Rahul Pokharna rkp43

EECS 391

P1 Writeup

1. **Code Design**

I wrote my assignment in python, with the imports at the top, the functions written, and then the main function. There are two global variables, of the goalState and goalBoard, and these are used to check and compare the current state to the goal and check for completion. I also use this to set the state to the final one before randomizing the state. I was unable to complete beam solve, but I have completed A\* solve with both heuristics. I decided upon creating a class that represents the board, with the parent state, and the move that was taken to reach that state. I redefined the equivalence to check the states against each other, as well as created a constructor that would have the basic needs of the puzzle when created. I had a function that would create a child class, and when that class is created, it initializes as a copy of the original with the parent class stored in a pointer, but when the child class takes a move, it updates the parentMove to store how it was changed to that state. I convert between a string representation of the board to a two dimensional array so as to allow for ease of understanding the current state during programming, as well as printing and moving the board states around. I also created an additional method to output the current board state as a string in the same format as the input. I also commented a brief description before very method and most of the complicated portions of the code, explaining the loops and if statements. Overall, the design is somewhat rushed, but due to the properties of python everything is in a set pattern.

1. **Code Correctness**

The only search algorithm included is that of A\*. The seed I used was always 100, and I did 100 steps of the random scramble of the board state. After running this with the A\* search with the displacement heuristic, I got a message when I exceeded the limit of maximum nodes allowed, but it was solved using the Manhattan heuristic. These solutions do work, after manually testing the solution by hand. Show in the attached code, I copied the solution and converted it to the proper format of the order, and you can see that the puzzle is solved there! I did this for both h1 and h2, as they both find the optimal solution path for the current puzzle state.

The A\* search starts off with a set of basic values put into place. To limit the number of nodes created, every time one is made, the number of nodes opened, numNodes is incremented. I create a list of all open nodes, the ones to keep expanding, and the closed nodes, ones we have decided to ignore at this point. I start off with the current state, and a while statement for as long as there is some open state, the loop will continue running and checking for children. If the loop has created too many nodes, or it is solved, the program exits the method with a return of the state of solution. If it is not solved or exceeded the number of allowed nodes, the program then starts branching and checking puzzles and their children states. This is where the class functionality of the puzzles comes in use, as when I create the new children node, I instantly have them take a move in the allowed direction, creating a new node. I make a list of every possible legal child for the current board state being analyzed, and then check if any of those have been seen before, and are in the open or closed list of nodes. Depending on whether the heuristic is good or bad, if we encountered this list before, we change the values already existing or move the value from the closed list to the open list. If we have not encountered this state before, we add it to the open list of children to view. After iterating through every child for the current state, we sort the list according to the f value, calculated by the depth, or number of moves to get to that point, added to the heuristic value that we chose to use, either H1 or H2 inputted at the call of the function. We then go to the start of the loop once again, and pop off the value that has the lowest f value, and repeat the process until we find a solution, run out of nodes, or there are no more states in the open list, thus having no possible solution for the puzzle. This way, we explore every value, and ignore those that have a worse f value than our current best states and solutions, even if we find a solution state, we go to the lowest f value until that state is a solution.

1. **Experiments**
   1. For beam search, the greater the width, the better the algorithm, but also the more resources it will use. A\* Search will always find the solution, and generally will not exceed the limit, but with h1 it is more likely to do so. As we decrease the maxNodes, we get less and less solvable puzzles for both states, decreasing more and more for each search algorithm.
   2. For A\* Search, H2 is better, as seen by the number of nodes used in the test done. H2 is a better heuristic because of the greater variance in the possible values of the heuristic, allowing for a greater disparity between two states that are somewhat similar, thus allowing to choose the best state to expand more reliably.
   3. Solution length is the same between h1 and h2 for A\* Search as they both get the best solution, while beam will return any solution it finds, if any. Beam is less reliable, and less likely to get a solution.
   4. I did not have time to do the experiments, but based on tests and theory as well as the textbook and other resources, I would say that generally there would be about 24 moves to solve each, a total maximum of 31 moves, h2 uses approximately 1600 nodes, while h1 uses around 40,000 and beam uses even more than that. I estimate that with an allotted 10,000 nodes, about half of the states are solvable with h1, all of them with h2, and beam search, I am unsure.
2. **Discussion**
   1. Based on my hypotheses, A\* with h2 as the heuristic is the better way to solve this problem. It finds the best path, in the least amount of nodes, and can only sometimes be slower than beam search.
   2. For implementing the beam search, I was not present for that class as I was sick that day, and so missed out on doing beam. I tried to implement it, but due to other reasons I wasn’t able to even have time to complete anything for beam. I read it over and decided that the cost of trying to do beam search was to lose out on the writeup, and so I decided to cut my losses. As for implementing A\*, I had some difficulties at first of just understanding the search, and how to implement it. Once I reread the chapter and the notes it became clearer, but it still took time to figure out how to create children and every possible child for the nodes. This part of the code took me the longest overall, and it was really difficult for me to complete, but once I got it, the rest of the program fit into place very quickly. As for testing, that was the hardest part as they took real time to solve, and I made some bad decisions and thus lost some time.