RL Assignment 1

Stephan van der Putten
s1528459

Murad Bozik
s2619822

Yuchi Zhang
s2724952

February 28, 2020

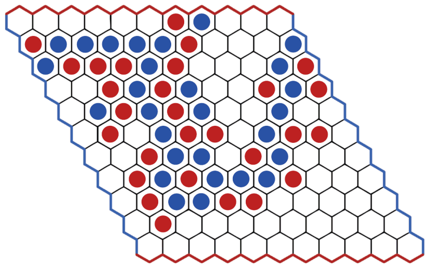


Figure 1: The game Hex. Red tries to connect top with bottom. Blue tries to connect left with right to win. Blue won in this game.
Source: `www.krammer.nl/hex/`

1 Introduction

In this report we will discuss the game Hex. Specifically we are going to implement Hex with heuristic planning and report our findings and problems. The game will be represented using a smaller $N \times N$ board. Before we start a quick introduction to Hex. Hex was originally invented in the 1940's[1], see Figure 1. The original game consists of and 11 x 11 playing field, which consists of hexes. The name Hex comes from ancient greece and means six. The players need to connect a path from one side of the board to the other. For which each side of a hex connected to another hex can form a path.

The reports consists of multiple sections. In section 2 we will talk about our building tools. Section 3 contains the heuristics or algorithms that we apply in our board game. The following section 4 is about the heuristic scoring and Dijkstra. Section 5, we talk about improvements to our standard algorithms using iterative deepening and transposition tables. In section 6 we talk about

[1] https://en.wikipedia.org/wiki/Hex_(board_game)

experiments we will perform. In section 7 our results are shown. We conclude the report with section 8, 9 that contain our discussion and conclusion and future work.

2 Preliminaries

The game is created using the programming language Python 3. A few extra packages were used, which we will briefly note: Graphviz, Trueskill, Numpy, keyboard. Furthermore we have received a base hex skeleton code from the course Reinforcement learning at University Leiden, given by Aske Plaat that we modified. A notable change from the original Hex game is that we did not include a Pie Rule, which eliminates the first player an advantage.

2.1 General approach

Our goal is to create an playable game. We divided the game in three files. One file consist of the main playable game, the other consists of the board and hexes and the thirth are the utilities. The main *game.py* can be run using the python command; it contains the functions to run the game. The *hex_skeleton.py* consists of the two classes, Hexboard and Hex. The hexboard consists of multiple hexes and the hexboard is able to perform moves and check the current game state. The *util.py* file contains the class UTIL that creates nodes (games states) and the bulk of algorithms. The computers generates moves based on the dijkstra evaluation function or dummy evaluation function. Our game is only tested on small hex boards of size $3 \times 3 and 4 \times 4$. 5×5 have been partially tested due to time limitations.

2.2 Short User Documentation

In this section our program is playable against computer. In order to start the game, *game.py* file should be executed in terminal. When the game start, the player will face two question:

1. Enter the board size you want to play:
2. In order to start the game, please choose a side 'R' or 'B':

The second question it not case-sensitive. After these two question are answered, game will begin. Everything will be cleared on the terminal screen and the board will be printed. The game will ask player for the coordinates to play:

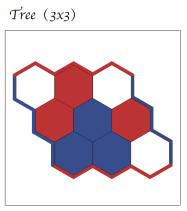
1. Choose the coordinate to place your color:
2. Example: 2 c :

It is important to answer this question as the example shows. But you don't have to use space between coordinates. If you write a coordinate which is out of bound, the game will tell you that the coordinates are out of board size.

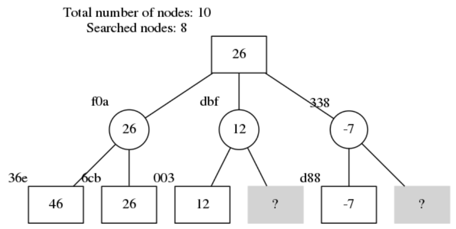
However if you use a letter for the first coordinate it will give an error and exits the game.

2.3 General Results with dummy

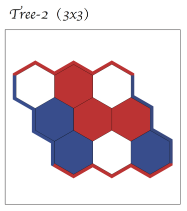
In this section we used only dummy evaluation function. Our program creates trees. For visualisation *graphviz* library is used, see Figure 2. In the figure max nodes are represented by box shape, min nodes are shown in circle shape. Unvisited nodes are shown with a grey background and they have no value. The total number of nodes and how many node are searched are presented in the visualizations.



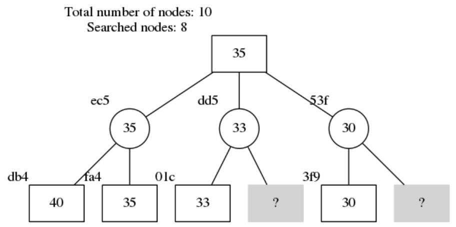
(a) Board State 1



(b) Search tree 1



(c) Board State 2



(d) Search tree 2

Figure 2: Board states and their respective search trees

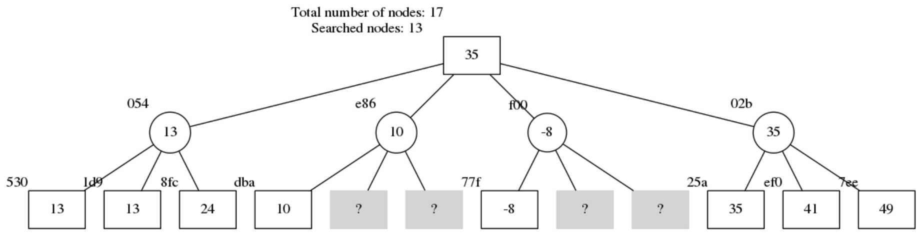


Figure 3: Search tree of 2x2 empty board state

In this stage our game is playable manually. Using the *game-section1.py* file and *VisualizeTrees.ipynb* similar results can be achieved as shown in Figures 2 and 3.

3 Algorithms

This section contains the search and evaluation algorithms implemented for Hex.

3.1 Alpha-beta Search

Alpha-beta is an improved minimax tree, which means it includes a pruning technique [3]. The alpha and beta values represents the boundaries. In order to use alpha-beta algorithm we needed to create a game tree. State space becomes huge depending on the board size. It is almost impossible to store all state space. The methodology of our implementation is create children for the nodes as go deeper. And after we found the best move and made that move we are updating our root node with current board state. For the next turn we already have its children and if the depth requires to get more children then we create them. This approach gives us a window to look into state space. Our implementation can be found in the scratch *alphabeta.py* or our main game files under scratch directory.

4 Heuristic Score Evaluation & Dijkstra

The score evaluation is the most important part for this game, because the score determines if the heuristic is good or bad. The score should consider two aspects: how many steps remain to win the game or how to prevent the other connecting the two edges. In the implementation, say red is the first hand, and red side want to maximise the score and the blue side want to minimise the score, we use the formula as the score evaluation [2]:

$$score = blue\_remain - red\_remain \tag{1}$$

in formula 1, blue_remain stands for the least steps for blue to connect the two side edges, and red_remain stands for the least steps for red to connect the two side edges. Red is the first hand, and it will try to maximise the score: make blue_remain larger or make red_remain smaller. Making blue_remain larger means the step will block the blue from connecting the two side edges, and making red_remain smaller means the step will try to shorten the distance of connecting the two side edges. The score will balance the two aspects.

The problem is how to calculate the blue_remain and red_remain, the shortest distance to connect the two sides. In our implementation, we use graph theory to solve this problem. First we build the graph model, the graph is an undirected graph. Each hex on the hex board is abstracted as the node on the graph, each

[2]https://towardsdatascience.com/hex-creating-intelligent-adversaries-part-2-heuristics-dijkstras-algorithm-597e4dcacf93

hex has 6 neighbours, then connects the hex with its 6 neighbours with 6 edges. We label the hexes with red or blue, if occupied, and white if empty. We define the following rules as the edge weights. We select any hexagon $h1$ on one edge and select any hexagon $h2$ on the opposite side, the red_remain can be calculated by calculating the shortest path from h1 to h2 on the undirected graph. The reason why we can choose any hexagon is the distance from $h1$ to any hexagon on the same edge is 0, and so is $h2$. So the shortest path distance from h1 to h2 will be the least remaining red hexagons to connect two side edges.

Dijkstra algorithm is the single source shortest path algorithm in Graph Theory [1]. The algorithm is invented by Dijkstra. The algorithm is has the following pseudocode:

```
function Dijkstra(Graph, source):
    dist[source] ← 0         // Initialization

    create vertex priority queue Q

    for each vertex v in Graph:
        if v  source
            dist[v] ← INFINITY   // Unknown distance from source to v
            prev[v] ← UNDEFINED  // Predecessor of v

        Q.add_with_priority(v, dist[v])


    while Q is not empty:        // The main loop
        u ← Q.extract_min()      // Remove and return best vertex
        for each neighbor v of u:    // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

.

The steps to calculating blue_remain are the same as red_remain. The implementation can be run using *game-section2.py*, we do recommend small boards. Our implementation can be found in the scratch *dijkstra.py* in our main game files under scratch directory.

5 Iterative Deepening & Transposition Table

Iterative deepening goes together with Transposition tables to create an improved search [2]. In a real life game of Hex time is limited which is why an algorithm can't simply search the entire search space. To help improve, iterative deepening only searches for a certain depth until time is done and return to best score so far. Furthermore improvements can be achieved by saving the best moves in the searched space in a transposition table. When iterative deepening needs to calculate the next move, it can continue based on the moves in the transposition table. A notable disadvantage is the increased expansion of nodes.

We implemented iterative deepening as follows:

```
def iterativeDeepening(self, node, maximizer):
    depth = 1
    end_time = datetime.now() + timedelta(seconds=3)
    while datetime.now() < end_time:
        # Use line below if you want to press a key to stop
        ↪  iterative deepening
        # while datetime.now() < end_time and not
        ↪  keyboard.is_pressed('space'):
        bestScore = self.alphaBetaSearch(node, depth, -9999999,
        ↪  9999999, isMaximizer=maximizer)
        depth = depth + 1
    return bestScore
```

We implemented the transposition table as follows, for which *FirstTable* is a dictionary storing the states:

```
boardState = node['board'].copy()
try:
    move = FirstTable[boardState.tobytes()]
except KeyError:
    best_value = util.iterativeDeepening(node, True)
    move = util.getBestMove(node, best_value)
    FirstTable[boardState.tobytes()] = move
game = makeMove(move, game)
node = util.updateNode(node, game)
```

We follow the this report by experiments in which we first compare the ELO of the computers without iterative deepening and tranposition tables, run *game-section3.py* which has 3 combinations of computers playing against each other. The experiments with the improvements are run with *game-section4.py* comparing a depth 3 dijkstra algorithms against a depth 4 dijkstra. We will give further details in section 6.

6 Experiments

In this section we will talk about the experiments we performed. The algorithms in the previous few sections change the performance of the computer. As such we divided the experiments where compare the different computers using specific algorithms. The experiments will be run and compared using a rating system called Trueskill[3].

The elo rating of each player starts at $\mu = 25$ and changes with the win and lose rate. Our first experiments will consists of three programs. Each experiment will be run using 100 games. We created three folders for experiments. Fair folder indicates changing starting player halfway to fairly compare the computers. Blue folder indicates the first player was blue player during all 100 game. Red folder indicates the first player was red player during whole experiments. Inside these folders, each number states another 100 game experiments. The three players are as follows:

- search depth 3 with random evaluation
- search depth 3 with Dijkstra evaluation
- search depth 4 with Dijkstra evaluation

In the results we will try to answer the following questions:

1. How many games should be played for the rating to stabilise?
2. Can we calculate this number
3. How fast is our program, what board size will we use

In the second part of our experiments we will introduce iterative deepening and transposition tables. We will rerun the experiment comparing the dijkstra 3 and dijkstra 4 player and see how the elo rating changes, we limited the search time to 3 seconds.

7 Results

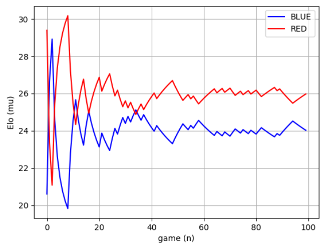
This section shows the results of the experiments. We will briefly discuss the results and give answers to certain questions.

The Figures can be seen in Figure 4. The results can be reproduced using the *game-section3.py* and setting the board size to 4 or 5.

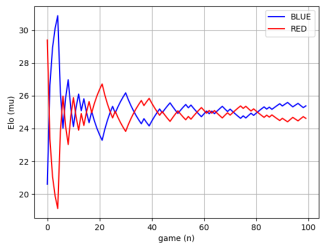
We answer the following the questions 6 of the experiments given our results below:

1. As Trueskill is based on statistics. As we see in our Figures 4, around the 30 to 40 mark we see that the ELOscores converges and stabilises.

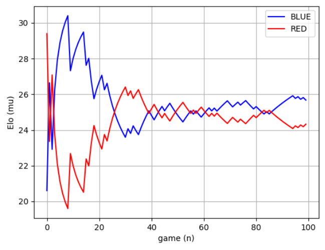
[3]https://trueskill.org/



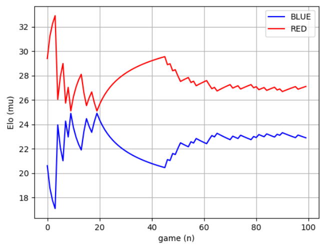
(a) Red: dijkstra with depth 3, Blue: dijkstra with depth 4



(b) Red: Random, Blue: dijkstra with depth 3



(c) Red: Random, Blue: dijkstra with depth 4



(d) Red: dijkstra with depth 4, Blue: dijkstra with depth 3 on a 5×5 board

Figure 4: Comparing ELO of a 4×4 board unless stated otherwise

2. We can state that from the central limit theorem that around 30 samples is the minimum and in our case we notice around the 30 cases are necessary.
3. In our experiments we used empty board sizes of $4 \times 4 and also 1 experiment with 5 \times$ 5. Although we did not note the exact time each experiment took in this report, this can be seen in the output of our experiment code. The range is somewhere between seconds for the most simple players to 2-3 hours for our most advanced players (to run 100 games).

Next up we introduce the improvements stated in section 5. Figure 5 shows an interesting change. The increase of games shows that there is only a monotonic ELO growth. We interpret the results as dijkstra with depth 4 that is having a winning streak against dijkstra 3, because dijkstra 4 is a smarter player.

8 Discussion

This section is to discuss the limitations of our report and issues we encountered that we would like to explain.

The Figures 4b and 4c clearly indicate that something is either wrong with our dijkstra algorithm or our random evaluation. A suitable solution would

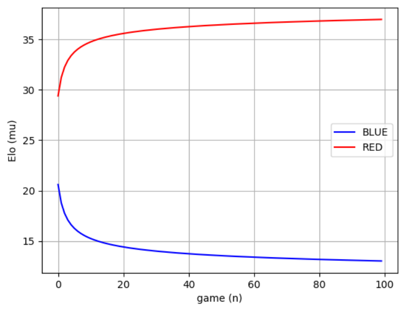


Figure 5: Red: dijkstra with depth 4, Blue: dijkstra with depth 3 with iterative deepening and transposition table

have been to perform testing on bigger boards using board states that have clear winners and losers. This would have helped debugging our programs.

The experiments based on the elo rating have been performed with only 1 vs 1 cases. With the 3 players it should be possible to have 1 rating to represent each player as is stated on the Trueskill website. We did not find out how this is achieved as such our results are a more simple version. We would however state that the methodology is not wrong only less conclusive in general.

Dijkstra algorithm's heuristics are based on the differences between paths. However, it is not a good heuristics, because in the board state some move are clearly much more important than other but they got the same value because path distances were same. Also at first we got best move based on Dijkstra evaluation value, although there are much more moves that have the same value we got the first one of these moves. During experiments we noticed that generally we got the best move from the bottom side of the board. That's why blue player generally won. Because our implementation had no tendency to pick move top to bottom. Then we changed our implementation, we stored all best moves than selected from them randomly. It gave us much more logical results.

We made our iterative deepening function time restricted. We also implemented keyboard listener in order to make it anytime algorithm, however the library 'msvcrt' was working only on windows os. We worked on both windows and linux machines (in the lab), in order to make our program os independent we removed keyboard listener implementation.

9 Conclusion

The goal of this report was the construct a Hex game with multiple advanced algorithms to improve the computer. As such we looked at implementation of alpha-beta search, dijkstra for evaluation and improvements through iterative deepening and transposition tables. Furthermore we experimented using ELO to compare multiple agents built using more and more advantages techniques.

In our experiments we noticed a converges of ELO rating for each player, which is around the 30-40 games. This is in agreement with the central limit theorem. Furthermore we created figures to compare the ELO rating of the different players and see that in general the deeper a player can explore the better the ELO rating. The noticeable exceptions are the times when a random evaluation function is used. We have no clear explanation for this, we can only conclude that there is a bug in the game which has not been fixed yet. Furthermore the report is focused on reproduceability and all experiments and figures can be recreated using the noted python code. Our final game can be played using *game.py*.

9.1 Future Work

We learned a lot form this assignment. The debugging has been underestimated and the code needs to be better structured. The readability in our code can be greatly improved, which would make reproducability easier. We performed our results using multi-threaded systems, however we need to also improve the effici"ency of our code. One way would be to write test code to check if our program can be improved. We belief our game should also have a nicer interface with more options. One goal at least is to create a single program that can perform all assignments.

9.2 Acknowledgements

This report is based on the Reinforcement Learning Course master course at University Leiden. The program is created in combination with the knowledge gathered from the course book [2], the lectures, the help of the lecturer or teaching assistants.

References

[1] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[2] Aske Plaat. *Learning to play.* 2020.

[3] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.