Assignment 1: Search
CSE 352 Spring 2021

8-Puzzle or 15-Puzzle: A* and RBFS with Hamming and Manhattan Distance Heuristics

1. Problem Formulation
Describe how you defined the problem in the TileProblem class.

I defined the TileProblem in terms of state, actions, transition functions, and goal test. A TileProblem object has the following variables: size, state, action. Size is the size of the puzzle (N=3 for 8-puzzle and N=4 for 15-puzzle). The state is the state of the problem, which is a 2D array that represents the puzzle board's configuration. Action is an array of legal actions that the state can do next, which can be: left, right up, down.

The transition functions are: state_actions(self, state) which returns an array of valid actions the state can do next. change_state(self, state, action) takes in a state and action and then returns a new TileProblem.

The new TileProblem's state is a new state which is calculated from the state argument and action argument that was passed in the function (new state = state argument transitioning with the action argument). Creating the new TileProblem uses a helper function, copy(self, state). These functions help transition one state to the next.

2. Heuristics
Describe the two heuristics you used for A*. Show why they are consistent and why h_1 dominates h_2.

The two heuristics I used for A* are the Manhattan distance and Hamming distance. h_1= Manhattan distance, h_2= Hamming distance

Manhattan distance is a summation of each tile's distance to the correct position. A tile's distance value is calculated by adding the number of moves needed for the tile to be moved in the correct position. Each move costs 1. Hamming distance is the total number of displaced tiles.

A heuristic is consistent "if its estimate is always less than or equal to the estimated distance from any neighboring vertex to the goal, plus the cost of reaching that neighbor. Formally, for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P." Source used: https://en.wikipedia.org/wiki/Consistent_heuristic

"Consistency: h(n) ≤ h(n') + c(n, a, n') i.e, estimated cost of node ≤ estimated cost of successor + step cost. Consistency => admissibility" Source: Lecture Slides

These heuristics are consistent because the estimated cost of the node (to the goal) is less than or equal to the estimated cost of the successor (to the goal). The cost of each move (step cost) = 1. All nodes O_i∈OPT. OPT is the optimal path.

For Hamming distance, h(n) is the number of displaced tiles. Each move can only affect one block. h(O_i) ≤ h(O_{i+1}) + 1. It is consistent. h(n) ≤ h(n') + c(n, a, n') holds (estimated cost of node ≤ estimated cost of successor + step cost)

For Manhattan distance, h(n) is the summation of each tile's distance to the correct position. Each move will change a tile's distance by 1 and you are only able to move vertically or horizontally. The Manhattan distance between two points will always be the same. h(O_i) ≤ h(O_{i+1}) + 1. It is consistent. h(n) ≤ h(n') + c(n, a, n') holds (estimated cost of node ≤ estimated cost of successor + step cost)

"If $h_1(n) \geq h_2(n)$ for all n (both admissible) then h_1 dominates h_2 and is better for search." Source used: https://pages.mtu.edu/~nilufer/classes/cs5811/2012-fall/lecture-slides/cs5811-ch03-search-b-informed-v2.pdf

h_1= Manhattan distance, h_2= Hamming distance. h_1 dominates h_2 because $h_1(n) \geq h_2(n)$ for all n (both admissible). For each tile, Hamming distance will calculate whether it is displaced or not (maximum cost for each tile is 1). For each tile, Manhattan distance will calculate how far away the tile is from the correct position (maximum cost for each tile is greater than 1). For Hamming distance, a tile that is far away from its correct position will cost the same as a tile that is 1 distance away from the correct position.

3. Memory issue with A*

Describe the two heuristics you used for A*. Show why they are consistent and why h_1 dominates h_2. Why does this happen? How much memory do you need to solve the 15-puzzle?

The memory issue that happens when running A* is that the memory usage grows exponentially. This happens because A* has a space complexity of $O(b^d)$ since it stores all the generated nodes (during the search) in memory. b = branch factor, d = solution depth. With no duplicate nodes, the number of nodes would be $b + b^2 + b^3 + \ldots + b^d$ which is $O(b^d)$

The amount of memory needed to solve the 15-puzzle is $O(b^d)$. The branching factor, b, would be a maximum value of 4 as a puzzle can move left, right, up, down from one state to another. The solution depth is d.

4. Memory-bounded algorithm

Describe your memory bounded search algorithm. How does this address the memory issue with A* graph search. Is this algorithm complete? Is it optimal? Give a brief complexity analysis. This analysis doesn't have to be rigorous but clear enough and correct.

The memory bounded search algorithm, Recursive Best-First Search (RBFS) is a recursive function that uses linear space. RBFS runs similarly to the recursive depth-first search algorithm. Depth-first search explores a branch as far as possible (indefinitely) before it backtracking to another branch. Unlike depth-first search, RBFS keeps track of an f-limit variable. The f-limit keeps track of the best alternative path available from any ancestor of the current node. Keeping track of the f-limit limits recursion.

If the current node exceeds the f-limit, the algorithm returns false because there exists an alternative path with a lower cost. Then the algorithm unwinds back to the alternative path. As the recursion unwinds, RBFS updates the f-value (f(n)) of each node along the path with the best f-value of its children. In this way, it can decide whether it's worth re-expanding a forgotten subtree.

Sources used:
https://pages.mtu.edu/~nilufer/classes/cs5811/2012-fall/lecture-slides/cs5811-ch03-search-b-informed-v2.pdf
http://mas.cs.umass.edu/classes/cs683/lectures-2010/Lec6_Search5-F2010-4up.pdf

RBFS addresses the memory issue with the A* graph search, it uses less memory usage. A* space complexity is exponential while RBFS space complexity is linear. This is because the algorithm keeps track of the f-limit, the f-value of the best alternative path. When the recursion unwinds (when the current subtree no longer looks the best, checking the f-limit, there exists an alternative path at a lower cost), it forgets the sub-tree to save space. RBFS only needs memory space for the current search path and the sibling nodes along the path and the f-limit.

Source used:
https://www.eecs.yorku.ca/course_archive/2013-14/F/3401/slides/15b-RBFS.pdf

RBFS is complete. It will always find a solution if one exists. In the algorithm, we will explore the node's successors. The actions are a finite set.

RBFS is not optimal. This is because, for the 15-puzzle problem, RBFS may return a solution that is not optimal. For example, the returned solution of the puzzle can contain the following action sequences: L,R,L or U,R,D,L. These action sequences would prove that the solution is not optimal because it will have the same state as it was before it has moved in that sequence, making unnecessary action moves.

Complexity analysis: Time complexity is $O(b^d)$. b = branch factor, d = solution depth. Space complexity is $O(bd)$. b = branch factor, d = solution depth.

Time complexity is $O(b^d)$. In the worst-case, the algorithm may have to explore all nodes. This can also be represented as $O(b^m)$ where m is the max depth.
The space complexity will be$O(bd)$. O(longest path length searched). In the worst-case, the algorithm would need to track all nodes. The algorithm would have to go through all possible nodes/states (traverse the entire graph without repetition). This can also be represented as $O(bm)$ where m is the max depth.

5. Performance

A* with Manhattan Distance

File	States Explored	Time (ms)	Depth	Solution
puzzle1.txt	3	0.998000	2	R,R
puzzle2.txt	8	1.039000	4	U,R,D,D
puzzle3.txt	16	0.999000	8	L,U,U,R,D,L,D,R
puzzle4.txt	14	0.964000	6	L,L,D,R,R,R
puzzle5.txt	32	3.010000	10	R,D,R,U,L,D,R, R,D,D

A* with Hamming Distance

File	States Explored	Time (ms)	Depth	Solution
puzzle1.txt	3	0.997000	2	R,R
puzzle2.txt	5	0.998000	4	U,R,D,D
puzzle3.txt	27	1.994000	8	L,U,U,R,D,L,D,R
puzzle4.txt	7	0.994000	6	L,L,D,R,R,R
puzzle5.txt	29	2.028000	10	R,D,R,U,L,D,R, R,D,D

RBFS with Manhattan Distance

File	States Explored	Time (ms)	Depth	Solution

puzzle1.txt	3	0.987000	2	R,R
puzzle2.txt	5	0.955000	4	U,R,D,D
puzzle3.txt	27	1.032000	8	L,U,U,R,D,L,D,R
puzzle4.txt	15	1.953000	6	L,L,D,R,R,R
puzzle5.txt	24	1.994000	10	R,D,R,U,L,D,R, R,D,D

RBFS with Hamming Distance

File	States Explored	Time (ms)	Depth	Solution
puzzle1.txt	3	0.951000	2	R,R
puzzle2.txt	5	1.041000	4	U,R,D,D
puzzle3.txt	37	1.043000	8	L,U,U,R,D,L,D,R
puzzle4.txt	7	0.998000	6	L,L,D,R,R,R
puzzle5.txt	31	1.007000	10	R,D,R,U,L,D,R, R,D,D