Solving Problems by Searching

The 8-puzzle problem is a game involving a 3 x 3 grid with eight tiles, each labeled with a number between one through eight. Using the single blank square to rearrange the tiles, the goal is to arrange the tiles in the order (0 1 2), (3 4 5), (6 7 8), with the zero representing the blank square.

The code I developed generates random valid boards and utilizes informed search strategies to find solutions given a valid board. The first step in my tree search function is to determine the index of the zero in the board, which functions as the blank. Information on how I did this can be found here: Returning the Index of an Element in a 2D Array - Stack Overflow. The location of the blank is kept track of throughout the rest of the search process by methods provided in the class named state. The search process requires a closed list to store states that have already been explored and a frontier which holds nodes that have been generated but not explored. The closed list is implemented as a set and the frontier as a priority queue. The value considered when popping a node off the frontier is the f value, which is calculated as f(n) = g(n) + h(n). Here, g(n) is the number of moves it has taken to get to the current node, and h(n), called the heuristic, is a calculated guess on how many more moves are required to get to the goal state. If two nodes have the same f value, then the most recently generated node will be chosen first. This choice does not impact the optimality of the solution and is made in order to provide a predictable order of expansion. After a node has been popped off the frontier and checked against the closed list, all possible states that can be generated from that state (maximum of four by shifting the blank square up, down, left, or right) are added to the frontier. The program continues this process, starting with popping a node off the frontier, until the frontier is empty (a failure), or the goal state is reached. When the goal is found, the path is returned and reversed before each board state leading to and including the solution is displayed. An example of the program's output is given in the figure on page three.

The user is given the option of using one of four heuristics to find the solution. The first option is simply h(n) = 0. In this scenario, the program is given no additional information about how far certain states are from the goal and must make do with the current path length (the number of previous states). This is called a Uniform Cost Search. The second option is to use a heuristic calculated by summing the number of tiles that are displaced. In other words, a value of one is added to h(n) for each tile that is not correctly placed given the desired tile order: (0 1 2), (3 4 5), (6 7 8). In my program, this heuristic value is first calculated by creating a 3 x 3 list with values of one where items in the board do not match the desired position and zero where the board tiles do match. The sum of this 2-dimensional list is then found. Code for calculating this value can be found here: Summing 2D Arrays - Stack Overflow. The third heuristic option is similar to the second but takes more detail into account during the calculation. The third heuristic is calculated by determining a tile's distance from its desired position (given that the tile can only be moved up, down, left, or right). For each tile displaced, this value is added to the total sum and used as the estimated number of moves away from the goal state. This is called the Manhattan distance. The fourth and final heuristic is calculated by adding one to the Manhattan distance for each title that is misplaced and not directly next to the blank (zero).

For each heuristic, I calculated general performance measures to compare the efficiency of the heuristics. These measures are: one, the total number of nodes expanded, two, the

maximum number of nodes stored in memory (this includes the closed list and the frontier), three, the depth of the optimal solution, and, four, the approximate effective branching factor (calculated as b = (nodes_in_memory)^(1/depth_of_solution). For each heuristic, I ran the program one hundred times with one hundred unique random boards. Then, I calculated the minimum, median, mean, maximum, and standard deviation of the heuristics' performance measures. The resultant statistics are in the table below.

Heuristic 1	Minimum	Median	Mean	Maximum	Standard Deviation
Nodes Expanded	10	24153	40165.2	160912	40233.23567
Max Nodes Stored	23	46018	67778.18	209321	60262.3213
Depth of Solution	2	18	17.72	26	4.449674
Effective Branching Factor	1.6007303	1.8013660	1.8505703	4.7958315	0.3219094
Heuristic 2	Minimum	Median	Mean	Maximum	Standard Deviation
Nodes Expanded	2	1298.5	3457.03	36398	5685.758
Max Nodes Stored	7	2646.5	6897.81	68606	10968.42
Depth of Solution	2	18	17.72	26	4.449674
Effective Branching Factor	1.49027	1.542007	1.555011	2.645751	0.11476
Heuristic 2	Minimum	Median	Mean	Maximum	Standard Deviation
Nodes Expanded	2	275.5	547.31	4623	730.4035
Max Nodes Stored	7	556	1084.85	8774	1408.671
Depth of Solution	2	18	17.72	26	4.449674
Effective Branching Factor	1.306421	1.420996	1.438473	2.645751	0.134586
Heuristic 3	Minimum	Median	Mean	Maximum	Standard Deviation
Nodes Expanded	2	272	482.48	4036	680.7507
Max Nodes Stored	7	557	962.15	7844	1326.159
Depth of Solution	2	18	18.08	28	4.739006
Effective Branching Factor	1.284279	1.39207	1.423407	2.645751	0.140813

The paths found using the first three heuristics are all optimal. This can be seen in the identical values in the row "Depth of Solution" for all three heuristics. While the data on the other statistics varies greatly, the depth of the solution for these three heuristics remains consistent. This is an indication that the heuristic is admissible, or, in other words, it never overestimates the number of moves it will take to reach the goal state. There is a significant improvement in the total number of nodes expanded, the maximum number of nodes stored, and the effective branching factor between the first and second and between the second and third heuristics. The pattern of consistent solution depths is broken by the fourth heuristic, which indicates that the fourth heuristic is not admissible. While the fourth heuristic does not guarantee an optimal solution, there is a slight overall improvement in the number of nodes

expanded, the maximum number of nodes stored, and the effective branching factor compared to the third heuristic (the Manhattan distance).

```
V=2
N=5
d=2
b=2.23606797749979

1 2 0
3 4 5
6 7 8

1 0 2
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8
```

Figure 1: example output including the relevant performance measures above the boards.
Command: cat OLA1-input.txt | ./random-board.py 50 4 | ./a-star.py 2