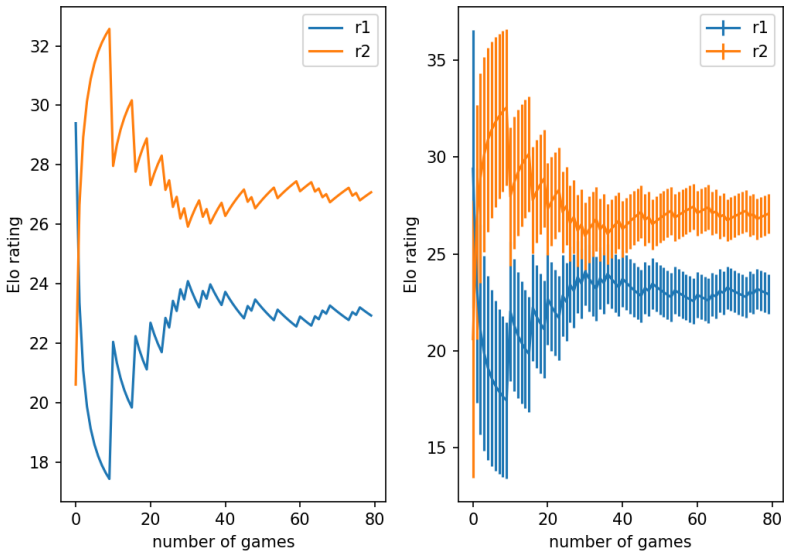


Figure 5: Elo rating of random evaluation in depth 3 (r1) and Dijkstra in depth 3 (r2). Left subplot shows mu of two agents after a certain game number and right subplot also includes sigma for mu (errorbar).
The total game number is 80. Dijkstra is better than random, which meets our expectation.

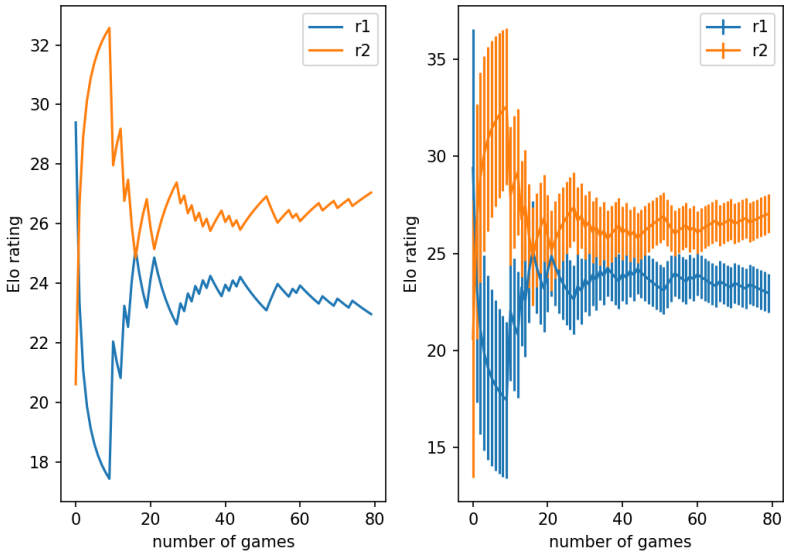


Figure 6: Elo rating of random evaluation in depth 3 (r1) and Dijkstra in depth 4 (r2). Left subplot shows mu of two agents after a certain game number and right subplot also includes sigma for mu (error bar). The total game number is 80. Board size=3*3. Dijkstra is better than random, which meets our expectation.

and computer plays, so the limitation of the algorithm contributes to this result more or less.

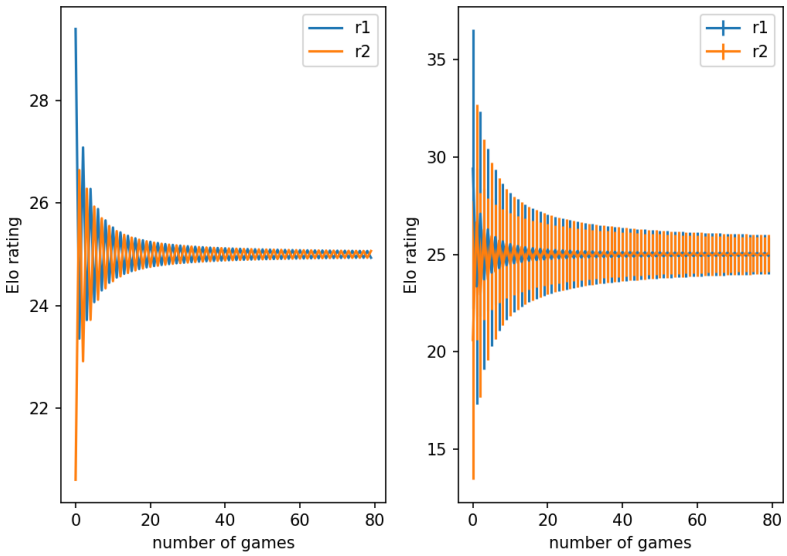


Figure 7: Elo rating of Dijkstra evaluation in depth 3 (r1) and Dijkstra in depth 4 (r2). Left subplot shows mu of two agents after a certain game number and right subplot also includes sigma for mu (error bar).
The total game number is 80. Board size=3*3. Dijkstra (4) is almost the same as Dijkstra (3) in a board of size 3*3.

In addition, the small vibration in the left panel comes from first move priority. Because our code becomes slow when board size is 4*4, it's very computational expensive to do this experiments for larger board. We cannot tell if our Dijkstra can work better depth 4 than depth 3. What's more, the three experiments after 20 games. The number is a little higher than the theoretical number from trueskill document $(2 * 2 \times 3 \times log_2(2))$.[3]

4.2.2 Iterative deepening and transposition tables

One experiment was done between alphabeta in depth 3 and alphabeta with iterative deepening and transposition tables (IDTT). The evaluation function for both agents is Dijkstra. We set the search max depth for IDTT to be 5 and max time allowed for search is 5 seconds. If the IDTT doesn't finish in 5 seconds, it returns the best move from the last search depth. We did observe many terminations because running out of time and the search depth can go down to 4 or 5 in some moves. The board size is 3*3 for this experiment. However the running time of IDTT is slower than alphabeta, probably because of our algorithm is not efficient. We made the two agents play 20 games. Figure 8 shows the Elo ratings of the two agents. We can see that like the experiment 3 of alphabeta with depth 4: though IDTT searched deeper, it wasn't better than alphabeta in depth3 for a board size of 3*3. Again, we need larger board

to see the IDTT's enhancement. The final Elo rating of alphabeta is 24.75 and 25.25 for IDTT. The total running time is 722 seconds, much longer than former experiments.

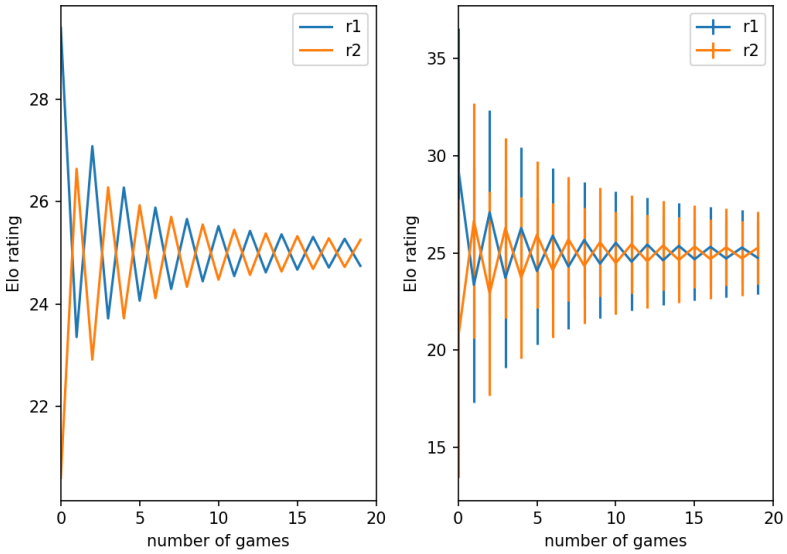
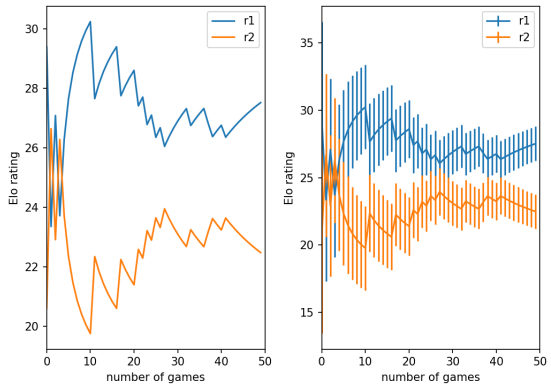


Figure 8: Elo rating of alphabeta in depth 3 (r1) and IDTT (r2). Both evaluation function is Dijkstra. Left subplot shows mu of two agents after a certain game number and right subplot also includes sigma for mu (error bar). The total game number is 20. Board size=3*3.

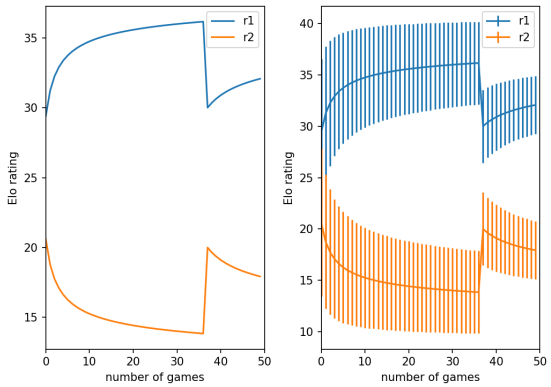
4.2.3 MCTS

Two experiments were carried out to compare the Elo rating of MCTS and alphabeta with Dijkstra evaluation for a depth of 3. The number of games is 50 for each experiment. The board size is also 3*3. We fixed the $cp = 1$ and the number of simulations N = 10 and 100. Figure 9 shows the Elo ratings of MCTS (r1) and alphabeta(r2) in the three experiments. MCTS is already better than alphabeta and tends to converge after about 20 games since $N = 10$. It is again because of the small board. But we still suspect it's related to the limitation of our alphabeta with Dijkstra evaluation. Because Heuristic planning should work very well when the board size is small. But now our MCTS just with $N = 10$ can beat alphabeta with Dijkstra. The final Elo rating of MCTS is 32.07 when $N = 100$ higher than the final Elo rating of MCTS when $N = 10$, which is 27.52. It agrees with our knowledge about MCTS: the higher simulation number is, the better and more stable result.

We also played two experiments between MCTS and IDTT. The settings are the same as the former experiments: boardsize=3*3; cp=1; N=10 and 100; max search depth=5; max time for IDTT =5s. The number of games is 25, because of the slow speed of IDTT. The results are shown in Figure 10. When N=10, MCTS is not stable in just 25 games; when N becomes



(a) Elo rating of MCTS(cp=1,N=10)(r1) and alphabeta with Dijkstra for a depth 3 (r2). Left subplot is mu values and right subplot includes sigma for mu.



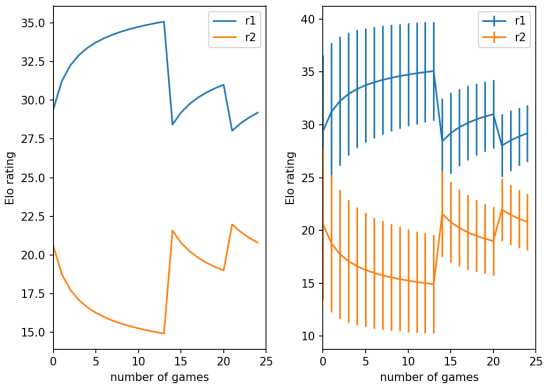
(b) Elo rating of MCTS(cp=1,N=100) (r1) and alphabeta with Dijkstra for a depth 3 (r2). Left subplot is mu values and right subplot includes sigma for mu.

Figure 9: MCTS against alphabeta with Dijkstra for a depth of 3.

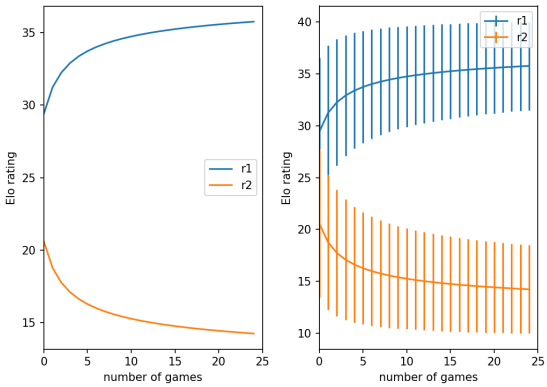
bigger, MCTS becomes more stable. We can tell that our MCTS is better than our IDTT in this case. Again, because IDTT is based on our alphabeta with Dijkstra, we suspect the limitations of our IDTT algorithm affect the result when N is very small. But in general, when N is larger, MCTS out performs alphabeta if we find good parameters. Though alphebeta is heuristic planning, it is not always better than MCTS. Because alphabeta has a certain search depth - even with iterative deepening and transposition tables, it's still limited by the running time - it can be stuck in some local optima. However, because MCTS simulations many paths until the end of the game and it has C_p to ensure adaptive sampling, it can escape some local optima and outperform heuristic planning. Also because heuristic planning assumes that the opponent also makes best move each turn. In reality, it is not always the case. MCTS is more tolerate to these changes. In addition, after 15 games, the results become statistically stable. When N is smaller, more games are needed for the results to be stable.

4.2.4 Parameter experiments for MCTS

C_p and N are two important parameters in MCTS. Thus we need to design parameter experiments to investigate their influences on MCTS's performance. Different from other experiments, we set the board size to be 4*4, because MCTS's running speed is relatively faster. And a larger board size is better for this kind of experiments as long as the computation cost is acceptable. We only use MCTS agents to play against each other, so there is no alphabeta agent in these experiments. We use the control variable method to do experiments. 1. N=[10,100,500]: for each N value, there are 4 MCTS agents with C_p=[0.1,0.5,1,1.5] and the total number of games is 500. During each game, we randomly pick out two agents to play against each other and update their Elo ratings.



(a) Elo rating of MCTS(cp=1,N=10) (r1) and IDTT (r2). Left subplot is mu values and right subplot includes sigma for mu.



(b) Elo rating of MCTS(cp=1,N=100) (r1) and IDTT (r2). Left subplot is mu values and right subplot includes sigma for mu.

Figure 10: MCTS against IDTT.

2. C_p=[0.1,0.5,1,1.5]: for each C_p, there are 4 MCTS agents with N=[10,100,500,1000] and the total number of games is 500. During each game, we randomly pick out two agents to play against each other and update their Elo ratings.

All Elo ratings reached convergence after about 150 games. In Figure 11, N=10 is bad for all C_p values and the Elo ratings are all around 25; when N=100, C_p=0.5 is the best and C_p=1.0 and 1.5 are slightly smaller than C_p=0.5; when N =500, C_p=1.0 is the best and C_p=0.5 and 1.5 is very close to C_p=1.0, while C_p=0.1 is much smaller than others. It means when N becomes larger, doing more exploration(larger C_p) improves the performance of MCTS. In Figure 12, when C_p is fixed, increasing N can improve MCTS's rating, because it becomes statistical stable. Overall, parameter experiments shows that when N is relatively small (e.g. you have limited computational resources or the search space is very large), we should choose a small C_p to do more exploitation; when N is relatively larger, we should choose a large C_p to do more exploration.

5 Conclusion and Discussion

We implemented two kinds of algorithms: Alphabeta (heuristic planning) and Monte Carlo Tree Search (adpative sampling) to play the Hex game. Alphabeta includes random and dijkstra evaluation functions, and is improved by iterative deepening and transposition tables. We fully understand the main idea of the algorithms and implement them in our own style. We wrote a text-based interface for human to play Hex game against computer. And We did experiments on different algorithms to evaluation their performance. We list our results and conclusions as following:

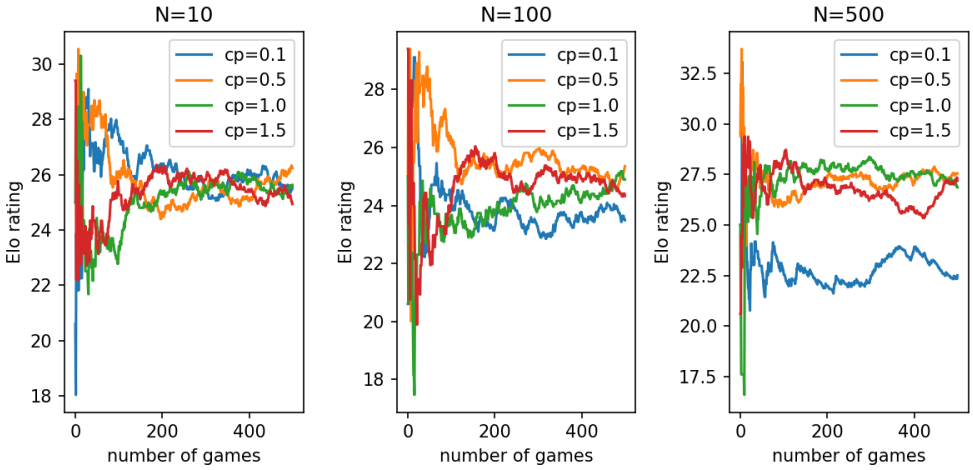


Figure 11: Four groups experiments on N and C_p. The number of games for each subplot is 500. In each subplot, N is fixed and the 4 MCTS agents with different C_p plays against each other.

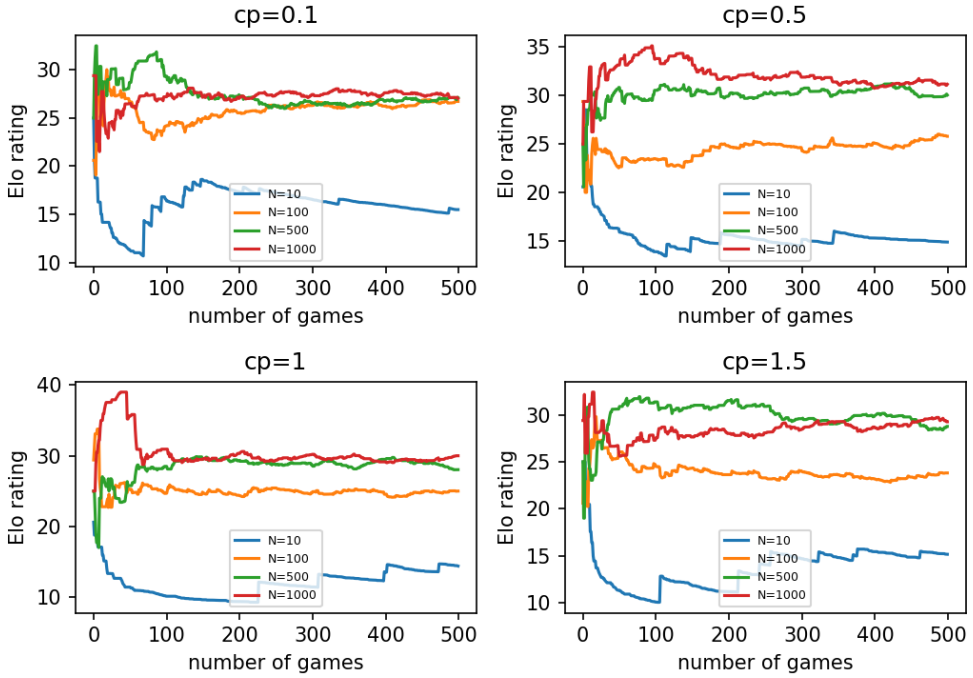


Figure 12: Four groups experiments on N and C_p. The number of games for each subplot is 500. In each subplot, C_p is fixed and the 4 MCTS agents with different N plays against each other.

1. Our Dijkstra heuristic function is not as perfect as we thought, but it yields a good evaluation for the state of the Hex board.

2. Our Alphabeta enhanced by iterative deepening and transposition tables (IDTT) has three advantanges:can stop when running time exceeds allowed time; allows ordering good moves (so can found more cutoffs); searched state will not be searched again. But its enhancement for a board of size 3*3 is small. Further modification is needed to make it work efficiently on a larger board in both running time and storage.

3. Monte Carlo Tree Search (MCTS) algorithm is relatively simpler to implement than Alphabeta, but it works better than our Alphabeta algorithms. The two parameters C_p and N have significant effects on the performance of MCTS and need to be chosen properly in different situations.

4. Almost all experiments become stable after a small number of games (30), except for MCTS's parameter experiments (it has four agents not two). Trueskill needs about 12 games to estimate the real skills for two players. In fact, after 12 games, we can already see which one is better.

References

[1] Aske Plaat. Learning to play-reinforcement learning and games, 2020.

[2] Reinforcement learning 2021 practical assignment 1: Search, 2021.

[3] https://trueskill.org.