# CS 440/ECE448
# Assignment 2: Planning

*March, 2018*

YUHAO CHEN,
ZIRUI FAN,
JUNHAO PAN

**Credit Hour: 3**

# 1 Introduction

## 1.1 Overview

In this assignment, we have chosen python as our language to implement the functionalities. We majorly use jupyter notebook as the developing environment.

This assignment includes two parts: Planning and Go-Moku Two-Player Game Playing Agents, each of which addresses a distinct topic covered in this class during the past few weeks.

In part 1.1, we have implemented path-finding algorithms of which the goal is to find sequences to visit factories and gather widgets in certain orders. Mainly we use A* search to find the fewest stops and the shorted distances.

In part 2.1, we have developed a Reflex Agent which follows certain rules given by the documentation when playing the Go-Moku game on a $7 \times 7$ grid. In part 2.2, we have developed Minimax Agent and Alpha-Beta Pruning Agent which play the game of Go-Moku "smarter" than the Reflex Agent.

## 1.2 Files

| | |
|---|---|
| ./Smart Manufacturing.ipynb | jupyter nootbook for part 1 |
| ./Game of Gomoku.ipynb | jupyter nootbook for part 2 |
| ./search.py | supporting python file for part 1 |
| ./gomoku/* | gomoku library |

# 2 Smart Manufacturing

## 2.1 Planning Using A* Search

### 2.1.1 Implementation

Our goal is to spend the fewest stops/distances to collect all the components for widgets in the specified orders. Therefore, this problem is similar to graph traversing.

First, we have constructed a graph which holds the information of the distance, and, apparently, all factories are inter-connected. Thus, it is a complete graph. However, we can only traverse the graph in the order that is given by the widgets. For example, at the very beginning, we can only begin with one of factory **A**, **B**, or **D**. Then, supposed, we start from factory **A**, the following factories which we can visit are **B**, **D**, or **E**.

Therefore, we use an array to keep track of the frontier at which we are facing. The frontier is kept in order with a priority queue according to both the heuristic and cost as in

the nature of A* Search. When we pop one factory from the front of the frontier, we push the factory behind the popped one on to the frontier at its correct place. If this factory is already on the queue, we need to update the distance. Until the frontier is clear, we keep visiting the factory in the front of the queue. When there is no more factory which needs to be visited, all the components are collected, and we are done.

### 2.1.2 Heuristics

The assessment of a node for A* consists of the cost and the heuristic. The cost is quite straight-forward since it is simply the distance which it has traveled from the start factory to the current factory. We realized that the distance in between cities does not satisfy triangular inequalities, meaning that it is sometimes closer to go through a third factory. As a matter of fact, except for the pair of **A** and **C**, any other pairs of factories would have to go through **E** to make the distance closer.

The heuristic of this problem is not quite as clear. We first thought of the average number of components left in the widgets. Soon we realized it was idiotic. At last, we tried to think of a relaxed problem with fewer constrains. In this case, we did not take the order of the components into account and then count the stops we needed to make. It may not be the best approach, but it did do the job.

This heuristic is admissible since it would never over-estimate the cost. In this particular problem, since the orders of the components add quite some complexities to the traversing, considering the path without the orders is always shorter than what is really needed.

### 2.1.3 Results

We have the shortest path

**BEAEDCADBDCE**

and the shortest distance 6034, expanded 54219 nodes. We have also tried to use uniform cost search (Dijkstra's algorithm), but the node expansion seems to be huge. Therefore, we gave up on that.

# 3 Game of Gomoku

## 3.1 Reflex Agent

### 3.1.1 Implementation

The goal of this section is to develop agents that plays the game of Gomoku based on a given game state which can be read from the board.

The reflex agent is the kind of agent which follows certain rules which are set by the developers and does not predict the future. In this case, it will only focus on the current game state and make decisions base on that, meaning that it will not care about the consequences of the moves which it has made.

There are majorly four rules given by the documentation. The reflex agent will strictly follow those rules when making a decision. First, when it sees a friendly un-blocked four-in-a-row, meaning there are four of its own pieces in consecutive position on the board and either end of it has no hostile piece, it will place a move at an open end and thus effectively end the game. Second, when it sees an un-blocked hostile four-in-a-row, it will try to block it by place a piece on either end. Third, if it sees the opponent has openings on both ends of a three-in-a-row, it places a piece on either openings. Lastly, it calculates the winning-blocks and makes a decision based on the numbers and positions of the friendly pieces in the winning-block.

We have constructed a library to serve the purpose of this problem. We have a Board class, which keeps track of the states on the board. For example, an agent and call the functions to ask for the numbers of four-in-a-row, three-in-a-row, and winning-blocks. It also has function which tells if a space on the board is movable, as well as if any side has won the game. There are also classes of each agent. The agent classes basically finds a move based on the board given. The Reflex class does the job of find the next move based on the rules.

All the sources are imported into jupyter notebook where the board is initialized and passed to each playing agent as parameters when they are making moves. The game is essentially a while loop. When an agent takes its turn, it assesses the board state and make a move. After each turn, the board checks the winning conditions and either proceed to the next turn or end the game.

Specifically to this problem, when breaking a tie, the assignment requires us to make the move that is leftmost and downmost. Thus, we have an array of possible moves which satisfy all the criteria and sort it based on the col and row coordinates to break the tie. If the rules cannot generate any moves, we decided to make a random choice from all empty spaces on the board.

### 3.1.2 Reflex vs. Reflex

```
6   . . . . . . .
5   . k . . . A .
4   F H J K B . .
3   f E D C G h .
2   e g d i I . .
1   c a . . j . .
0   b . . . . . .

    0 1 2 3 4 5 6
```

Figure 1: Reflex vs. Reflex

## 3.2 Minimax Agent

### 3.2.1 Implementation

The Minimax Agent is supposedly "smarter" than the Reflex Agent because it makes decision after calculating the consequences of the future steps. We use the feature of a minimax decision tree to help the agent make decisions.

In general, minimax tree starts with a root node that takes the max value of the second level. Second level, however, are min nodes which takes the minimums of the third level. The min and max duality takes turns until a certain depth is reached. In this particular problem, we want to reach the depth of three. When a leaf node is reached, an evaluation function helps determine the value of this leaf and provide it to the parent node. Then, it will back trace to the top where a final decision is made. This model of decision making can effectively simulate the two-player gaming. The max nodes simulate the moves that the player is making, and the min nodes simulate the moves that the opponent is making.

In the Minimax class, there is a recursive function that does the backtracing in the minimax tree. It is called when the agent is taking a turn. It goes down to depth three. At the depth three, it calls the evaluation function and returns the values. At other depths, it checks if it is a level of min nodes or max nodes and choose the nodes to return accordingly.

Thus, the decision making process of a Minimax agent borrows some basic rules from the Reflex agent. For example, if the opponent has a four-in-a-row, the agent has to fix it immediate without having to go into the tree. This will save some time and space. If none of those rules can generate a move, the agent will dive into the tree and seek for a move. If that does not work, it will make a random choice on the other available spaces.

### 3.2.2 Evaluation

The evaluation function which we have developed is quite simple. We takes several components into account: five-in-a-row, four in one winning block, three in one winning block, and two in one winning block. We evaluate states of both player and opponent. The function will add points to the components in favor of the player and take off points to the components that benefits the opponents. For example, if the player is making a move that will lead to a four-in-a-row for the opponent and does nothing good for itself, it will most like receive a very bad or even a negative score for that move. We have realized that the winning blocks can be overlapping. We have taken those factors into account and tuned the weights carefully. However, we believe that the formula can be further perfected because we see some "surprising" moves occasionally which are most likely caused by mislead evaluations.

### 3.2.3 Alpha-Beta Pruning

Alpha-Beta Pruning is a strategy based on minimax tree to save calculations and increase performance. It produces the same outcomes as the minimax tree even with the pruning but it will expand considerably less nodes.

### 3.2.4 Plays

The Alpha-Beta pruning yields the same products as minimax. Therefore, we expect to see the same outcomes in the game play. However, due to the nature of the pruning, the node expanded by Alpha-Beta should be less than Minimax without pruning. Here are all the outcomes we have.

```
6   r Q M G U K l
5   . O n g o p P
4   . I f e i J h
3   . c D a N . L
2   H C B d F m k
1   j b u E . . .
0   A q R e S t T

    0 1 2 3 4 5 6

Winner is blue
```

Figure 2: Minimax vs. Alpha-Beta

```
6   r Q M G U K l
5   . O n g o p P
4   . I f e i J h
3   . c D a N . L
2   H C B d F m k
1   j b u E . . .
0   A q R e S t T

    0 1 2 3 4 5 6

Winner is blue
```

Figure 3: Alpha-Beta vs. Minimax

```
Move:1 0 0
Move:2 12084 4704
Move:3 24567 7497
Move:4 1314 10976
Move:5 1893 0
Move:6 429 4116
Move:7 551 0
Move:8 612 0
Move:9 739 0
Move:10 271 9751
Move:11 5463 4802
Move:12 372 1568
Move:13 263 1519
Move:14 339 1666
Move:15 397 0
Move:16 135 0
Move:17 0 833
Move:18 0 441
Move:19 0 245
Move:20 0 245
```

Figure 4: Minimax vs. Alpha-Beta

```
Move: 1 0 0
Move: 2 4165 4106
Move: 3 4214 6349
Move: 4 5537 6843
Move: 5 3528 0
Move: 6 2058 2685
Move: 7 245 0
Move: 8 4508 0
Move: 9 1862 0
Move: 10 1176 1315
Move: 11 1421 1678
Move: 12 2205 469
Move: 13 1421 292
Move: 14 196 159
Move: 15 686 0
Move: 16 0 0
Move: 17 0 188
Move: 18 0 49
Move: 19 0 9
Move: 20 0 9
```

Figure 5: Alpha-Beta vs. Minimax

```
6   . F . c . B A
5   L e E D C f .
4   j K J . . l .
3   M g m a h n .
2   . d k . . . .
1   I b . . . . .
0   H G . . i . .

    0 1 2 3 4 5 6

Winner is red
```

Figure 6: Alpha-Beta vs. Reflex

```
6   . . . . . . .
5   h A C G D H .
4   . . . g B . .
3   . . . a . . .
2   d c b e E . .
1   . . . f . . .
0   . . . F . . .

    0 1 2 3 4 5 6

Winner is blue
```

Figure 7: Reflex vs. Alpha-Beta

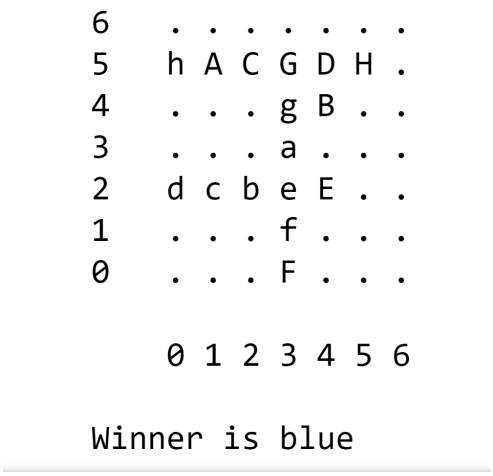


Figure 8: Reflex vs. Minimax

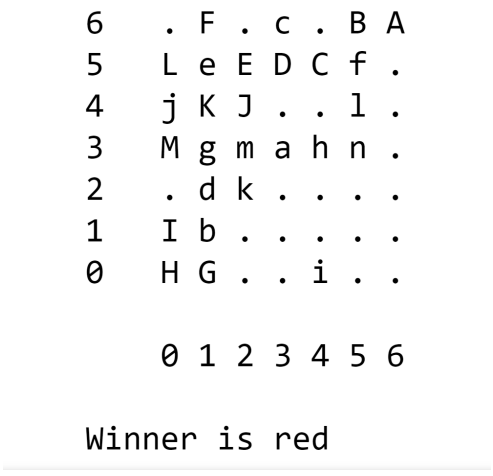


Figure 9: Minimax vs. Reflex

# 4 Credits

We worked together wonderfully in coding the solutions. Everyone contributed in constructing heuristics and finding algorithm. Junhao Pan typed the report in LaTeX, and the others reviewed it.