CS 440/ECE 448 Spring 2018

Assignment 2: Smart Manufacturing, Two-Player Games

Due at: Monday, March 12, 11:59 PM

In this assignment, you will build search algorithms specialized for two types of special problems: planning a shipping logistics problem, and planning the actions of an agent that plays a game.

You are free to use any high-level programming languages you are comfortable with, including but not limited to Java, C++, Python, and MATLAB. You have the option of working in groups of up to three people, all of whom should take this course for the same number of credit hours. We focus on the entire problem solving process and you should hand in **the code** with your implementation (as a ZIP file) and **a written report** with your analysis (as a PDF, separate from the ZIP file).

Point totals are listed next to each section header. Three-credit students can earn a maximum of 24 points plus 6 extra credit points, whereas four-credit students can earn a maximum of 32 points plus 8 extra credit points. Four-credit students are required to finish Part 2, and can try Part 2 extra credit section for extra credit. Three-credit students can try Part 2 for extra credit, but are not supposed to finish Part 2 extra credit section since your extra credit points will be capped for balance.

Contents

Part 1: Smart Manufacturing

1.1 Planning Using A* Search (for everyone; 8 points)

You are the owner of a smart manufacturing company, with factories in five different cities: Atlanta, Boston, Chicago, Denver, and Edmonton. Each factory makes a different type of component: for convenience, let's call the components A, B, C, D, and E. The locations of your factories are a bit inconvenient to one another; the following table gives the distance (in miles) from one factory to another.

Distance	A	B	C	D	E
A	0	1064	673	1401	277
B	1064	0	958	1934	337
C	673	958	0	1001	399
D	1401	1934	1001	0	387
E	277	337	399	387	0

From those components, you manufacture five different types of widgets. Each widget is composed of five widgets of different types, as follows:

- Widget 1 = Components AEDCA

- Widget 2 = Components BEACD
- Widget 3 = Components BABCE
- Widget 4 = Components DADBD
- Widget 5 = Components BECBD

Thus, for example, widget 1 is manufactured by building components A, E, D, C, and A, in that order. Each component can only be added once the previous components are already in place; you can't manufacture it in advance.

Every day, you put five empty widgets in a truck, and the truck visits factories in sequence as necessary to make all five widgets. While you're at factory A, you can add component A to any widget that needs it, but you can't work ahead. The truck must visit all of the factories necessary to make all five widgets, in the order specified by the widgets. In other words, you need to come up with a sequence of letters that contains each of the sequences above as a subsequence. Here are some example solutions:

- AEDCABEACDBABCEDADBDBECBD (25 stops)
- ABBDBEEAAEDABDCCCCBBADEDD (25 stops)
- ABDBEAEDABDCBADED (17 stops)

The goal of this assignment is to implement A* search that will find two different factory sequences:

1. First, find the factory sequence with the smallest number of stops (the smallest number of factories visited).
2. Second, find the factory sequence with the smallest number of miles traveled.

Solve this problem using A* search. Your report should specify your state representation, actions, and your heuristic. The heuristic need not be very smart, but it needs to be non-zero, and it needs to be admissible and consistent.

1.2: Planning Using a Planning Graph (4 points required for 4-credit students; 2 extra credit points optional for 3-credit students)

Implement the GRAPHPLAN algorith, as described in section 10.3.2 of the textbook, and in this article. In this algorithm, you will create a graph in which each level of the graph is a list of (variable=value at time t) assignments (called "fluents"), and a list of mutex (mutually exclusive) liks among the fluents. Levels are linked by actions, such as "travel from city A to city B." When you reach a level at which all of the goal conditions are non-mutex, you attempt a backward search to find a sequence of actions. If the backward search fails, you add another level to the graph and try again.

Use the planning graph to find the minimum-distance sequence of factories. Hand in a plot of the lowest-distance solution resulting from a graph with N, N+1, ... levels, where N is the first level with any solution.

1.3 Extra Credit (2 points each)

Extra credit tasks are worth up to two points each, up to a maximum of 25 percent (maximum extra credit = 6 points for 3-credit students, 8 points for 4-credit students).

1. Implement a random problem generator, that randomly generates five widget definitions of length N, where N is a parameter that you can specify (in the required problem, N=5). Hand in a plot of the number of nodes expanded by each of your two A* search algorithms, as a function of N, for 3<=N<=8.
2. Implement uniform cost search (Dijkstra's algorithm) for each of the two problems specified in part 1.1 (minimum number of stops, minimum distance traveled). Compare the number of nodes expanded using uniform cost search versus A*.

Part 2: Game of Gomoku (Five-in-a-Row)

Created by Zhonghao Wang

The goal of this part of the assignment is to implement an agent to play a simple 2-player zero-sum game called Gomoku.

Rules of the Game: This game plays with stones of two colors (red and blue in these figures), on a board with 49 empty spaces arranged in a 7x7 grid. Players alternate turns placing a stone of their color on an empty intersection (in these

instructions, the player who goes first uses the red stones, and the player who goes second uses the blue stones). The winner is the player who first forms an unbroken chain of five stones horizontally, vertically, or diagonally. The game is a tie if all intersections are filled up with stones but neither side wins the game.

Part 2.1 (4 points for all): Reflex Agent

The goal of this section is to create a reflex agent following a set of pre-specified rules, so that, given the same initial board position, the reflex agent will always react in exactly the same way. It uses the following strategies:

1. Check whether the agent side is going to win by placing just one more stone. If so, place the stone which wins the game. For example, 4 blue stones form an unbroken chain shown in figure 1. The agent at the blue side would place a stone on either head of the chain, (0, 4) or (5, 4), to win the game. To break a tie, choose a move in the following order: left > down > right > up. Therefore, (0, 4) would be the move to make.

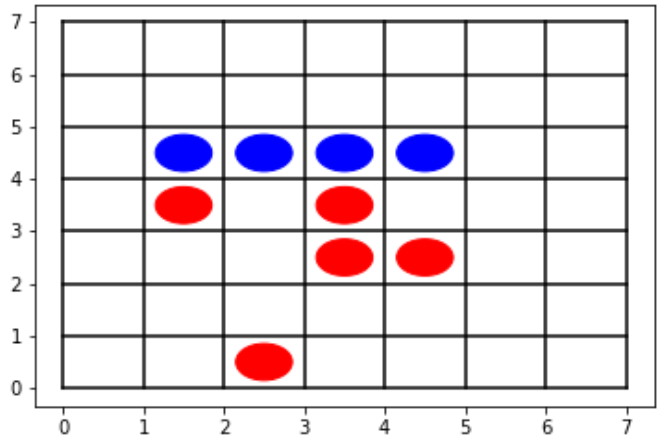

Figure 1: Illustration of reflex agent rule #1

2. Then check whether the opponent has an unbroken chain formed by 4 stones and has an empty intersection on either head of the chain. If so, place a stone on the empty intersection. In the case shown in figure 2, the agent at the blue side would put a stone on (5, 3) to prevent the red side winning the game.

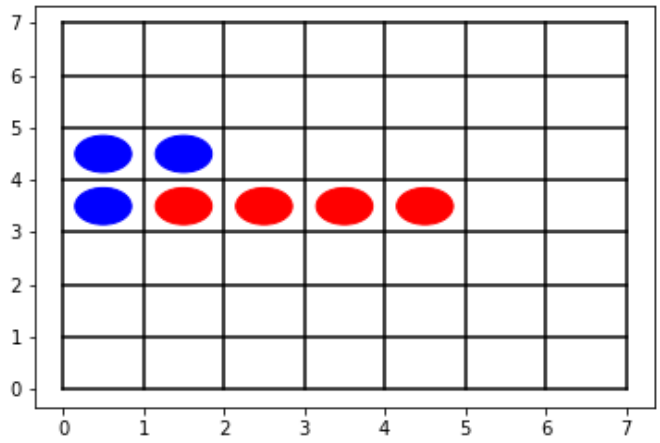

Figure 2: Illustration of reflex agent rule #2

3. Check whether the opponent has an unbroken chain formed by 3 stones and has empty spaces on both ends of the chain. If so, place a stone on an empty space at one end of the chain. Choose which end of the chain to fill following the order: left > down > right > up. In figure 3, the agent at the blue side would place a stone on either (1, 4) or (5, 4). To break a tie, (1, 4) would be chosen.

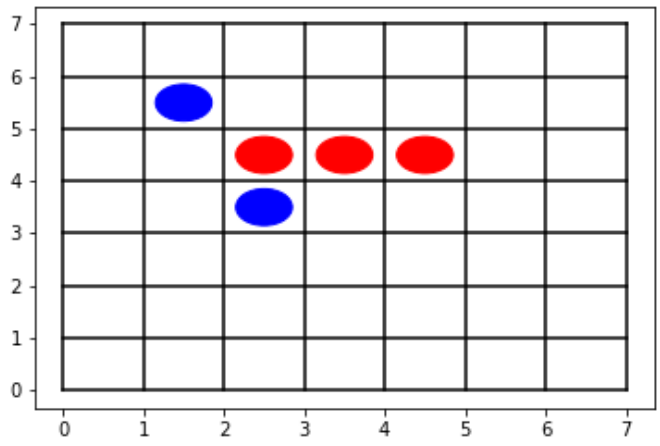

Figure 3: Illustration of reflex agent rule #3

4. If none of the previous conditions hold, then find all possible sequences of 5 consecutive spaces that contain none of the opponent's stones. Call each such block a "winning block," because it's a block in which victory is still possible. Then, find the winning block which has the largest number of the agentâ€™s stones. Last, in the winning block, place a stone next to a stone already in the winning block on board. You may find multiple winning blocks or multiple positions within a winning block to place a stone. To break a tie, find the position which is farthest to the left; among those which are farthest to the left, find the position which is closest to the bottom. For example, in figure 4, the agent at the red side would place a stone on (0, 1).

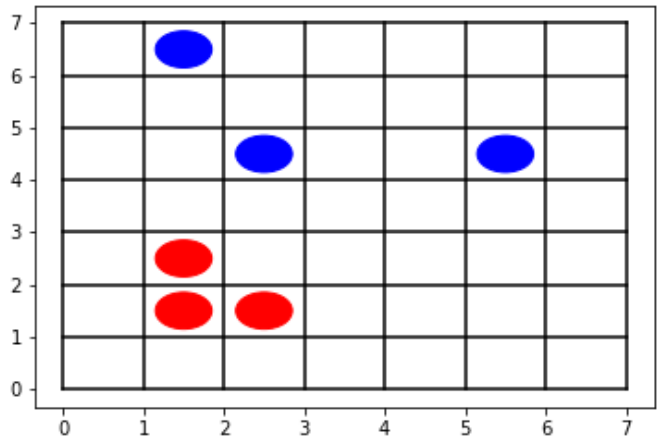

Figure 4: Illustration of reflex agent rule #4

5. **Special rule for the agent's first move:** For the very first move, you can implement any strategy you like. You can follow rule #4 above (by playing in the bottom left corner), or you can play at random, or you can make an optimal move (in the middle), whatever you like. This applies only to each player's first move; each player's second move must follow rules #1-#4.

Test your reflex agent to make sure it responds, as indicated, to the four examples shown above.

2.2 Minimax and Alpha-beta Agents (12 points for all)

Implement both minimax search and alpha-beta search agents. Implement your minimax and alpha-beta agents so that each of them searches to a depth of three (agent moves, opponent moves, agent moves, and then evaluation function is applied to evaluate the board position). Test your minimax and alpha-beta agents to make sure that, when given the same board position, they both produce exactly the same move, but the alpha-beta agent comes up with that move after expanding fewer nodes in the search tree.

Design an evaluation function that is accurate enough to permit your minimax and alpha-beta agents to beat the reflex agent. Give the same evaluation function to both the minimax and alpha-beta agents. Describe your evaluation function in your report.

Part 2.3 Stochastic search (4 points required for 4-credit students; 2 points extra credit for 3-credit students)

Implement a Stochastic-search agent for the 7x7 Gomoku game. Your Stochastic-search agent should be an alpha-beta agent with a depth of 2, but the value of each leaf node in the search tree should be estimated by generating N random games from that leaf position. Experiment a little with the following parameters, and be ready to defend your choices based on either experimental or theoretical arguments, e.g., by considering the quality of your result versus computational complexity.

- **Policy>** At each step of the simulation, how do you decide where to play?
- **Breadth:** How many simulations do you use, from each board position, to determine its value?
- **Depth:** If you wish, you can abandon a simulation whenever it exceeds some maximum simulation depth. If so, what depth makes sense? Why?
- **Value:** How do you estimate the value of a board position based on the simulations? For example, how much is a tie worth? How much is it worth if a game has to be abandoned because it exceeds the maximum simulation depth?

2.4 Extra Credit (2 points each)

Extra credit tasks are worth up to two points each, up to a maximum of 25 percent (maximum extra credit = 6 points for 3-credit students, 8 points for 4-credit students).

1. Implement a user interface that allows a human being to play against the board, using a graphical user interface that paints a picture of the board on the screen. Show screenshots of a user playing against the agent as figures in your report, and submit your code.
2. Use supervised learning to estimate an evaluation function: (1) randomly generate a large number of sample games, (2) implement a function that featurizes each board position, and estimates the value based on those features, e.g., using linear regression as described in lecture. Turn in both your training and test code. Note: you can use numpy and scipy for this, but not TensorFlow, Theano, or any other toolkit specialized for machine learning.

Report Checklist

Your report should briefly describe your implementation and fully answer the questions for every part of the assignment. Your description should focus on the most "interesting" aspects of your solution, i.e., any non-obvious implementation choices and parameter settings, and what you have found to be especially important for getting good performance. Feel free to include pseudocode or figures if they are needed to clarify your approach. Your report should be self-contained and it should (ideally) make it possible for us to understand your solution without having to run your source code. For full credit, in addition to the algorithm descriptions, your report should include the following.

- Part 1.1 (Required for all)
  - Give a path with the smallest possible number of stops. Describe your heuristic, and give the number of nodes expanded.
  - Give a path with the smallest possible distance. Describe your heuristic, and give the number of nodes expanded.
- Part 1.2 (Required for 4-credit; Extra-credit for 3-credit). Provide a figure or table showing, as a function of the graph level, each of the following facts about your planning graph. The number of levels in the graph should be large enough that (1) there is at least one solution from the last level, and (2) the number of mutex links, and the number of miles traveled by the shortest path from that level, are unchanged between the second-to-last and last level of the graph.
  - The number of fluents at that level of the graph (integer)
  - The number of mutex links (integer)
  - Whether or not all of the goal fluents are non-mutex at that level (binary)
  - Whether or not there is any solution from that level (binary)
  - The minimum number of miles traveled by any solution that can be achieved at that level

Give the minimum-distance solution achieved in this way (the sequence of cities, and the distance traveled).

- Part 2.1 Reflex Agent (Required for all). Play the reflex agent against itself, starting from the board position shown in figure 5. Show us the ending game board. Use '.' to denote empty intersections, use small letters to denote each of the red stones (the first player to play) in the order in which they were played ('a', 'b', 'c', etc.), and use capital letters to denote each of the blue stones (second player) in the order in which they were played ('A', 'B', 'C', etc.). You may submit your final board position as a plaintext matrix, or as a graphic, but you MUST show us the order of play using consecutive letters. Note: since the first two moves are given to you, the special first-move rule (rule #5) doesn't apply; all remaining moves must follow rules #1-#4.

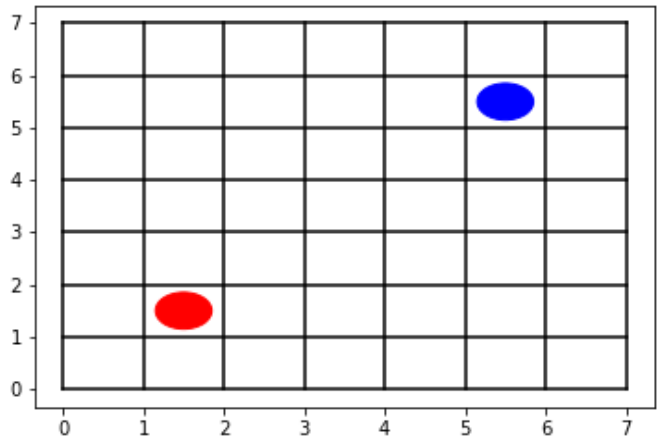

Figure 5: Starting board position for the required reflex agent vs. reflex agent game.

- Part 2.2 Minimax and Alpha-Beta Agents (Required for all)
  - Explain how you implement minimax search and alpha-beta search. Specifically, explain your evaluation function and how you do the alpha-beta pruning.
  - Implement the following match-ups starting from an empty board, and return the final board positions. Use '.' to denote empty intersections, use small letters to denote each of the red stones (the first player to play) in the order in which they were played ('a', 'b', 'c', etc.), and use capital letters to denote each of the blue stones (second player) in the order in which they were played ('A', 'B', 'C', etc.). You may submit your final board position as a plaintext matrix, or as a graphic, but you MUST show us the order of play using consecutive letters.
    1. alpha-beta vs. minimax
    2. minimax vs. alpha-beta
    3. alpha-beta vs. reflex
    4. reflex vs. alpha-beta
    5. reflex vs. minimax
    6. minimax vs. reflex
  - For the alpha-beta vs. minimax and minimax vs. alpha-beta match-ups, provide a table showing, as a function of the move number, how many nodes in the search tree were expanded by each agent in order to find its move. The table should look something like this, but with different numbers:

move	**Nodes expanded, red**	**Nodes expanded, blue**
1	156748	96400
2	102354	74300
...	...	...

  - Briefly analyze the relationship of the number of nodes expanded per move between minimax agent and alpha-beta agent.
- Part 2.3 (required of 4-credit students)
  - Implement the following two matchups, starting from an empty board. Here "alpha-beta" uses the evaluation function you implemented in part 2.2:
    1. Stochastic-search vs. Alpha-beta with depth of 2
    2. Alpha-beta with depth of 2 vs. Stochastic-search

    Print out the final state of each of these two boards. Use '.' to denote empty intersections, use small letters to

denote each of the red stones (the first player to play) in the order in which they were played ('a', 'b', 'c', etc.), and use capital letters to denote each of the blue stones (second player) in the order in which they were played ('A', 'B', 'C', etc.). You may submit your final board position as a plaintext matrix, or as a graphic, but you MUST show us the order of play using consecutive letters.

- Answer the four questions listed in the assignment description, about the policy, breadth, depth, and value of your stochastic search algorithm.
- Report the total number of simulated moves necessary for the Stochastic-search agent to make each move in the actual game, for each of the two match-ups. Your table should look something like the following, but with different numbers:

move	Game A	Game B
1	45637	139873
2	129864	34563
...	...	...

- Report the value of the move chosen by the Stochastic-search agent at each move, in each of the two games. This value should be a number between 0 and 1, specifying what the Stochastic-search agent believes to be the probability that it will win the game if it makes its best possible move. Your table should look something like the following, but with different numbers:

move	Game A	Game B
1	0.536	0.598
2	0.574	0.629
...	...	...

Submission Instructions

By the submission deadline, **one designated person from the group** will need to upload the following to **Compass2g**:

1. A **report** in **PDF format**. Be sure to put the **names** of all the group members at the top of the report, as well as the number of credits (3 or 4). The name of the report file should be **lastname_firstname_a1.pdf** (based on the name of the designated person).

2. Your **source code** compressed to a **single ZIP file**. The code should be well commented, and it should be easy to see the correspondence between what's in the code and what's in the report. You don't need to include executables or various supporting files (e.g., utility libraries) whose content is irrelevant to the assignment. If we find it necessary to run your code in order to evaluate your solution, we will get in touch with you. The name of the code archive should be **lastname_firstname_a1.zip**.

Compass2g upload instructions:

1. Log in to **https://compass2g.illinois.edu** and find your section.
2. Select **Assignment 2 (three credits)** or **Assignment 2 (four credits)** from the list, as appropriate.
3. Upload **your PDF report** and **the ZIP file containing your code** as two attachments.
4. Hit **Submit**. *-- If you don't hit Submit, we will not receive your submission and it will not count!*

Multiple attempts will be allowed but only your last submission before the deadline will be graded. We have variable grace periods, so if you submit your files on 12:00 am the next day due to network latency, do not panic. However, please never rely on the grace period and go any further. **We reserve the right to take off points for not following instructions.**

Late policy: For every day that your assignment is late, your score gets multiplied by 0.75. The penalty gets saturated after four days, that is, you can still get up to about 32% of the original points by turning in the assignment at all. Extensions without penalty of up to 72 hours will be granted for any assignment if you are ill (documented by a note from your doctor) or have a life-threatening emergency (documented by a note from the emergency dean, a court document, or some comparable documentation). If you are part of a group, extension is granted only for the part of the project that is affected by the illness or emergency; this is the only reason for which separate submission of sub-projects is ever accepted.

Be sure to also refer to course policies on academic integrity, etc.