Build a Game-playing Agent

In this project I implemented the minimax and alpha-beta algorithm and a custom heuristic score to develop an AI agent that plays the game Isolation (using a move-set like the knight in chess) and ideally outperforms the agent using the given benchmark heuristic.

Heuristics

To test the functionality of the minimax and alpha-beta algorithm, I started by implementing an extremely simple heuristic, the **#moves** available to the player. Since this heuristic points in the right direction but is overly simplified (e.g. it completely ignores the opponent), I initially modified it to take the #moves available to the opponent into account (**#moves - #opponent_moves**). This way the algorithms maximise the number of moves available to the player and minimise the moves available to the opponent. I tried to modify this further and put a stronger emphasis on restricting the opponents ability to move, leading me to the first of my final set of implemented heuristic:

Heuristic1 = #player_moves - 2 * #opponent_moves

```
player_moves =  len(game.get_legal_moves(player)) / (game.width * game.height)
opp_moves = len(game.get_legal_moves(game.get_opponent(player))) / (game.width * game.height)
heuristic1 = player_moves - 2 * opp_moves
```

Next, I tried to penalise certain areas of the board to achieve the same goal of maximising the freedom of movement of the player. The positions on the board where the movement is restricted the most (or where it is easiest to restrict the movement in the future) is close to the border of the board, and close to the opponent. This lead me to the next two heuristics that will be used in the final set:

Heuristic2 = player_location - opponent_location

```
heuristic2 = sum([abs(a-b) for a,b in zip(game.get_player_location(player), game.get_player_location(game
```

Heuristic3 = player_location - board_center

```
heuristic3 = sum([abs(a-b) for a,b in zip(game.get_player_location(player), ((game.height-1)/2, (game.wid
```

As can be seen, I normalised all of these heuristics by the size of the board. This scales them to a similar range and generally makes it easier to choose and vary the individual weights when combining different heuristics.

As a next step, I looked at a more complicated heuristic that is computationally more demanding but should be more accurate. It is not feasible to use this throughout the whole game, as it would restrict the search depth in the early stages of the game too much. I ended up using this late-game heuristics when less than 3/4 of the game board is still open.

The first part of this heuristic is to check if the game board is divided into two parts. To keep it easy to compute, I decided to restrict this to a simple check: Are there two adjacent rows or columns that are completely blocked and are the two players on opposite sides of the wall?

```
# check if two adjacent rows or columns are completely blocked off
    rows, cols = zip(*blanks)
    rowcount = [rows.count(x) for x in range(game.height)]
    colscount = [cols.count(x) for x in range(game.width)]
    try:
        hwall = [sum(x) for x in zip(rowcount[:-1], rowcount[1:])].index(0)
    except:
        hwall = False
    try:
        vwall = [sum(x) for x in zip(colscount[:-1], colscount[1:])].index(0)
    except:
        vwall = False
    # if there is a wall, check if players are on opposite sides of it (not on it)
    if (hwall and ((playerpos[0] < hwall < opppos[0]-1) or (opppos[0] < hwall < playerpos[0]-1))) or (vwal
```

If this condition is fulfilled, it is safe to assume, that the players cannot interfere with each other. Because of this, the player with the longest chain of moves available, will win the game.

```
def get_chain_length(game, player, time_left, TIMER_THRESHOLD):
    max_moves = 0
    if time_left() < TIMER_THRESHOLD:
        raise Timeout()
    legal_moves = game.get_legal_moves(player)
    if not legal_moves: return 1
    for move in legal_moves:
        nmoves = 1
        new_game = game.forecast_move(move)
        new_game.__active_player__ = player
        nmoves += get_chain_length(new_game, player, time_left, TIMER_THRESHOLD)
        if nmoves > max_moves: max_moves = nmoves
    return max_moves
```

Heuristic4 = length_move_chain

Tournament test

I ran a few short tournaments to find a combination of heuristics that seem to give encouraging results. Using the heuristics above, I used the following combinations for the three short (num_matches = 5) and two long tournaments (num_matches = 100):

- Just Heuristic1
- (Heuristic1) and Heuristic4 (after 1/4 of the board has been filled)
- (Heuristic1 + Heuristic2) and Heuristic4 (after 1/4 of the board has been filled)
- (Heuristic1 - Heuristic3) and Heuristic4 (after 1/4 of the board has been filled)

These are the results:

Heuristic \ #matches	5	5	5	100	100	Combined
Improved_ID	72.86%	70.00%	77.14%	76.50%	76.04%	76.07%
Heuristic1	75.00%	75.71%	75.71%	74.89%	76.00%	75.45%
(Heuristic1) and Heuristic4	71.43%	80.71%	77.14%	76.07%	76.89%	76.48%
(Heuristic1 + Heuristic2) and Heuristic4	77.86%	72.86%	72.14%	75.75%	77.43%	76.43%
(Heuristic1 - Heuristic3) and Heuristic4	78.57%	75.71%	76.43%	75.39%	74.46%	75.06%

It looks like contradictory to the results from the initial short tournaments, putting a stronger emphasis on restricting the opponents movement does not improve the results compared to Improved_ID. Adding the late-game heuristic does improve the results, which is as expected. In later stages of the game, the board movement is more restricted and the extra computational effort does not reduce the maximum search depth as much as it would've done in early stages of the game. Adding the extra constraints about board positioning does not seem to change the results much and even has a slight negative effect.

Based on these result, I ran another tournament (num_matches = 100) but modified the simple heuristic, so that it is identical to the Improved_ID.

Heuristic \ #matches	100
Improved_ID	74.68%
(#moves_player - #moves_opp) and Heuristic4	74.46%
(#moves_player - 2 * #moves_opp) and Heuristic4	76.00%

These results are interesting. Previous tests suggested that putting more emphasis on restricting the opponents movement does not improve the results when only the simple heuristic is used. When combined with the late game heuristic, it does seem to have a positive effect though. Restricting the movement of the opponent early on seems to be beneficial for putting the opponent into a situation where he will end up on the smaller side of a wall. The average search depth reached using the Improved_ID heuristic is ~7.1, which drops to ~6.7 when any of the other two are being used. It seems like the heuristics all have a similar performance, where a shallower search depth is balanced

by a more accurate heuristic. The length of the tournament might not be enough to give a reliable order of the heuristics though.

The heuristic of my choice is **(#moves_player - 2 * #moves_opp)** throughout the game combined with **length_move_chain** in the later stages of the game. It slightly outperforms the Improved_ID heuristic in most tests and delivers a fairly consistent performance.

Additional Sources:

https://en.wikipedia.org/wiki/Minimax https://en.wikipedia.org/wiki/Alpha–beta_pruning