CS 4341: Introduction to Artificial Intelligence

Project 1 Documentation

Team: DeltaGo
Kyle Carrero
Benjamin Longo
Harsh Rana

About

The project was created by Benjamin Longo, Kyle Carrero and Harsh Rana, all students of Worcester Polytechnic Institute, as part of their course work during *CS 4341: Introduction to Artificial Intelligence,* under professor Ahmedul Kabir. This document will highlight some of the features of the program, implementation details and compiling instructions.

Compiling and Running the program

This program uses **Python 3.6** and utilizes the following pip modules: **Fire, Halo, Numpy, Pytest** (if you want to run the test cases)

To make the AI play itself locally run-

```
python start.py local --a1=[agent_from_list] --a2=[agent_from_list] --h1=[heuristic_from_list] --h2=[heuristic_from_list] start
```

For Example-

```
python start.py local --a1=ab --a2=ab --h1=winning-windows --h2=winning-windows start
```

To play a reffed game run-

```
python start.py reffed --team_name=[name_string] --a=[agent_from_list] --h=[heuristic_from_list] start
```

For Example-

```
python start.py reffed --team_name=DeltaGo --a=ab --h=winning-windows start
```

List of Heuristics-

`Winning-windows` (Suggested)

`threat-space`

List of Agents-

`ab` - Alpha-Beta Pruning w/ iterative deepening (Suggested)

`nm` - Negamax

To get rid of all the referee files run-

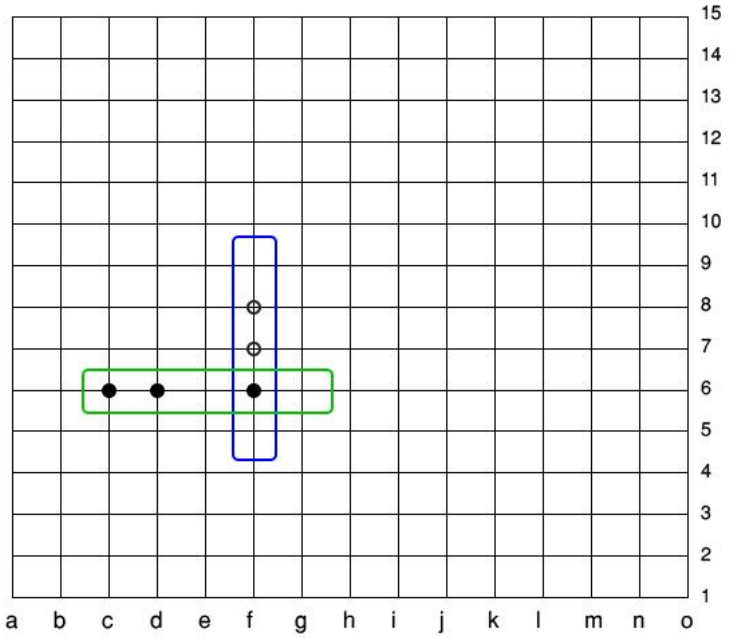
```
./clean
```

Evaluation Function

Currently, we have two evaluation functions that can be used as strategies. These are the *winning windows* and *threat space* evaluations. We decided on these strategies based on research done by the team members and feasibility of the said strategies. Additionally, we interpreted these strategies and created our own versions of them (mostly simplified versions). An in-depth description of both of these strategies can be seen below:

Winning Windows
The winning windows gomoku strategy is fairly simple to understand. A winning window is defined as a set of 5 consecutive positions where there are:

> 1 favorable (your) pieces and 0 unfavorable (opponent's) pieces



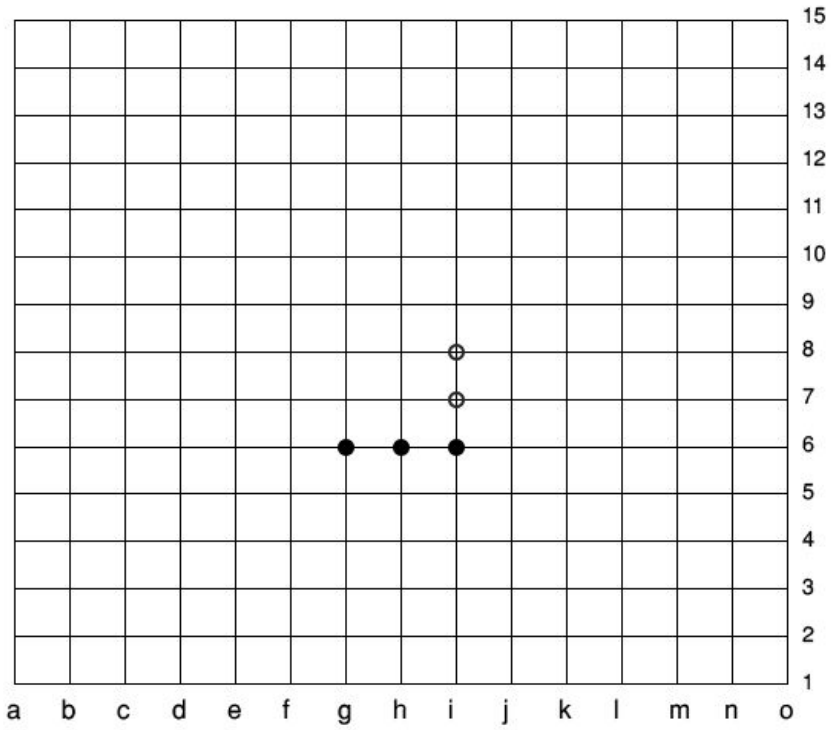
The above board shows two options for such a window. The blue window contains two white stones and a black stone and thus neither stones can win in that window. On the other hand, the green window contains 3 black stones and 0 white stones and thus can be considered a "winning window" for the black stone player.

These windows are responsible for winning games as it is impossible to create a 5-in-a-row when there are 0 winning windows. Thus, this evaluation function relies on increasing the number of winning windows to thus improve the chances of winning a game.

Our current implementation of this evaluation function finds all such winning windows for a player and values the board state based on this.

Threat Space

Threat space strategy was the second evaluation function we used. The threat space strategy is very similar to the winning windows strategy, where it looks at the different opportunities that a player has to win and evaluates a board state based on the number of such “threat spaces”. Where threat space search beats winning windows is by understanding and differentiating between different threats. This is done by caching a set of predetermined “threat types” and placing different values on each of them. Thus, when analysing a board state, the threat space search strategy can cross reference any threat found with its “bank of threats”. An example of this is show below:



In such a scenario (as shown above), the threat space evaluation function will realize the immediate threat posed by the g6, h6 and i6 pieces and thus place high value on “stopping such a threat”. Threat space strategy can be highly effective because it finds a potential winning sequence and then proceeds to find moves to curb such a threat.

Utility Function

We were able to use our evaluation functions (as mentioned above) as our utility functions as well. This was possible because the winning windows and threat space strategies can both evaluate a given board state and accordingly decide the utility of that state. Further details about how both the strategies do this is explained below:

Winning Windows

As mentioned earlier, the winning windows strategy focuses on creating the maximum number of windows where a certain player can win. Thus, given a certain board state, it analyzes all the potential windows, determines all the possible winning windows and attaches a utility score to the board accordingly. Additionally, the more winning windows available on the board, the higher the attached utility. Lastly, inside each of the winning window, the higher the number of desirable pieces, the higher the attached state utility.

Threat Space

In a similar manner to what Winning Windows accomplishes, Threat Space search places a higher priority on longer runs of consecutive pieces. Given a certain board state, it will analyze all of the runs of pieces and calculate a value based on the run lengths for each player, returning lower, sometimes negative, values for states that are at a greater risk of leading to a loss, and higher values for states with a better probability of winning.

Time Limit Considerations

In order to stay within the time limit of making a move, we implemented alpha-beta pruning with iterative deepening to make sure that we could limit the depth of the search tree without keeping it a constant. We also only allow our program to search for a move for 8 seconds so that we have enough time to perform the file I/O to write the move and alert the referee. We also implemented our program in such a way that many of the functions that are called on the board state, like what moves are available or whether there is a winner, cache their results to memory, so that any subsequent calls on that state do not incur time costs due to calculation.

Results and Observations

During our time creating and testing this program, we had our program play against itself to judge its performance against an equally matched opponent. At first, our program did not play very intelligently. It would often make seemingly random moves and ignore a win-scenario by ignoring a 4-in-a-row and making another random move instead. After implementing Alpha-Beta pruning with iterative deepening, our program improved a little, but it seemed like whichever instance was the black piece would make intelligent moves, and whichever instance was the white piece would make the worst moves possible. We discovered that there was a bug in our implementation that caused the player to always choose the best move and opponent to always choose the worst move (as though the minimax algorithm had flipped). Upon fixing this issue our program began to play relatively intelligently, and would actively play offensively and defensively depending on the situation.

Our program could definitely use some work though. The selection of a move takes longer than we had hoped and we would like to add several optimizations in the future to improve the speed of the move selection so that it can evaluate moves farther down the tree.

We are proud of our results and look forward to improving upon our implementation in the future.

Choice Justification

We believe that the evaluation/utility function that we chose was a good choice, as they actively look for threats to a win and seek to mitigate these threats. We also wanted a function that was easily understood so that we could implement it within the time of the project. The Winning Windows approach is a nice balance between offense and defense that is easily understood. We thought that for the sake of the project, a better defence was a better strategy than a good offense. For future implementations, however, we would like to find an evaluation/utility function that is more aggressive.