CSDS 391 HW3

jxc1906

September 2024

1 Problem 1

We can prove that the Uniform Cost Search as a special case of $A*$ by recognizing the evaluation function of both. For UCS, the path is found by expanding the node with the lowest cost currently. Thus, the evaluation function can be recognized as $f(x) = g(x)$, where $g(x)$ is the cumulative sum of the path. For A^*, we write the evaluation function as $f(x) = g(x) + h(x)$, where we now consider both the cumalitive path sum and the estimated cost of getting to the goal. Since heuristics are also a function, we can have a very bad, but still valid estimation of the cost to be uniform, or 0. Hence, we get $f(x) = g(x)+0 = g(x)$.

2 Problem 2

Please see Figure 1

3 Problem 3

- – Hill Climbing
  - – A local beam search will only keep a limited amount of nodes in its memory at a time. When we only fit one state at a time in memory, we are now essentially performing the hill climbing algorithm where we only find the best move at any given moment and attempt it.
- – Breadth First Search
  - – This is essentially just breadth first search as, with the algorithm allowing unlimited nodes to be stored, beam search will search all possible neighbors and expand each one of them. This is the same as BFS as BFS also explores all its possible neighbors until the goal state is reached.
- – Stochastic Hill Climbing

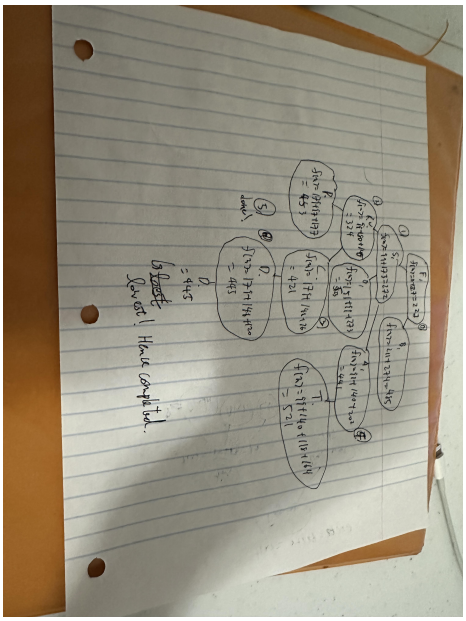

Figure 1: Drawing for Problem 2

  - We know for simulated annealing, the probability of picking a bad move is porportional to the Boltzmann distribution, defined as $e^{-\Delta E/T}$. When T=0, this equates of $p = 0$, hence the program will only choose the best move. As simulated annealing starts off at a random state, this is the same as stochastic hill climbing, where the program starts off at a random state and only move the best move.

- – Random Walk
  - With $T = \infty$, the computer will chose with a probability $p = e^0 = 1$, hence the computer will choose any move. This is thus the same as random walk, where system will allowed to explore all possible states randomly through the state space.

4 Problem 4

4.1 Implementing h1

To get the heuristic of the number of misplaced tile (where we call h1), I simply ran through the goal list. While iterating, as long as the element (on the eightboard) is not 0, I simply added 1 to the number of misplaced tiles. Then I return the number

4.2 h2

To get the Manhattan Distance, we iterate through the list to find the distance for each element (the distance from its current position to the position on the goal state). To calculate, we break the total distance down to the sum of the distance vertically and horizontally. To get the distance vertically, we simply calculated which row the number is at by dividing by 3 and only getting the integer part (since each row has three columns, thus for elements 0, 1, 2, they are in row 0, and, floor(0/3)=floor(1/3)=floor(2/3). The principle is the same for the rest of them). Then, we get the difference in column by taking its modulus with respect to 3 (since indexes 2, 5, and 8 are in the same column). After that, we simply added up the two distances. Then, for each element, we add the distance (skipping over the empty element) to get the final Manhattan distance for the board.

5 Problem 5

To do state checking, I simply implemented a set call 'visited.' The set data structure is chosen as it can automatically hash the element being added and quickly determine if the element is already present in the set. To allow hashing, before adding the state to the set, I first had to cast the eightBoard from a list to a tuple (as a tuple is immutable). Then, throughout the program (currently only in A* as required by the assignment), every time I try to expand a node, I first check if the node is already in the set, and, if yes, simply skip the node and expand the next one. If it has not been, expand it (adding it to the queue) and add it to the set. To demonstrate that this is done, a test command was put into one of the test case file (in line 47) to display its exact process.

6 Problem 6

To do the A* search, I simply modified the breadth-first-search algorithm. The first change I made was to ensure that the queue is now a priority queue. To achieve this, I used the heapq library from Python to use heapsort to sort by the smallest f(n) (where f(n) is the heuristic, as defined by the user, plus the current number of moves). This is achieved with the pushheap command available, where we push a 3-element tuple: the first element being f(n), second element being the f(n), and the third element being the move list. At the initialization stage, I simply pushed the tuple into an empty queue. Then, similar to breadth first search, while the queue is not empty, I pop out the 0th element (the element with the smallest f value). Then, I check for whether or not the maximum node has been created (this is done by using a counter that updates every time a node is expanded). If not, then we check if the solution state has been reached, and if it has, simply print out the values as defined in the assignment. If neither of these cases are true, I simply check if the state has already been visited. If it hasn't, calculate the f value, push this to the heap with the same three-tuple

format, and add the node to the visited set. This is repeated until either the program has run out of nodes or has found the solution. In the case that the f value are the same, I simply ensured that the program, as defined by the loop, explored left, right, up, down repectively.

7 Problem 7

For the testing, I only tested out code relevant to this week's assignment, thus functions like setState are assumed (and tested previously) to work. For this week, I used different states (some from the textbook and some from scrambleStates) to test out the different heuristics and the A* algorithm. For the scrambleState ones, I often compare to BFS as it is guranteed to give me the optimal number of moves (with a very suboptimal runtime).