- *Briefly describe the structure of your game-playing program. Give an overview of the major modules and/or classes you have created and used.*

The structure of our game is relatively similar to the one we submitted for Project A. The only module we are using is *numpy* and we utilize only one of its functions to make the code neater.

For this project we have decided to apply a different kind of algorithm — Best-Reply Search (BRS) which should count as a creative technique because it is out of this course's scope. Our main source of information was a paper called "Best-Reply Search for Multi-Player Games" by Maarten P.D. Schadd and Mark H.M. Winands[1]. In general, the structure of the program is as following: the program updates the gamestate with provided values and the gamestate to BRS. BRS searches for possible paths — while pruning those that it knows that are not worth considering — and picks the move that would lead to the path with highest value. Then with each turn, this cycle is repeated.

- *Describe the approach your game-playing program uses for deciding on which actions to take. Comment on your search strategy, including on how you have handled the 3-player nature of the game. Explain your evaluation function and its features, including their strategic motivations.*

When it comes to deciding what actions to take, the algorithm considers the weight of each possible path. The value of each path is a sum of values of nodes that compose that path. Since our optimal (by optimal we mean depth deep enough to be able to look ahead, but also not requiring too much time in order to squeeze in the time constraint) depth of the algorithm is four, then that means that the algorithm considers two of its own turns and two opponents turns. That allows the program to lookahead and take into consideration opponent's moves.

The evaluate() function is in charge of determining the value of each node, which in the long run will impact our player's decision which move to pick. It contains a dictionary of variables, which define weights to reflect each variable's importance. The variables are: 'distance' (average distance of all pieces to the exit), 'score' (the score ensures that if our player has exited, then that path would receive additional value for the exit), 'pieces_on_board' the number of our pieces on the board), 'turn' (the turn which determines whether if any of the piece's on the exit's edge should exit now or later), and 'winning' (which is the variable that shows that this certain move would lead to winning the whole game).

We included alpha-beta pruning[2] in this method to increase efficiency and filter out all moves that are not worth of being considered. Moreover, whenever BRS considers' opponents turn (at a MIN node), it only picks the player that has the strongest counter move and does not even consider the other player. This saves on computation time and space, and also explains how we have approached the 3-player nature of the game. As you can see, BRS is not a fully paranoid

[1] Link: https://dke.maastrichtuniversity.nl/m.winands/documents/BestReplySearch.pdf
[2] Code primarily based on this pseudo code: https://en.wikipedia.org/wiki/Alpha–beta_pruning

algorithm, but it is rather just a cautious one — it does not consider every single opponent's move, but just chooses the one that is most threatening.

To further save on time, we implemented the opening book because we have noticed that the first moves really do not impact the efficiency of our algorithm. We tested the algorithm with and without the opening book and the success rate was not influenced. Moreover, it saves on time which means that we can commit more time to the later part of the game, where each decision impacts the final outcome of the game.

- *If you have applied machine learning, discuss the learning methodology you followed for training and the intuition behind using that specific technique.*

No machine learning was implemented.

- *Comment on the overall effectiveness of your game-playing program. If you have created multiple game-playing programs using different techniques, compare their relative effectiveness.*

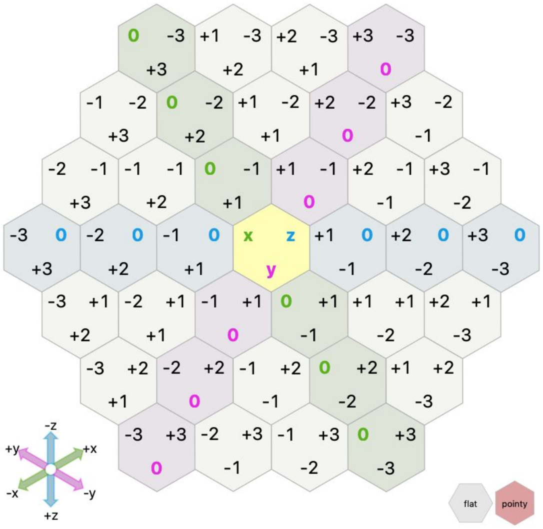
To check our algorithm's effectiveness we have played it against a greedy algorithm (written by us, which is the same as BRS but just doesn't look ahead, which means that depth is equal to one) and a random algorithm (that picks random moves). We have noticed that BRS usually outperforms both greedy and random algorithms. We also tested our algorithm on the battleground and noticed that the success rate is acceptable.

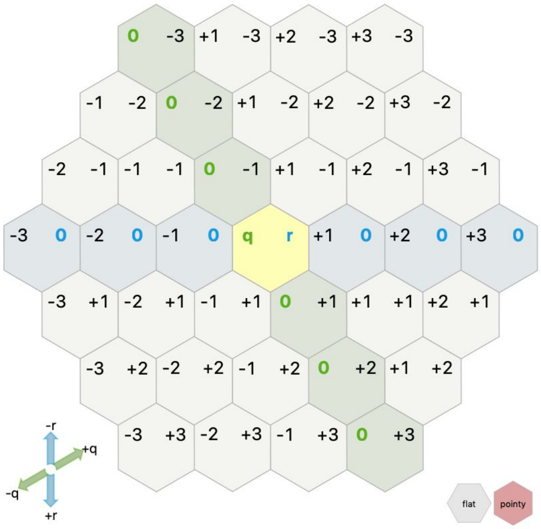
- *Include a discussion of any particularly creative techniques you have applied (such as evaluation function design, search strategy optimisations, specialised data structures, other optimisations, or any search algorithms not discussed in lectures) or any other creative aspects of your solution, or any additional comments you wish to be considered by the markers.*

Beside BRS being out of scope, another thing that we have used to optimize the algorithm is to use 3-value coordinates, instead of 2-value coordinates. The two pictures below visualize the 3-value coordinates (on the left) and 2-value coordinates (on the right)[3]. Specialized functions (such as axial_to_cubic())have been created to convert the coordinates from 2-values to 3-values and vice versa.

Increasing number of values of coordinates allowed us for easier computation of the distance. We have noticed that each edge of the game's board has something unique — one of the coordinates will be the same for all tiles that make up that edge. For example, if we are the blue players, our possible exits reside on the top edge of the game's board. That edge has something unique to that edge — all of its y values are equal to 3. That means that in order to search for distance towards our exit, we have to consider only the z coordinates.

[3] Pictures and the idea is based on: https://www.redblobgames.com/grids/hexagons/#coordinates


0 -3 +3
+1 -3 +2
+2 -3 +1
+3 -3 0
-1 -2 +3
0 -2 +2
+1 -2 +1
+2 -2 0
+3 -2 -1
-2 -1 +3
-1 -1 +2
0 -1 +1
+1 -1 0
+2 -1 -1
+3 -1 -2
-3 0 +3
-2 0 +2
-1 0 +1
x z y
+1 0 -1
+2 0 -2
+3 0 -3
-3 +1 +2
-2 +1 +1
-1 +1 0
0 +1 -1
+1 +1 -2
+2 +1 -3
-3 +2 +1
-2 +2 0
-1 +2 -1
0 +2 -2
+1 +2 -3
-3 +3 0
-2 +3 -1
-1 +3 -2
0 +3 -3
-z
+y
+x
-x
-y
+z
flat
pointy


0 -3
+1 -3
+2 -3
+3 -3
-1 -2
0 -2
+1 -2
+2 -2
+3 -2
-2 -1
-1 -1
0 -1
+1 -1
+2 -1
+3 -1
-3 0
-2 0
-1 0
q r
+1 0
+2 0
+3 0
-3 +1
-2 +1
-1 +1
0 +1
+1 +1
+2 +1
-3 +2
-2 +2
-1 +2
0 +2
+1 +2
-3 +3
-2 +3
-1 +3
0 +3
-r
+q
-q
+r
flat
pointy