Project 1

8-Puzzle

CS 420 - 01

Kristin Adachi

knadachi@cpp.edu

August 14, 2017

Project Specification:

This project focused on the implementation of the A* search algorithm that is commonly used for pathfinding and graph traversal. In our case, we used this algorithm to find the solution for solvable 8-puzzles. We were required to use two different heuristic functions for the search:

1. h(1) = the number of misplaced tiles
2. h(2) = the sum of the distances of the tiles from their goal positions

My program is able to solve a user defined 8-puzzle or a randomly generated puzzle. A step-by-step solution is displayed for the user. The solvability and validity of the puzzle is checked before attempting to find a solution. Throughout the process, we were required to compare the heuristic functions by their efficiency.

My Approach:

I used five different classes for my program: `Puzzle`, `BoardNode`, `AStarSearch`, `FCompare`, and `UI`. `Puzzle` contains the main method and `UI` handles all the user interface for my program. `BoardNode` represents one state of the 8-puzzle board and keeps track of its state, parent, f(n) value, g(n) value, and h(n) value. This class also calculates h(n) depending on the heuristic used. `AStarSearch` is where the A* search algorithm is implemented and where the 8-puzzle solution is produced.

I used a tree structure with graph search to implement the A* search algorithm. Child nodes were generated from each state which was expanded based on the smallest f(n) in the frontier. I used a priority queue to implement my frontier. It used my `FCompare` class to compare and order the states by their f(n) value. This f(n) value was composed of g(n) which represented the depth and h(n) which was calculated depending on the heuristic used. To keep track of already visited nodes, a hash set was used to represent the explored set. Rather than storing 2D arrays of the puzzle board, states were stored in the hash set by using a generated key. I used a method to convert the 2D array to a String and stored that instead. With this implementation, I was quickly able to check if a node had already been expanded. I also used an ArrayList to store the solution path of the tree.

Analysis:

To compare the two heuristic functions, I generated 100 solvable puzzles and recorded the number of nodes generated in the tree as well as the time it took to run for each heuristic (as shown in the following table). The nodes represent the size of the tree and the time was recorded in nanoseconds. I recorded the data by depth ranges and averaged the number of nodes and times. The number of nodes accounted for each range is also displayed.

For smaller depths, I noticed that both heuristics generated about the same amount of nodes. However, the A* search with h(1) performed slower than the A* search with h(2). As the depth increased, A*(h_1) continued to generate more nodes and take much longer than A*(h_2) to run. The difference between the two heuristics became more and more prominent with each depth increase.

In conclusion, the A* search algorithm with the second heuristic (the sum of the distances of the tiles from their goal positions) was much more efficient than the first heuristic especially as the depth increased.

Average Nodes Generated and Times

Depth (d)	***A*(h_1)***		***A*(h_2)***		***Count***
	Nodes	*Time (ns)*	*Nodes*	*Time (ns)*	
< 6	8	232524	8	175909	16
6 - 10	38	788300	20	381974	18
11 - 15	268	3391940	83	877791	15
16 - 20	2967	13455211	441	1932721	17
21 - 25	23327	54800462	2188	6380589	18
> 25	99433	193892222	5777	12649242	16

This table represents the averages of the data found after running the program several times.