Reinforcement Learning – Heuristic Planning

Qing Zhou

Feb, 2020

1 Introduction

Heuristic planning is essential to reasoning and decision making and reasoning in Artificial intelligence. The basic idea of it is looking ahead a few steps, and try to evaluate how each decision will benefit your current situation. One implementation of it is the application in the board games, where players always try to plan a few moves ahead and see how it helps their current situation. The motivation of this assignment is to understand the fundaments of heuristic planning, the basic algorithms and possible enhancements, via a simple board game, Hex.

2 Search

2.1 Minimax

The core of heuristic planning is the search-eval architecture. Since the basic idea involves 'looking ahead a few moves', an effective way is to construct the search space as a tree, where the root nodes are all the possible current moves a player has, and the leaves are the nodes at the depth where you want to carry out the evaluation. For our Hex game what we expect the search algorithm to do is the following: for each input board state, the search function is first able to assess the attribute of each node on the board, if it is empty or already occupied. Once that is done the remaining is very straightforward: each 'children' of the 'root' is a possibility of your opponent's move, and there is an alternation from one player to another each time you move to a higher depth, and the evaluation is called at the leave nodes. This is the so-called 'Minimax' algorithm where the entire tree is traversed exhaustively, and best move is saved.

2.2 Alpha-beta

For a game with small board size this is possible but with the increasing of state space we need a more efficiently yet effective way to search the state space but without loosing information. The Alpha-beta algorithm provides such a way. Since not all nodes play a role in deciding the value of the root node, we can prune the tree a bit and keep only the branches whose nodes have influence on the value of the root node. The key to understanding this algorithm is that, two values are kept in the end, deciding the minimum value of the root player and the maximum value of the opponent, respectively. They are initialised with plus and minus infinity and players started with worst score, after each iteration the score of the opponent is assured to decrease and the root player increase, until the point where the root player has a minimum value which exceeds the maximum of the opponent's.

Now let us get started with a simple 2×2 board. The first player has already placed a red piece on $(1, 1)$ and now the algorithm should firstly search and then create a (partial) tree of possibilities to place the next piece. The current board is We are left with locations $(0, 0)$ $(0, 1)$ and $(1, 0)$. At

```
board = HexBoard(2)
board.place((1,1),board.RED)
print(board)
```

```
   a b
  ----------------------
0 |- - |
1 |  - r |
   ----------------------
```

Figure 1: Current board state.

each of this locations we create a tree with depth 2 and at the leave node evaluate. Now understand

```
search = Search(board, player='BLUE', timedelta=60)
```

```
def eval_1(n):
    if n.state.is_color((0, 0), search.player):
        return 1
    return 0


def eval_random(n):
    return np.random.randint(-10, 10)
```

Figure 2: Two evaluation function: evaluation with specific scoring point and random evaluation.

how the search algorithm works by writing the evaluation function in a way such that the output can be expected. Suppose that coordinate $(0, 0)$ is a scoring point that whoever gets it will score point 1. Afterwards if works as expected then the search function should be able to tell you the next best move is at $(0, 0)$ with a score of 1. The outcome is as follows The total nodes visited is 11, meaning

```
search.eval = eval_1
player, score, bestmove = search(method='alphabeta', depth=9999)
print("score of root", score)
print("solution tree", bestmove)
print("next best move", bestmove[0].state.lastmove)
print("Total number of nodes searched ",search.root.count()) #which returns the total number of nodes
```

```
score of root 1
solution tree [NodeTree(MIN_0), NodeTree(MAX_0), NodeTree(MIN_0)]
next best move ((0, 0), 1)
Total number of nodes searched  11
```

Figure 3: Test for the search function. The outcome shows that the search function has successfully found the scoring point at location $(0, 0)$.

that the search algorithm stopped at depth 2, the extra 1 node is the already occupied node $(1, 1)$ in the previous board state. The following shows the test result using a random evaluation function. For more testing functions for the search algorithm please see "test_search.py"

3 Eval

In the previous section we have known how the search is carried out, and how to seed up the searching procedure by prunning the tree created. Now we want to apply an algorithm which allows the evaluation to work properly. This is done by applying the Dijkstra's algorithm. It functions as follows: for each node, calculate its distances to its neighbouring nodes. This distance can be understood as a 'cost' which varies according to the occupancy of each node, meaning if one node is occupied by your opponent, even if it is in the neighbourhood of your previous move, arrving here is going to cost you infinity.

```
search.eval = eval_random
player, score, bestmove = search(method='alphabeta', depth=9999)
print("score of root", score)
print("solution tree", bestmove)
print("next best move", bestmove[0].state.lastmove)
```

```
score of root -2
solution tree [NodeTree(MIN_2), NodeTree(MAX_0), NodeTree(MIN_0)]
next best move ((1, 0), 1)
```

Figure 4: Test result for the search function with random evaluation.

The following case provides an example of a test of the Dijkstra's algorithm. The board is initialised such that the blue player is one step way from winning and the red is 7 steps away. More test examples see "test_dijkstra.py". For the algorithm to work it must be able to successfully

```
board_eval = HexBoard(4)
board_eval.place((2, 0), board_eval.BLUE)
board_eval.place((2, 1), board_eval.BLUE)
board_eval.place((2, 2), board_eval.BLUE)
board_eval.place((0, 1), board_eval.BLUE)
board_eval.place((0, 2), board_eval.BLUE)
board_eval.place((0, 3), board_eval.BLUE)
print(board_eval)
dijk = Dijkstra()

print("evaluation for player RED ",dijk(board_eval, board_eval.RED))
print("evaluation for playerBLUE ",dijk(board_eval, board_eval.BLUE))
```

```
   a b c d
 -----------------------
0 |- - b - |
1 | b - b - |
2 |  b - b - |
3 |   b - - - |
   -----------------------

evaluation for player RED  7
evaluation for playerBLUE  1
```

Figure 5: A 4×4 board with 7 Blue pieces.

calculate that for the Blue player it is now one piece away from winning and for the red player 7 pieces away. The result is as expected, meaning that the algorithm is functioning.

4 Experiment

4.1 Random depth 3 vs. Dijkstra depth 3

The test the programs against each other we start by a board of size 3.

```
"""Now change the board to size = 3. If player 1 (BLUE, random eval) wins, return 0; else return 1 """
def play(size=3):
    board = HexBoard(size)
    p1 = Search(board, player='BLUE')
    p1.eval = eval_random
    p2 = Search(board, player='RED')
    while True:
        _, score, bm = p1(method='alphabeta',depth=3)
        p1.play(bm)
        #print(board)
        if board.game_over:
            break
        _, score, bm = p2(method='alphabeta',depth=3)
        p2.play(bm)
        #print(board)
        if board.game_over:
            break
    if board.check_win(board.RED):
        return 1
    elif board.check_win(board.BLUE):
        return 0
    else:
        return None
```

Figure 6: The play function which allows programs to play against each other.

4.1.1 Board size 3

To test the performances of player using random evaluation with depth 3("Blue") and player using Dijkstra's algorithm with depth 3("Red"), we allow both players to deploy the "Alpha-beta" algo-

rithm, and let the "Blue" player with random evaluation to start. We keep records of the "Red" player so that each time he wins we record 1. Otherwise if the "Blue" player wins, we record 0. To test the performance of two players we use "Trueskill" to rate each play, and see the outcome after 200 plays.

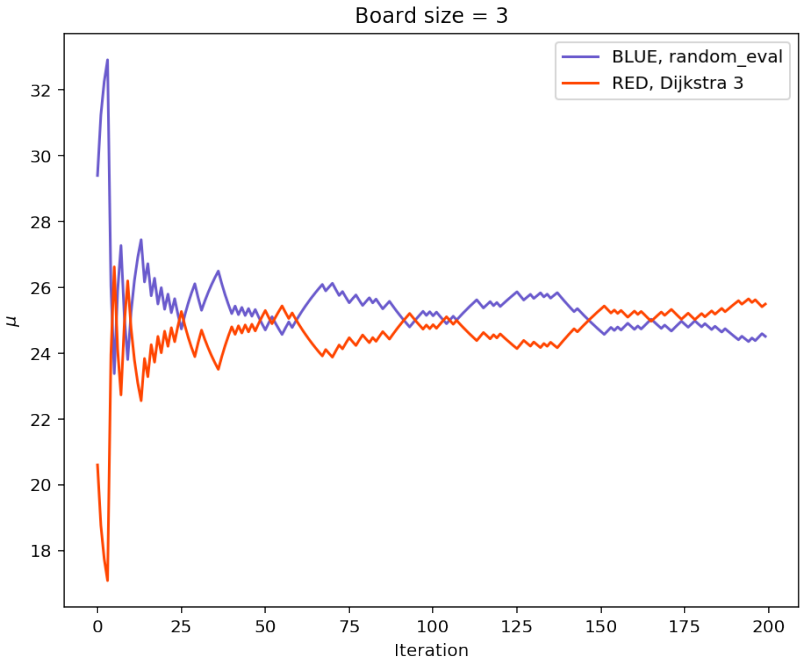

Figure 7: Allow 200 plays to see the ratings for a board of size 3.

The total run time for rating 200 plays is 7.49 s, with an average of 0.038 s per game. It can be seen from the above graph, after approximately 150 plays μ_1 and μ_2 are still indistinguishable, with the "Red" player having a slight advantage over the "Blue" player. However the advantageous player oscillates and alternates $\sim$ every 25 plays. Hence we need to increase the board size to investigate further.

4.1.2 Board size 4

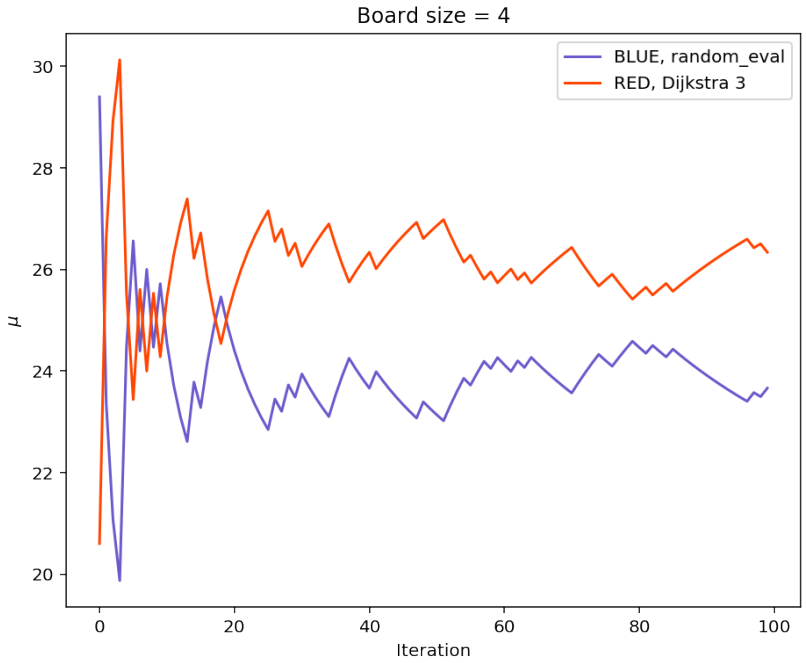

Figure 8: Allow 100 plays to see the ratings for a board of size 4.

The total run time for 100 plays is 28.21 s, with an average of 0.28 s. Now it can be seen that after some 20 iterations the μ values become divergent. A higher value in μ stands for better performance. Hence with the increasing number of plays we can see that the player with Dijkstra's algorithm has a better evaluated performance hence better chance to win, which is what we expected. Yet at the same time it can also be seen that Dijkstra with depth 3 does not have a significant advantage over the random player. What we need to do is increasing the board size and at the same time, increasing the depth visited.

4.2 Dijkstra depth 3 vs. Dijkstra depth 4

Again we start with a board of size 3. The "Blue" player is the one with depth 3 Dijkstra and the "Red" the depth 4 Dijkstra. We let the "Blue" player to start.

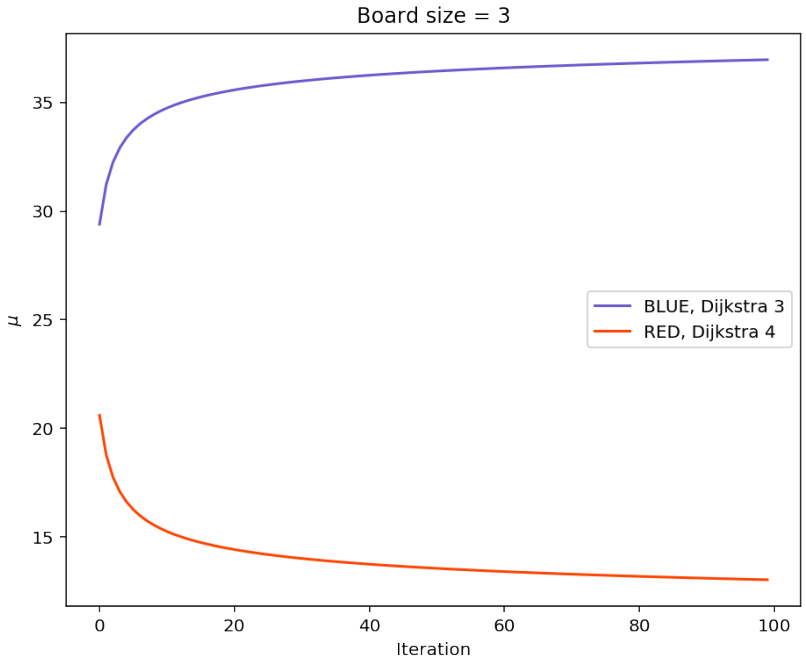

Figure 9: Allow 10 plays to see the ratings for a board of size 3.

The total run time for 100 ratings is 4.79 s, with an average of ~ 0.048 s per rating. Surprisingly for a small board size (3) we can see that the advantageous player is now the starter. For a game with small board size, increasing the evaluation depth does not necessarily improve our chances of winning as expected. Interestingly, the starter always wins (100 out of 100). To see the effect of beginner's advantage we now switch the players.

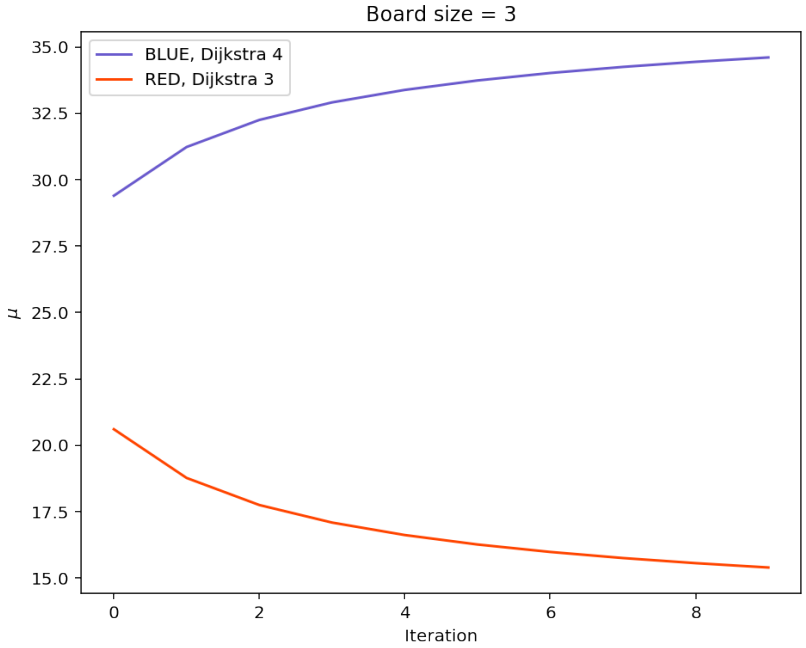

Figure 10: Allow 100 plays to see the ratings for a board of size 3.

The total time for 10 ratings is now 0.59 s, with an average time of 0.06 s per game. It can be seen now that by switching the sequence of playing, now the player with depth 4 Dijkstra becomes the advantageous player. And again, the starter always wins (10 out of 10). This outcome implies

that in a game with small board size, the starting player has a better chance of winning, and this advantage can not be easily beaten by increasing the search depth. Below shows the outcome after increasing the board to a size of 7.

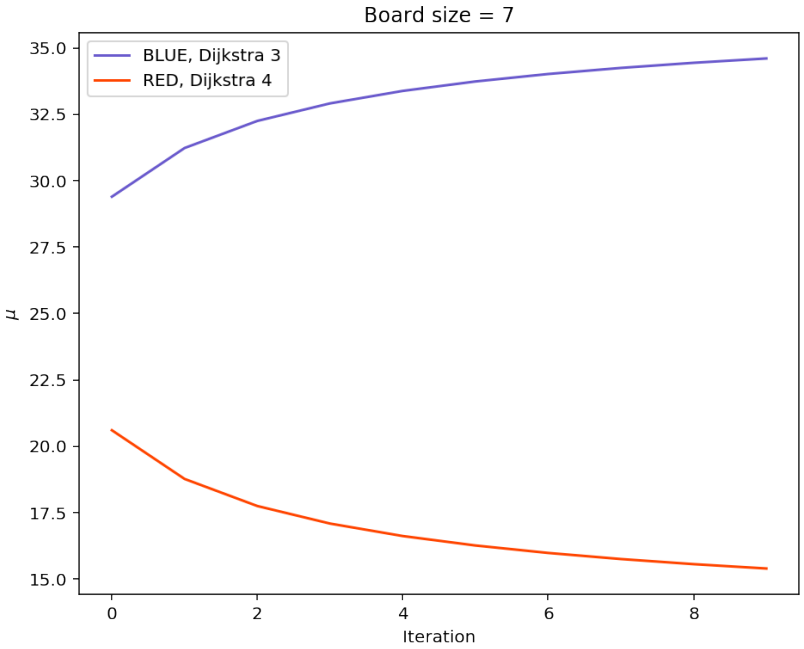

Figure 11: Allow 10 plays to see the ratings for a board of size 7.

The total run time for 10 plays for a board of size 7 is 317.53 s, with an average of 32 s. To test when this advantage breaks down here is another following experiment. Now we let the "Blue" player to be the one with depth 1 Dijkstra and "Red" with depth 4 Dijkstra, and allow the "Blue" player to start.

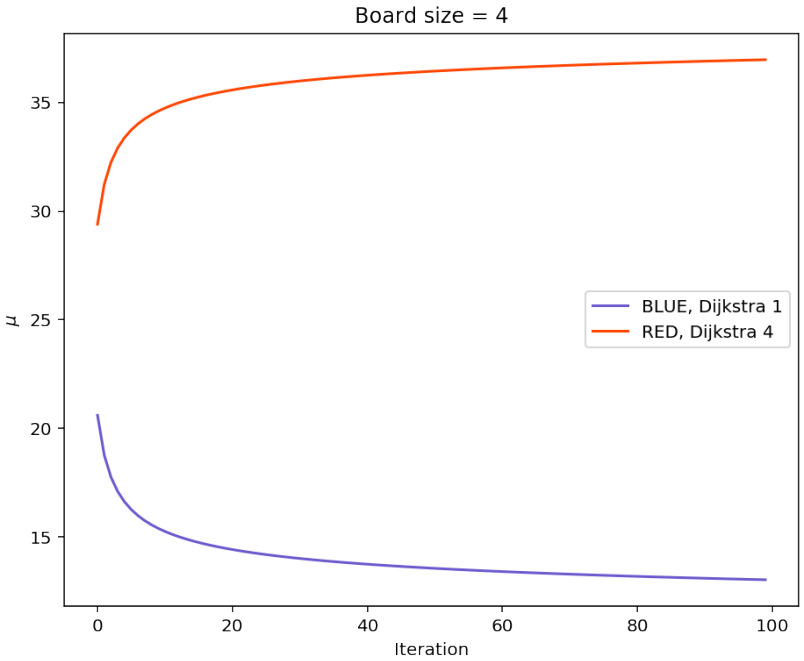

Figure 12: Allow 100 plays to see the ratings for a board of size 4.

It can be learnt from the above outcome that only when the depth is significantly larger than the other and when the board size is large (at least 4), the second player has an advantage.

4.3 Random depth 3 vs. Dijkstra depth 4

Now we start with a board of size 5 and allow the "Blue" player to be the one using random evaluation and the "Red" player the one using depth 4 Dijkstra. We again let the "Blue" player to start. The following shows the result for 100 plays.

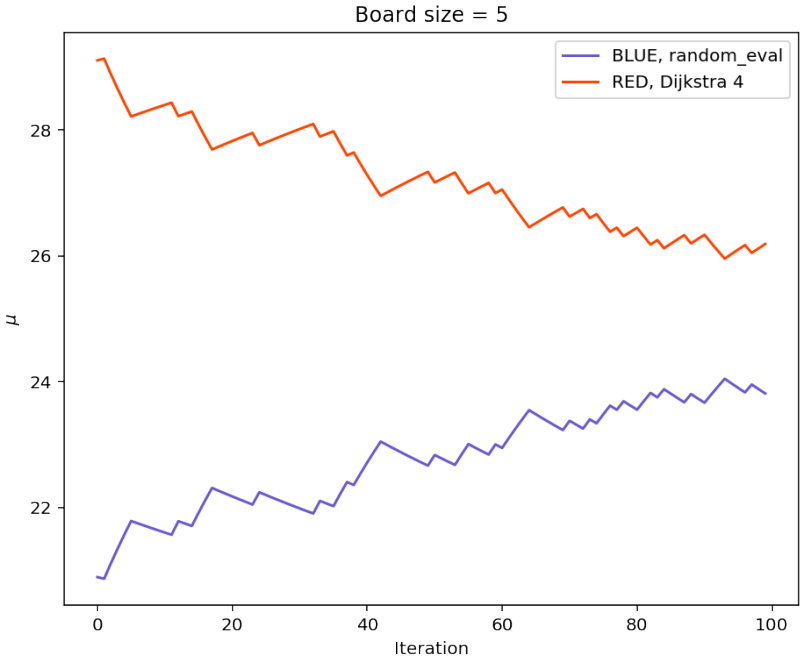

Figure 13: Allow 100 plays to see the ratings for a board of size 5.

It can be seen now the "Red" player proves to have an advantage over the "Blue" player, which finally beats the beginner's luck.

5 Iterative Deepening and Transposition Tables

Iterative deepening is an enhancement method which allows enhancement to the search algorithm. It provides an "any-time" algorithm instead of a fixed depth, so that the algorithm can go to the biggest depth possible in the allowed time. Transposition tables can be used as a way to efficiently search from one node to another, by preventing visiting duplicated board states. Together the two enhancement technique can improve the performance of the "Alpha-beta" algorithm.

5.1 Dijkstra vs. Dijkstra

We start again with a board size of 4.

```
"""Now change the board to size = 3. If player 1 (BLUE, Dijk 3) wins, return 0; else return 1 """
timedelta = 60
def play(size=3):
    board = HexBoard(size)
    p1 = Search(board, player='BLUE',timedelta=timedelta)
    p2 = Search(board, player='RED',timedelta=timedelta)
    while True:
        _, score, bm = p1(method='TTab',depth=3)
        p1.play(bm)
        #print(board)
        if board.game_over:
            break
        _, score, bm = p2(method='TTab',depth=4)
        p2.play(bm)
        #print(board)
        if board.game_over:
            break
    if board.check_win(board.RED):
        return 1
    elif board.check_win(board.BLUE):
        return 0
    else:
        return None
```

Figure 14: Redefining the play function to allow search enhancements.

Now the method "alphabeta" is replaced by "TTab" (which is shorthand notation for Transposition table + Iterative Deepening.) We limit the search time to 60 s and do the experiments in Section 4 again. Notice that now the constraints $depth = 3$ and $depth = 4$ do not apply anymore since the iterative deepening automatically goes to the highest depth possible.

```
r1, r2 = Rating(), Rating()
data = []
num = 10
mu1 = np.zeros(num)
mu2 = np.zeros(num)
start = time.time()
for i in range(num):
    rank = play(4)
    print("%d: winner: "%i, rank)
    if rank == 0:
        r1, r2 = rate_1vs1(r1,r2)
        print(r1, r2, '{:.1%} chance to draw'.format(quality_1vs1(r1,r2)))
        mu1[i] = r1.mu; mu2[i] = r2.mu

    if rank == 1:
        r2, r1 = rate_1vs1(r2,r1)
        print(r1, r2, '{:.1%} chance to draw'.format(quality_1vs1(r2,r1)))
        mu1[i] = r1.mu; mu2[i] = r2.mu

end = time.time()
print("Run time for 10 play with a board of size 4: ",np.round(end-start,2),"s.")
```

Figure 15: Rewrite the ratings for 10 games.

The total run time for 10 games is 4.16 s, with an average of 0.4 s per rating.

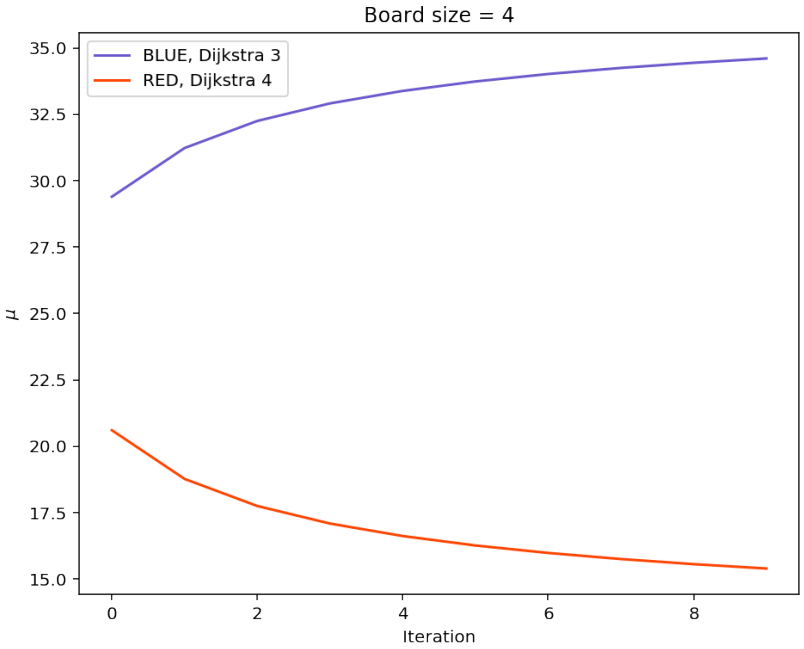

Figure 16: Rating for 10 games.

In this round again, the first player has beginner's luck. It can be noticed under this circumstances it is actually both players with Dijkstra's algorithm which can be stopped at any depth within the time allowed, playing against each other. Not surprisingly the first player will always win. Next we do the experiment again with a player using random evaluation and a player using Dijkstra's algorithm.

5.2 Random vs. Dijkstra

We start with a board size of 4 and again let the random evaluation player be "Blue" and be the first player, and set the time limit to be 60 s. The outcome is analysed based on the results of 50 plays.

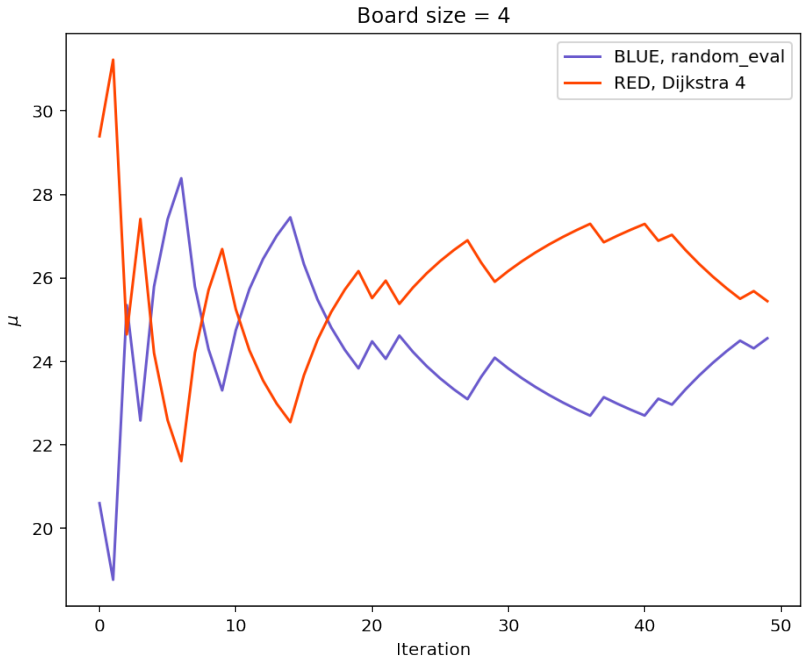

Figure 17: Rating for 50 games with "Blue" being the first player with random evaluation.

The total run time for 50 games is 4.54 s with an average of 0.09 s per rating. It can be seen that in the very beginning the "Blue" player has a slight advantage over the "Red" player, but this soon broke down after some 20 plays. Afterwards the "Red" player has a non-negligible advantage which overshadows the beginner's luck. Compared with the outcome of random player vs. Dijkstra depth 3 in section 4.1.2 it can be seen the efficiency is greatly improved without decreasing the chances of winning (in both cases the advantage of the Dijkstra player showed after $\sim$ 20 plays).

6 Conclusion

It can be learnt from section 4.1.1 to 4.1.2, when increasing the board size from 3 to 4, the average time per play is increased from $\sim$ 0.03s to $\sim$ 0.3s. In cases where the board size is small (< 7), the beginner's advantage is very hard to beat even when the player is moving randomly. And very surprisingly when two players with both Dijkstra's algorithm but different search depth playing against each other, the winner is always the beginner. However, with an increase in the board size as well as the search depth, this advantage is suppressed and ultimately beaten. On the other hand, when adding the transposition table and the iterative deepening and analysing the performance in a limited time, the beginner's advantage is suppressed greatly in the very first beginning. In the case of self-playing alpha-beta players with different depth sizes, we found that when the difference in depth size is small, the dominating factor on the outcome is actually the sequence of playing (which is proven by switching the sequence of players). The beginner's advantage is suppressed only when the board size gets larger and the search depth difference is greater than 3 (see cases with depth = 1 and depth = 4).

All in all, the algorithm works as expected, despite the largely underestimated 'No Pie Rule'. It can be learnt that if I were to design a game, I would increase the board size and at the same time, limit the time for the player to react. And if I were the human player in a board game, I would choose the one with lowest possible board size to increase my chance of winning, even if it is missing the whole gist of the game.