CS440: Assignment 2: Smart Manufacturing, Two-Player Games

Oluwatobi Ijose (okijose2), Luke Staunton (stauntn2)
3 Credit Hour Section
3/12/18

1.1 Planning Using A* Search

To solve part 1.1 of the MP we used a StateNode class as our primary data structure to represent our states. The StateNode object contained the path that the agent took up to that point. It was able to do this by having a character representing the agent's current location, a progress vector showing how far the agent is in the assembly of each of the widgets, and pointer to the parent StateNode.

The algorithm works by creating a StateNode with no current position, and a progress vector filled with zeros. It then expands that node using a "get_transitions()" method to get a list of child nodes and puts all of them into a priority queue. The priority queue ranks the nodes by the heuristic function, plus the length of the path it has already traveled. It determines valid transitions as all nodes created by transitions that would advance the progress on at least one widget. When those new nodes were created, if the transition caused the advancement of the development on a widget, the element in the progress vector describing that widget was incremented.

It then pops a state node from that heap and repeats the process until a node is found that has all five widgets completed. Or in other words, all five elements in the progress vector were equal to five. Since our heuristic was consistent and admissible, the described algorithm is an A* search.

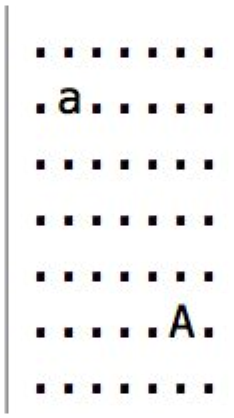
In the unit distance case, where all the factory locations were equidistant, our heuristic function is the maximum of two smaller functions. The first, being the maximum number of remaining stops to complete a given widget. This means that even if four of the widgets were completed and the last widget was un-started, the cost would be five. The second function computes how many unique characters still need to be visited. By taking the maximum of these two, we got a strong and consistent heuristic.

In order to adapt it to the non-unit distance case, we multiplied that cost times the minimum distance between two nodes. This gave us the path "BAEDCADBCDE" as our shortest unit-distance path and "BEDAEDCBCAED," which had a cost of 7043, as our shortest non-unit distance path. The unit cost version had to expand 1613 nodes while the non-unit cost expanded 6175 nodes to find the solution.

2.1 Reflex Agent

To solve part 2.1 of this MP, we were tasked to design a reflex agent to play the game of Gomoku. This agent needs to respond in a specified manner, to a specified state of the game. To accomplish this, we first represented the board as a 2 dimensional array of dots for the initially empty board, filling it in with lowercase letters 'a'...'z' corresponding to player 1's moves, and uppercase letters 'A'...'Z' for player 2's.

The first necessary class is our board class. This class has various functions to initialize the board, update the board, based on moves, and print the board after each move is made. The board stores the current player, and a state representation of the board. The state representation of the board is simply a 2d array of the characters at each position, and is used to print the board. Below is the initial state of our reflex agent vs. reflex agent game.

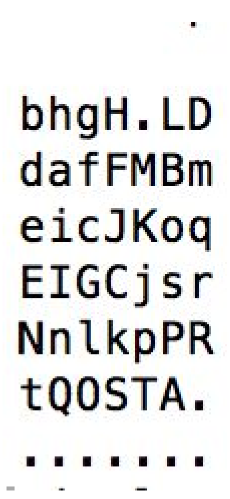


```
.......
.a.....
.......
.......
.......
.....A.
.......
```

The only other class written for this portion of the MP is the reflex agent class. This is to be our implementation of the basic reactionary player agent. Within this class we implement a search function to search for chains of a certain color, or chains lacking a certain color. The search function gets passed in the board as a pointer, an initial position (row, col) for the search, a final position for the search, and which color chip to search for. Actual implementation of this class is done in a for loop, which checks to see if the letter at each position is uppercase or lowercase. If the case at adjacent positions or adjacent diagonal positions match, then we know that it is the same chip at those positions. We then increment a counter to count how many chips of a specific player are in a row. Note that spaces without any chips are represented by dots, and thus will not satisfy the condition of being uppercase or lowercase and thus will not increment the count.

This method is used throughout each of the remaining functions of the class. The remaining functions correspond to the different rules the reflex agent must follow. The first rule checks to see if the current player has a winning move. To discern this, we call the search function and check if the count is equal to 4, i.e. if we have found four of the same chips in a row, column, or diagonal. If this is true, we first try to place a new chip in the following order: left, right, up, down. To find where the chip is to be placed, we check if the indexes are out of bounds, or occupied by a chip from the other player, than place them according to the given order. The second and third rule work in a similar fashion, except they are checking the opponents chip, as opposed to the current players, thus even though the current player is the one executing the function call, it is the opponent's chip color that is passed into the search function.

The final rule is to see if there are any open spaces on the board that can provide a winnable move for the current player. This function searches for chains of length 5, that have no consecutive opponent pieces, then checks among those blocks for the chains with the most consecutive current player pieces. Among those chains that have the most consecutive player pieces, we then choose the spot that is furthest southeast, and place our chip there. Below is a graphic showing the final state of the board, after player2 (uppercase letters) has won the game.



```
      .
bhgH.LD
dafFMBm
eicJKoq
EIGCjsr
NnlkpPR
tQOSTA.
.......
```

2.2 Minimax & Alpha-Beta Agents