1010! Game Agents

Greedy Best-First and Adversarial Search Approaches

Aviv Cohen — aviv.cohen2@mail.huji.ac.il

Ilia Bezgin — ilia.bezgin@mail.huji.ac.il

Ronel Charedim — ronel.charedim@mail.huji.ac.il

Netanel Tuito — netanel.tuito@mail.huji.ac.il

Introduction to artificial intelligence – Final Project

The Hebrew University of Jerusalem

Jerusalem, Israel

***Abstract* — 1010! is a single player static Tetris-like block puzzle game. Its main goal is to achieve the highest score possible by fitting pieces into the grid and completing vertical or horizontal lines to clear blocks. This paper explores two groups of agents playing the game. One using informed search and the other using adversarial search.**

I. INTRODUCTION

A. *The game*

1010! is a single player Tetris-like puzzle, but without the time constraint and with more possibilities to place the pieces on the board. The game board is a 10x10 grid (Fig. 1) and in each round of the game the player is given 3 blocks randomly chosen out of the set of 19 pieces. The player can place the blocks on the board wherever they fit and in the order he/she chooses but must place all of them in order to get the next 3 blocks. When an entire row/column is filled by blocks, the line is cleared, and therefore the game could be infinite in theory. The game ends when none of the given blocks can be placed on the board.

Fig. 1. The 1010! game mobile version interface

B. *Scoring*

The goal in the game is to achieve a high score by placing as many blocks on the board as possible. The scoring works as follows:

1. For every block the player is placing on the board, he/she gets points equal to the block size.
2. For each row/column being cleared from the board, the player would get basic 10 points, but when several lines are cleared by one action (define this number to be n), then the player receives $0.5 \cdot (n + 1) \cdot 10$ points.

If you want to plunge into the atmosphere of the game and try to play it, here is a link for a browser version of the 1010! game.

Our goal in this project is to build an agent that will play 1010! game the best way possible. Achieving the goal is measured by getting the highest score, of course it requires improving the playing method as much as possible and maintaining the game for as many rounds as possible. In addition, we would like to get the best results in terms of running time (number of pieces being placed in a minute).

II. MOTIVATION

We decided to create agents for solving the game in the first place because we enjoy playing 1010! and it was a nice challenge to develop a computer program that is more successful than us.

We think that the problem itself is a search problem since for every game state there are several possible successors (game state given after performing an action - a placement of a selected piece on a specific location on the board). There is no finite maximal score which makes this problem an infinite search problem.

In each round of the game, we are given only three pieces that need to be placed, so there is partial observability during the game process. In addition, during each round there are

$\sim 100^3 \cdot 3! = 6{,}000{,}000$ possible assignments of the three pieces (3! ways to choose the order of the pieces, and about 100 possible locations for assignment on the board for each piece). There could be thousands of rounds in an average game so the problem is computationally complex. For the reasons above, the problem could not be solved using naive approaches like DFS, that is why we had to find more sophisticated ways for solving the problem.

III. APPROACH AND METHODS

First of all, we had to work with a python implemented 1010 game. We found an implementation of the game on the internet (link to the git repository with the primary version of the game) but there were some issues with it. There were a lot of errors in the game itself and there was only a manual way to play the game using GUI, so we fixed them and added extra features allowing the agents to interact with the game without a human player and optionally without GUI (for better performance) (Fig. 2).

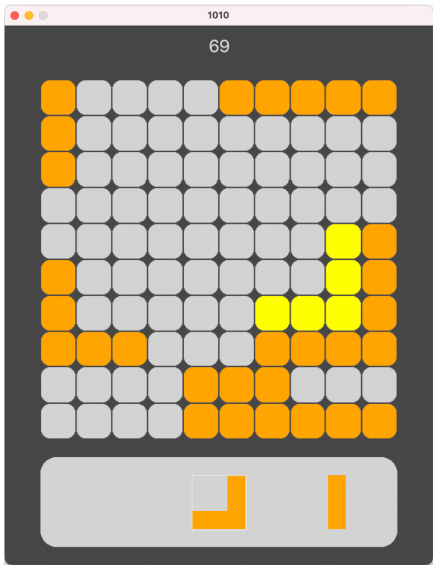

Fig. 2. The 1010! game python version interface

We decided to use two main approaches for solving the problem. The first approach was informed search and the second one was adversarial search. Using informed search, the main idea is to perform heuristic search in subspace of a single game round (placing three pieces), while adversarial search tries to predict the game states in future rounds and builds strategy accordingly. The core of both approaches is a quality set of heuristics and we tried to create a good one.

A. Heuristics

All of our search algorithms are using heuristics in order to determine which move is better in a certain point of the game. We had to find a way to compare different boards and actions, so we scored a board according to different categories which enabled us to decide which board state is better. We created two sets of heuristics; first is relying on game state only (after applying an action), second is relying on current game state and a given action.

In addition, we used null heuristic as a baseline to compare all the heuristics, in order to determine whether a certain heuristic contributes some useful information.

Game state only based heuristics:

- "*One square*" heuristic - a board is considered optimal if it contains as few "holes" as possible. A hole is an empty coordinate of the board, surrounded by blocks in all adjacent coordinates. Hence, when grading the board state, we gave better scoring for a board with least holes. The score given by this heuristic is the negative value of the number of holes in the board.
- "*Empty large square*" heuristic - we always tried to make sure there is enough space on the board for placing the largest piece (3x3 square). Most of the time, the game ends because the player gets a 3x3 square and it is not possible to place it on the board. Therefore, when scoring the board, a board with no space for the largest block would get a very low score (-100,000).
- "*Free space*" heuristic - we want to give a higher score to boards with as least occupied coordinates as possible, as it allows the player higher chances to place more blocks in the following rounds of the game. Hence, the heuristic returns the number of vacant coordinates.

Game state + given action heuristics:

- "*Surface*" heuristic - we prefer placing pieces in adjacent coordinates to existing blocks' locations or to board's boundaries. We increase the surface area between the current piece and the existing blocks on the board, which leads to decreasing the total surface area occupied by blocks and creating compact chunks of occupied cells. The heuristic returns the number of occupied or boundaries adjacent cells to the given piece.
- "*Row col completeness*" heuristic - another way of evaluating given action is checking whether given it brings a row/column closer to completeness. As the row/column is more complete, the faster it will vanish from the board. The heuristic returns a value according to the following formula: $\frac{\sum_{i=1}^{10} r_i^2 + l_i^2}{\sum_{i=1}^{10} r_i + l_i}$, while $\forall 1 \leq i \leq 10$ r_i, l_i is the number of filled cells in row i or column i accordingly.

In addition, we created another heuristic - "*combined*" heuristic, that is actually a combination of all heuristics

described above. It returns weighted sum of other heuristics' values with the following weights:

- $w_{one\ square} = 50$
- $w_{empty\ large\ square} = 1$
- $w_{free\ space} = 1$
- $w_{surface} = 2.5$
- $w_{row\ col\ completeness} = 0.5$

The optimal weights could have been calculated using computational tools, but as the time did not allow, we chose them manually. We examined the average range of values returned by each heuristic, that allowed us to make a reasonable weights' choice:

- "*One square*" heuristic: [0, 5]
- "*Empty large square*" heuristic: {0, -100,000}
- "*Free space*" heuristic: [50, 80]
- "*Surface*" heuristic: [0, 12]
- "*Row col completeness*" heuristic: [5, 12]

Empty large square by itself returns a very low value when there is not enough space for the largest piece (very unlikely state), so there is no need to give it an extra weight. *Free space* was the first heuristic we worked with, so it received weight 1, and the other heuristics were weighted relatively to it. *One square* heuristic received high weight, since it is very hard to fill holes and we highly enforce the agents to avoid them. The difference between minimal and maximal values of *surface* heuristic is 12, and we wanted to bring closer the significance of this heuristic to *free space* that has a 30 points range so we gave to *surface* a weight of 2.5. Empirically, we conclude that *row col completeness* indeed contributes to get better results, but only when its weight is relatively small (0.5).

B. Greedy best-first Search (Greedy BFS)

First, we wanted to start from a basic algorithm that will be our baseline for further algorithms. We decided to implement a simple version of a greedy search algorithm. This version will be called *greedy best-first search single*. It works by the following steps:

1. Pick the smallest piece from the given 3 pieces in the round of the game.
2. Remove the piece from the set of given pieces.
3. Iterate over all possible assignments of the piece on the board and evaluate the heuristic score of the board's state.
4. Apply the action that received the highest heuristic score.
5. Repeat steps 1-4 until the game is over.

The algorithm takes into account information of the assignments of a single piece and ignores the information derived from possible assignments of other pieces. Therefore, we mainly expect this solution to be not accurate but fast enough.

Second, we created a more complex version of a greedy search algorithm. This version will be called *greedy best-first search triple*, as it simultaneously takes into account information about possible assignments of all three pieces. It works similarly to *A* search* algorithm, but the main difference is that the evaluation function of *A** is $f(s) = g(s) + h(s)$, whereas the evaluation function of *greedy BFS triple* is $f(s) = h(s)$ (s is the state of the board, f is the evaluation function, g is the cost function and h is the heuristic function). This modification leads the solution to be not optimal but reasonable and relatively fast. The solution could even be considered successful, especially if the heuristic function is good enough.

C. Alpha-Beta pruning

In this approach we considered the game as a zero-sum game. There are two players - our agent and the board. Our agent is trying to maximize the score while the board is trying to minimize it. The Max player is supposed to place the three given pieces in the optimal order, while the Min player is reacting by giving the worst pieces so that the other player will lose. Using the *alpha-beta* algorithm, the agent is trying to predict future actions of the board, and is reacting accordingly.

First, we tried to implement the basic version of *alpha-beta* algorithm, but quickly we found out that the search tree is too large to be iterated over. We defined a node in a search tree to be a solution for one round (placement of 3 pieces). Empirically we calculated that on average, there are about 1,800,000 possible assignments of 3 pieces and the algorithm should go over them for each node. We reach such a huge number of possibilities already for depth 0, while for deeper levels the number will be much bigger. The first time the algorithm reaches beta player (depth 1), there are $\binom{19+3-1}{19-1} = 1330$possible ways to choose a triplet of pieces and in order to choose the "worst" triplet (that will lead to the lowest score), each triplet should be evaluated by the same number of iterations like an alpha node. To sum up, there are $1,800,000^2 \cdot 1330 \approx 4.3 \cdot 10^{15}$states that the algorithm should go over working with depth 1. In order to decrease this terrible running time, we turned to the idea of extra search tree pruning. The upgraded version 2.0 of algorithm was implemented in the following way:

Each node of the alpha player is defined to be all possible legal actions for any single piece out of the given triplet (primary actions). The next two actions are calculated using a helper algorithm (*greedy best-first search*) that achieves a local maxima for each primary action and then, the alpha player chooses the best one. Such behavior should decrease the search tree size for depth 0. In addition, we also reduced a beta player level by sampling only 10 triplets out of 1330 and by evaluating each of them using the *greedy best-first search* algorithm with reversed heuristic (because we are looking for a local minima). Using this method we eventually succeed to get some results for depths 0 and 1 with a long running time, but at least finite.

In an attempt to minimize the running time even more, version 3.0 of *alpha-beta* was implemented. The main difference from the previous method is that now, the alpha player first sorts the primary actions by their heuristic value and continues the process only with top 30 actions. The beta player acts without changes. Such a change is supposed to perform an extra cut of the search tree.

IV. RESULTS AND CONCLUSIONS

In order to evaluate and compare our different approaches, we ran hundreds of simulations. First, we compared the score achieved by each agent using a single heuristic at a time (for each agent 50 game runs per heuristic). In fact, in each heuristic we took in consideration an extra one - *empty large square*, which we found not to be so useful by itself but possibly effective when combined with another heuristic. The reason for that, is the fact that keeping free space for the largest piece doesn't affect the decision progress during the game until the board is almost full of the blocks (because the heuristic returns 0 score while there is enough free space on the board). Therefore, most of the time the "*Empty large square*" heuristic works like a *null* heuristic.

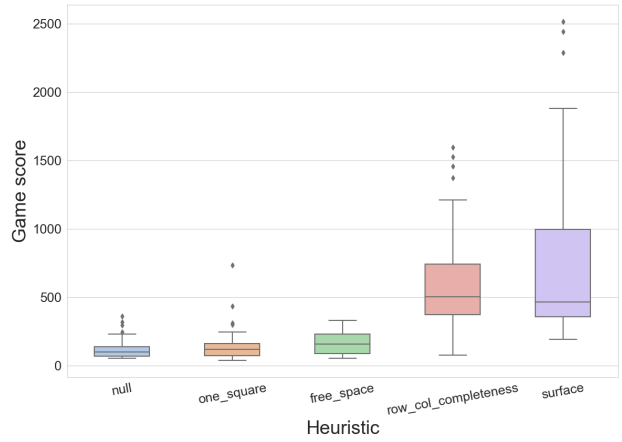

Fig. 3. Greedy best-first search single agent score using single heuristic

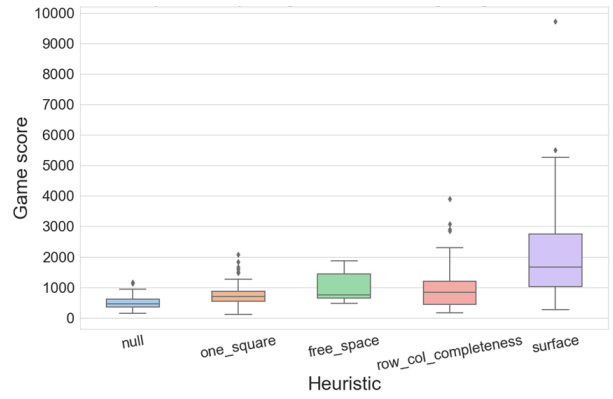

Fig. 4. Greedy best-first search triple agent score using single heuristic

As we can see, the ratio of scores between single heuristic is similar for both *greedy BFS single* (Fig. 3) and *greedy BFS triple* (Fig. 4) agents, but still, there are a few differences in the performance of the agents which will be explained below.

We can see improvement for *null* heuristic used by *greedy BFS triple* agent (Fig. 4) in comparison to *greedy BFS single* (Fig. 3). We think that it is related to the implementation of the algorithms - in the *triple agent*, the first three legal actions found for the given pieces will be returned. The agent goes over all board's coordinates in increasing order, that is why the positions returned are consequent, so the pieces will be located relatively close one to another. This will increase the probability of filling a row/column. As for the *single agent*, all possible assignments are scored by the heuristic and all scores are stored in a list. Eventually the location for a piece will be chosen using argmax function, and since the score is equal for all actions, the location is chosen randomly (depending on the implementation of argmax). That is why, the assignments of consequent pieces are not necessarily close one to another.

Similarly, *greedy BFS triple* agent (Fig. 4) achieved better results than *greedy BFS single* (Fig. 3) for *one square* heuristic. The reason for it is that most possible assignments do not create "holes" and that is why they receive the same zero heuristic score that leads to the *null* heuristic behavior. In fact, for both agents we can see slight advantage of *one square* heuristic relative to the *null* heuristic, possibly because of some corner cases where the first heuristic avoids "holes" that frequently ruin the game.

Regarding *free space* heuristic, *greedy BFS single* agent will receive the same score for almost all the assignments of a piece. Only if the piece completes a line, the assignment would get a better score, while all other cases are equivalent one to another. *Greedy BFS triple* searches for assignments of the three pieces simultaneously that allows it frequently to arrange them to a completed line that will disappear and lead to a higher heuristic score and more free space for future rounds. For these reasons, there is a score advantage in favor of *greedy BFS triple* (Fig. 4) in comparison to *greedy BFS single* (Fig. 3).

When using *row col completeness* heuristic, *greedy BFS triple* (Fig. 4) did better than *greedy BFS single* (Fig. 3). We think that the difference in this case is not related to the heuristic itself but to the peculiarity between these agents. The *single* agent will place first the smallest piece given and there is a possibility that it will not complete a line and there will not be enough space for the next larger pieces to be placed. While in the same game state, it is possible that a larger piece placed earlier could complete a line and free enough space for the rest of the pieces. In such a case, the *single* agent will lose the game earlier and achieve a lower score, while the *triple* has alternative possibilities for choosing the order of pieces placement so it will have better chances for continuing the game.

In comparison to other heuristics, *greedy BFS single* (Fig. 3) and *greedy BFS triple* (Fig. 4) got relatively good scores using *surface* heuristic. The heuristic creates compact chunks of occupied cells, which allows more freedom for further steps. According to the results, this is the most important parameter, since the most effective strategy is not striving for better score immediately but creating an infrastructure for next rounds.

All these heuristics in a separate use pale in comparison to the combined heuristic (Fig. 5, 6), which uses the strongest sides of each of them.

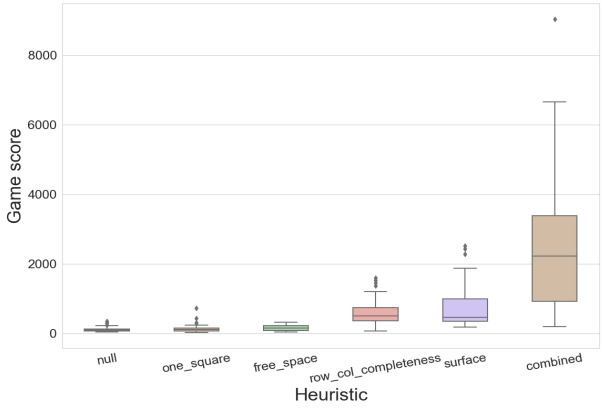

Fig. 5. Greedy best-first search single agent score using single and combined heuristic

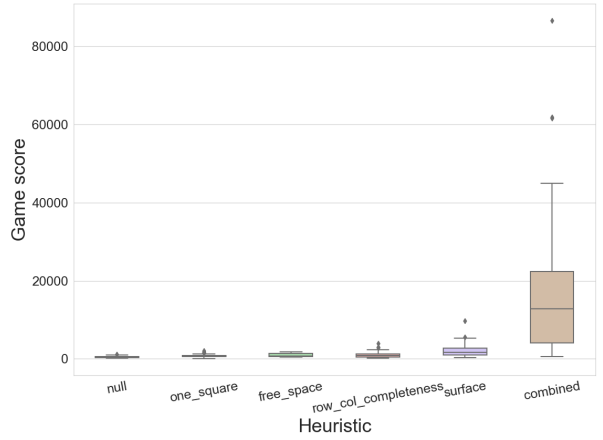

Fig. 6. Greedy best-first search triple agent score using single and combined heuristic

We already explained why it is not effective enough to use each single heuristic separately. Most of them have influence during different stages of the game, while in others they have no effect. That is why when we combine the heuristics we receive the outcome from each heuristic in its own finest hour, which leads to significant results (Fig. 5, 6).

As for the results of *alpha-beta* agents, we directly worked with our best heuristic (the combined one), since this group of algorithms was too slow and we wanted to increase their performance from the beginning. Like we mentioned earlier, the basic version of the algorithm was too ineffective so we did not even succeed to finish any run of it. In contrast, with version 2.0 of *alpha-beta* we managed to get some results. For depth 0 we expected results to be good at least as of *triple BFS* agent, since it searches for some local maxima, while *alpha-beta* tries to find a global one. With relatively slow runtime it achieved quite good results and even twice reached about 34500 points (Fig. 7). The next attempt was to increase the depth to be 1, but unfortunately it led simultaneously to significant runtime increasing and points decreasing (the best score was only 5500 with a running time of 5 hours). The reason for this phenomenon could be a poor sample (beta player works only with a 0.7% of the original search space); probably, there are not a lot of ways that may lead to a good score and this way we might cut most of them.

The main idea of using adversarial search was exploring the search space in deeper levels, but a massive running time bounded us. The version 3.0 was created specifically for bigger depths; it works faster but sacrifices a huge part of the search tree. All runs with depth 1 totally failed and were even worse than for version 2.0 with depth 1 (barely reaches 2000 points), but at least they were faster. Our hope was that it will achieve better results with depth 2, but again it was a fiasco. We performed several runs that each of them lasted for about 5 hours, but led only to a score lower than 1000 points. Again, the reason for it like in the previous case is a massive cut of the search tree. Probably the only way to get good results is going over the whole search tree or at least the lion share of it.

Only version 2.0 of *alpha-beta* agent with depth 0 got reasonable results, so for further comparisons with other agents when talking about *alpha-beta* we relate to this configuration of it.

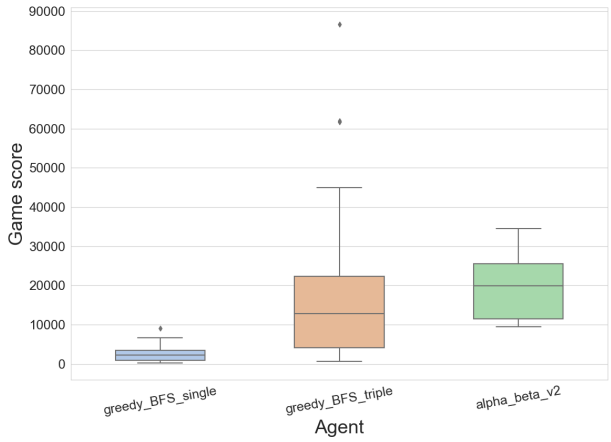

Fig. 7. Agents score using combined heuristic

Since we have seen that the combined heuristic is an absolute leader, we decided to compare the scores of our agents using it (Fig. 7). It is not surprising that *greedy BFS single* agent is an outsider because it operates with only partial information available to it. Both *greedy BFS triple* and *alpha beta* agents achieve higher scores. Indeed, *alpha beta* median score is the highest but it comes at the cost of running time (number of steps per minute), which is 30 times higher than of *greedy BFS triple* (Fig. 8).

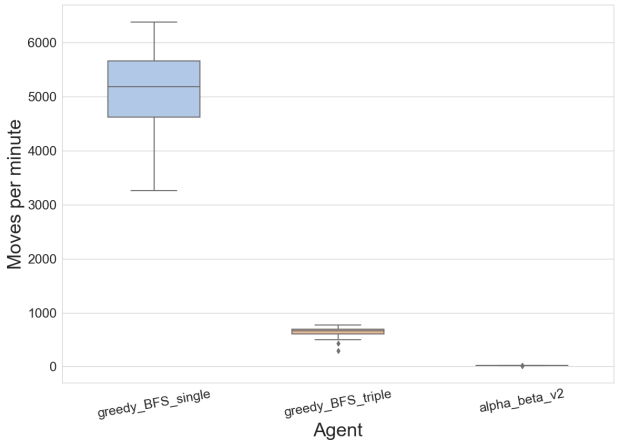

Fig. 8. The running time of different agent using combined heuristic

Greedy BFS single agent is checking placements for one piece (less than 100 possibilities), which makes it the fastest agent (Fig. 8). Next comes *greedy BFS triple,* it achieves a great score compared to its running time (only 5 times slower compared to the *single* agent). Last is *alpha beta* agent, that its speed is similar to a human player (very low), yet it is 200 times slower than *single* agent.

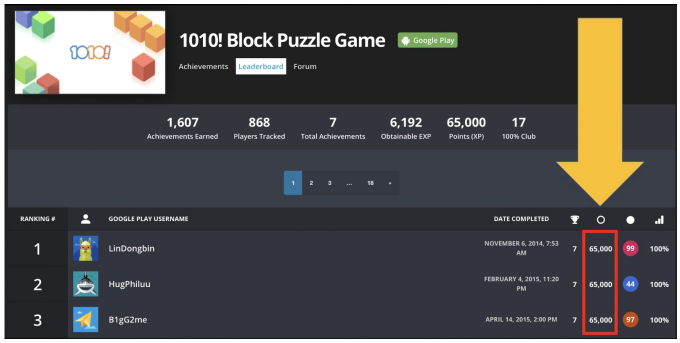

Fig. 9. A human players leaderboard top 3

We wanted to compare the results of our agents to human players, so we found a ranking table showing top android players' scores, in which we can see that top 3 players achieved 65,000 points (Fig. 9). In addition, we have found some sources that claim that there exist players who received more than a million points, but we suspect that they were actually advanced computer programs and not human players. This way, we can conclude that our *greedy BFS triple* agent who managed to get 86,625 points is successful enough, it beats most human players and definitely plays faster.

V. FURTHER WORK

As we mentioned before, it is possible to find optimal weights for the combination of the heuristics using automatic tools, e.g. genetic algorithms or linear regression. Future work can be also related to creating more sophisticated heuristics that will lead to a higher score. There are also some possibilities to upgrade *alpha beta* agent; we didn't succeed in creating a fast enough one that achieves good results with depth greater than 0, especially in a reasonable time. The expectimax algorithm could be tried for solving the problem, since the assumption that the board tries to minimize the score is not realistic enough (like we assumed for the *alpha beta* algorithm). Another approach that can be more effective in terms of running time is a local search; it possibly can get good results in a relatively small amount of time.

VI. CODE USAGE

The program can be runned by execution of the following command:

```
main.py [--agent={HumanAgent,
GreedyBFSSingleAgent, GreedyBFSTripleAgent,
AlphaBetaAgent}]
[--sleep] [--gui] [--repeat=REPEAT]
[--output=OUTPUT] [--version=VERSION]
[--depth=DEPTH]
```

--agent: Agent type (choose one of the given options). The default is HumanAgent;
--sleep: A boolean flag, if mentioned, then after each turn there will be a little pause (works only for computer agents and helps to follow the game flow);
--gui: A boolean flag, if mentioned, then the GUI will be shown;
--repeat: Choose a natural number of game repetitions. The default value is 1;
--output: Path to the existing output directory (with "/" at the end for unix platform and "\" for a windows one). If not mentioned, then output will be only printed, but not saved.
--version: If the chosen agent is alpha-beta, then this argument clarifies its version (1, 2 or 3). The default value is 2.
--depth: If the chosen agent is alpha-beta, then this argument sets the depth for the agent's run. The default value is 0.

An example of input:

```
main.py --agent=GreedySingleAgent --sleep
--gui --repeat=3 --output=output_dir/
```