Reinforcement learning: Assignment 1
Heuristic Planning

Ismigül Işlak (2781077), Navin Pophare (2691957) , Huilin Li (2556057)

February 2020

Contents

1 Overview

Two player games have been studied the most in artificial intelligence.For this assignment, we will build a small game playing program for Hex, a snake building game. It is a two-agent, zero sum, turn based and perfect information game. The program consists of a search and evaluation function. Heuristics will be used for scoring board positions. Furthermore, various enhancements will be introduced and the resulting programs will be evaluated, rated and compared.

As mentioned in the lectures, the alpha-beta algorithm is the first step in optimizing the search of a game tree. This will result in a smaller search tree, because unnecessary leaf nodes are cut off. The leaf nodes are assigned values using heuristics based on Dijkstra's shortest path algorithm.

However, we should also note that one node (board state) might be searched twice, which wastes time. This can be solved by transposition tables that store all visited nodes and their corresponding values.

At last, we organized our thinking processes in the following chart.

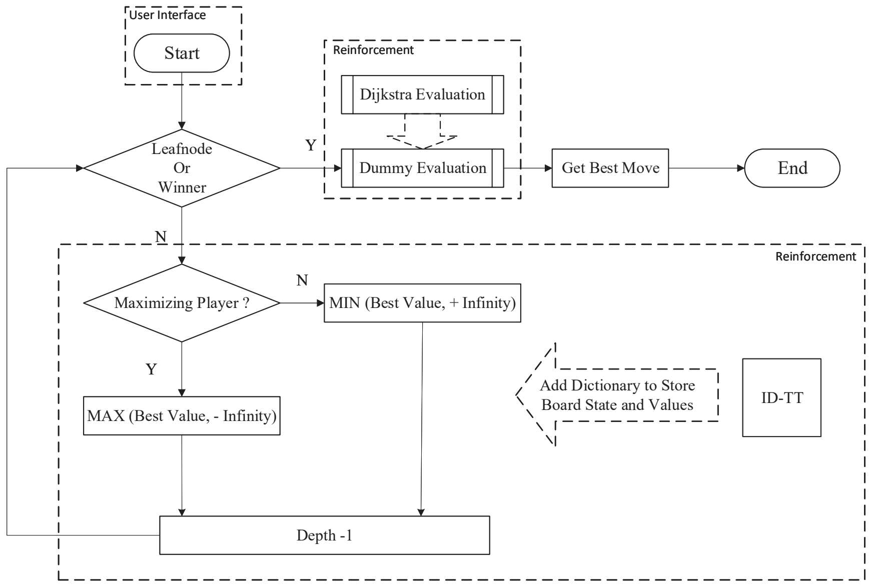


Figure 1: The Outline of all sections

2 Alpha-Beta with Random Evaluation

2.1 Search Function and Move Generator

The first step in our implementation of the Alpha-Beta algorithm was to generate a list of all possible moves depending on the board state (using the is_empty function). Two other useful functions were: **unmakeMove** and **makeMove**.

Our Alpha-Beta function has **7 input parameters: evaluation function, board, alpha, beta, maximum depth to be searched, type of player and the color of the maximizing player**.

In the function there are three possibilities:

• If depth 0 (a leaf node) is reached or if there is a win, the function returns a random integer number between -20 and 20 using the dummy_eval function.

Code Listing 1: Alpha-Beta Algorithm : Part 1

```
if depth == 0 or board.check_win(board.BLUE) or board.↩
    check_win(board.RED):
     g = eval_f(board, color)
    return g
```

• If the player is maximizing, the minimum value of the nodes is set to -INF. Every possible move is evaluated by calling the Alpha-Beta function (with a decreased depth and for the opposite player) to find its corresponding value.

Code Listing 2: Alpha-Beta Algorithm : Part 2

```
     elif is_max == True:
         g = -99
         m = {}
         for c in getMoves(board):
             makeMove(c, color, board)
             n_g = alphabeta(eval_f, board, a, b, depth=↩
                 depth - 1, is_max=False,
                                 color=color)
             g = max(g, n_g)
             m[n_g] = c
             unmakeMove(c, board)
             a = max(a, g)
             if g >= b:
                 break
         v[depth] = m
         return g
```

•If the player is minimizing, the maximum value of the nodes is set to INF.

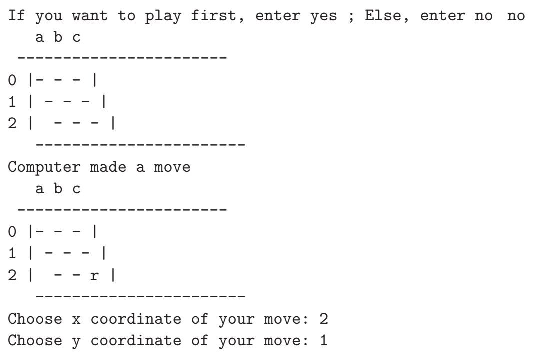
Every possible move is evaluated in the same way as described above. Except for the color used when making the move, for the minimizing player the opposite of the input color is used.

Code Listing 3: Alpha-Beta Algorithm : Part 3

```python
elif is_max == False:
        g = 99
        for c in getMoves(board):
            makeMove(c, board.get_opposite_color(color), ↩
                board)
            n_g = alphabeta(eval_f, board, a, b, depth=↩
                depth - 1, is_max=True, color=color)
            g = min(g, n_g)
            unmakeMove(c, board)
            b = min(b, g)
            if a >= g:
                break
        return g
```

2.2 Text-based user interface

For testing purposes, we added a **text-based user interface** that allows interactive playing of the game[2].



```
If you want to play first, enter yes ; Else, enter no  no
   a b c
 -----------------------
0 |- - - |
1 | - - - |
2 |  - - - |
   -----------------------
Computer made a move
   a b c
 -----------------------
0 |- - - |
1 | - - - |
2 |  - - r |
   -----------------------
Choose x coordinate of your move: 2
Choose y coordinate of your move: 1
```

Figure 2: User Interface

2.3 Test Results

2.3.1 Test the Depth

We created a small sized Hex board (2x2) and used search depth 3, to test whether the Alpha-Beta Algorithm works as expected.

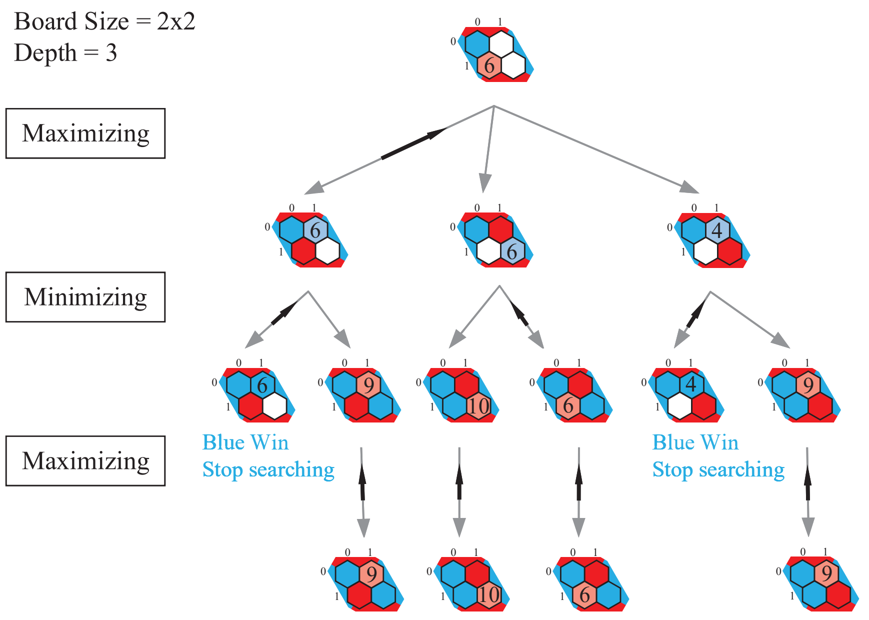


Figure 3: Trees with Depth =3

2.3.2 Test the Board Size

Next, we test the algorithm with board size 3 and search depth 2, to produce a bigger and more complex tree then above. The initial board state we used for this test included a BLUE piece on position (2,0). We set the maximizing player to RED. In this case, the total amount of possible nodes for search depth 2 is 56.

However, in Alpha-Beta Algorithm, the new parameters -alpha and beta provide the cutoff mechanism to decrease the number of searched nodes.

As this figure shows, after searching all of the children of node (0,0) of depth 1, the minimum value 1 from the child node (0,2) is returned. When the second child's value of the second possible move (0,1) of depth 1 is searched, due to the fact that Value(1,0)$\leq$ Value(0,0), the rest of the child nodes are not searched.

In total 17 nodes are cut off in this way.

2.4 User documentation

The **search.py** code that is provided can be used to find the minimax value or the best move for a given board state. So, the alphabeta function needs to be called with its 7 input parameters or the nextMove function. When using the alphabeta function, it is important to make sure that the global dictionary, v, is defined outside of the function and initialized with every call.

Furthermore, it is possible to interactively play the Hex game by calling the main function with the preferred board size as input.

2.5 Performance evaluation

Since, the Alpha-Beta algorithm is an enhanced version of the minimax algorithm, their performances are compared.

The time the programs need to find the optimal value of a board state is measured for different board sizes (between 2 and 11) and different search depths (between 1 and 6). The results can be seen in figures 4 and 5. It can be observed that after a board size of 5, the time for the minimax algorithm increases rapidly with increasing board size. In Figure 4, the same trend can be observed, starting from a search depth of 3.

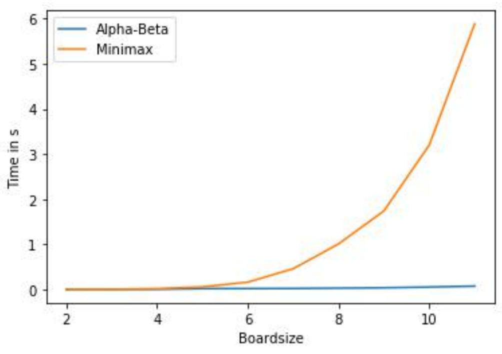
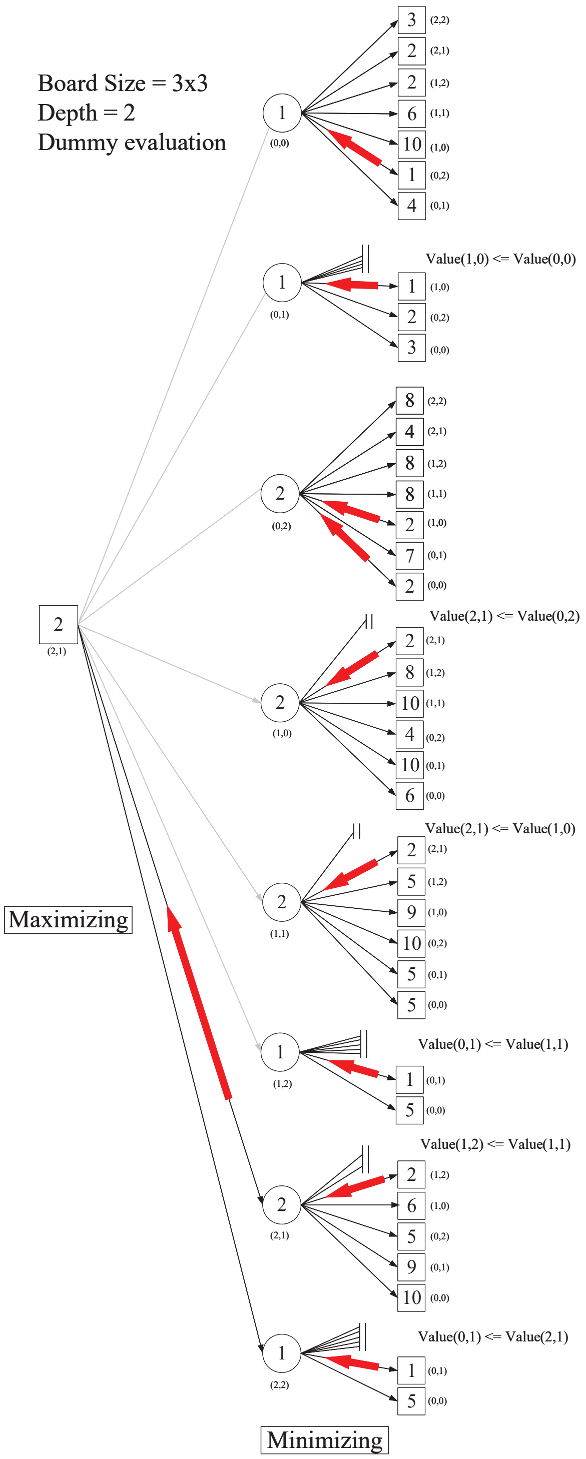


Figure 4: Empty board and depth 3



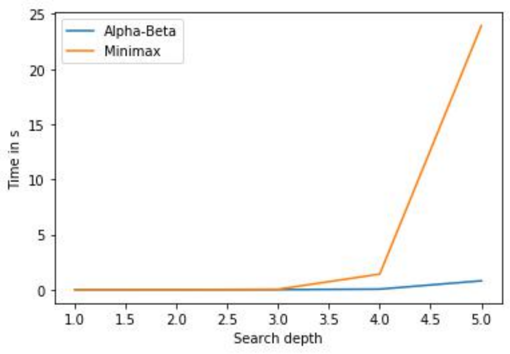


Figure 5: Empty board and board size 5

3 Dijkstra's shortest path

3.1 Dijkstra

We have calculated the heuristic value for a particular state as the difference between the remaining hexes for the Blue player and the remaining hexes of the Red player. **Specifically , if the current player is Red, the heuristic value would be: Heuristic = Remaining Hexes for Blue - Remaining Hexes for Red**[3] **and vice-versa if the player is Blue**.

Code Listing 4: Dijkstra's Evaluation

```python
def rem_hex(board, color):
"""To calculate the final heuristic value of a board state ↩
    and for a particular color"""
    red = dijkstra(board, board.RED)
    blue = dijkstra(board, board.BLUE)
    if color == board.RED:
        heuristic = blue - red
    else: heuristic = red - blue
    return heuristic
```

3.2 Test Results

We tested our program on different board states by generating a tree for the Alpha-Beta algorithm which uses the rem_hex() function for heuristic calculation. The picture in right depicts the search tree for an initial board state of(Blue(0,2)), which is same as the Random Evaluation one.

Dijkstra() has 2 input parameters: **the Hex board and the color of the player**.

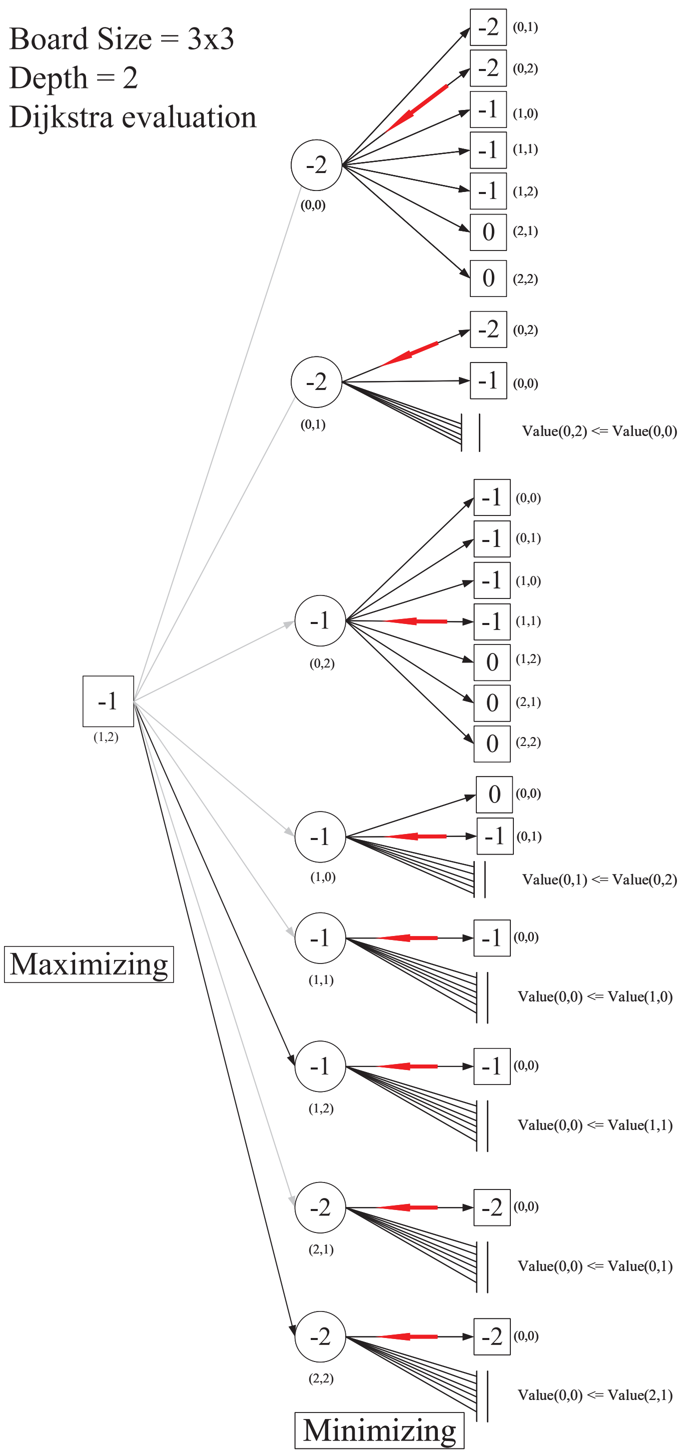
• Initialize the shortest distance to the start node as 0 and to every other node as infinity.
•Traverse each and every node from the source node and store the cost.
•Update the source node as the nearest neighbor of the earlier source node.
•Since we have connected the borders to extra nodes, we get some extra edges that are not a part of the actual board. So while calculating the shortest distance, the weights of these extra edges also get added up.
• To remove this extra distance, we just subtract 1 from the shortest distance calculated on the graph.
• If the border node is not empty and a piece of the player occupies it instead, the dijkstra() method will give the actual shortest distance because the edge connecting such node (filled with the player's color) to the extra node has a weight of 0.
• The method dijkstra() returns the shortest distance i.e.; the number of remaining hexes needed to be filled up in order to win. We calculate the remaining hexes for both the players.
• Define another method rem_hex() which subtracts the shortest distance for player Red from the shortest distance for player Blue to get the heuristic value for player Red.

3.3 User Documentation

We only have to invoke the rem_hex() method inside the Alpha Beta search function for the condition: depth=0 to get the heuristic value for that node.rem_hex() invokes the dijkstra()function to get the shortest distances and then rem_hex()gives us the heuristic value.

3.4 Performance Evaluation

We tested our algorithm by varying the board size and the search depth one by one keeping one of the two parameters constant and measured the execution times.



We used an empty board for this purpose. They are summarized below as follows:
•We observed that the search time increases with increasing board size.

Board Size	2	3	4	5	6
Time(seconds)	0.0010	0.1317	0.8252	3.1820	9.8812

Figure 6: For varying board size and constant search depth of 3

•We observed that the search time increases with increasing search depths.

Depth	1	2	3	4	5
Time(seconds)	0.1339	0.1392	0.1386	0.1402	0.1430

Figure 7: For varying search depths and constant board size of 6

4 Three Experiments

4.1 Elo Rating

For the evaluation of the strength of the programs that are created so far, the Trueskill rating system in Python is used. This system can identify and track the average skill of the players in a game. A rating is characterized by two numbers: the average skill of the player (μ) and its degree of uncertainty (σ). The ratings may change depending on the results of the games that are played.

Code Listing 5: Elo Rating

```python
def match(boardsize, n, eval_f1, d1, eval_f2, d2, ↩
    first_player):
    r1 = Rating()
    r2 = Rating()
    currentplayer = first_player
    count = 0
```

4.2 Performance Evaluation

Three experiments were performed to compare the strength of three programs: **search depth 3 with random evaluation, search depth 3 with Dijkstra evaluation and search depth 4 with Dijkstra evaluation**. In these experiments the ratings of the players are updated using the function **rate_1vs1** after every game played. Furthermore, the first player is changed with every game, because keeping it constant would lead to an overall advantage for the first player. This would result in unreliable ratings.

The ratings of the players are determined for a different number of games that are played. In this section, we assume there is a 4x4 board .

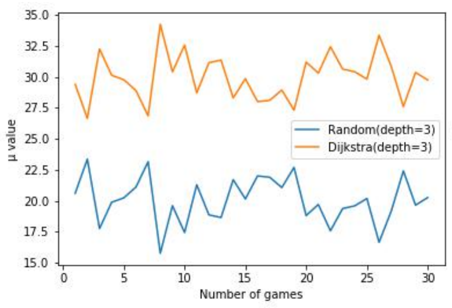


Figure 8: Random and Dijkstra in Depth=3

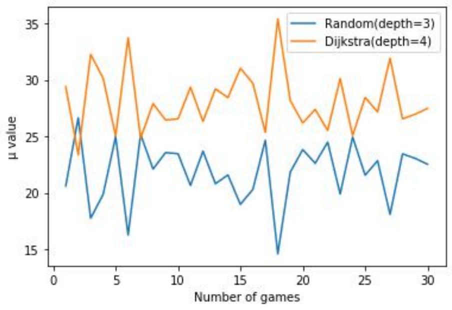


Figure 9: Random in Depth =3 and Dijkstra in Depth = 4

In Figure 8 we can observe that the player with Dijkstra evaluation (depth = 3) has way better ratings than the one with random evaluation. As we increase the search depth of Dijkstra from 3 to 4, we get better ratings for the player with Dijkstra evaluation in Figure 9.

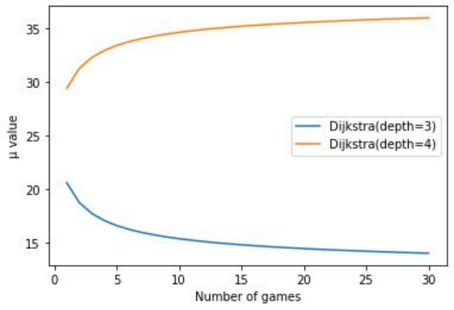


Figure 10: Dijkstra in Depth = 3 and 4

In Figure 10, we can see that the player with Dijkstra evaluation for depth 4 keeps getting better over the player with Dijkstra evaluation for depth 3. We can infer that the Dijkstra algorithm keeps getting better with increasing search depths, which means it is able to return the best moves efficiently.

5 ID-TT

5.1 Source code

Transposition Tables provide a dictionary to store the board state as key and the best move as values.

Code Listing 6: ID-TT(part)

```python
def alpha_TT(eval_f,board, a=-99, b=99, depth=2, is_max=↩
    True, color = board.RED):
    if board in table.keys():
        g = table[board_state(board)]
        return g
```

5.2 Board state Function

The board can be stored in a dictionary by converting it to a string of numbers. These numbers represent the coordinates and their colors

Code Listing 7: Board state Function

```python
def board_state(board):
    state = []
    for x in range(board.size):
        for y in range(board.size):
            state.append(x)
            state.append(y)
            state.append(board.get_color((x,y)))
    s = "".join(map(str,state))
    return s
```

6 What we learned and What we find

We implemented the Hex game in python and could search for considerably good moves for a player. We were able to use Dijkstra algorithm and Alpha Beta search algorithm together to search for good moves. After testing the Dijkstra algorithm against the random evaluation function, we can conclude that Dijkstra is not a perfect search algorithm but it can search for good moves decently. Also, we can conclude that Dijkstra keeps getting better with increasing search depths.