Matthew Ragoza
mtr22@pitt.edu

# Assignment 2

ISSP 2160
September 22, 2021

## Problem 1. Evaluation function-driven search traversals

a. The order of nodes expanded by the uniform cost search algorithm are:

$S \to A \to D \to E \to S \to B \to D \to D \to S \to A \to A \to D \to E \to E \to F \to A \to A \to B \to C \to D \to D \to D \to E \to E \to S \to S \to S \to S \to A \to A \to B \to B \to D \to D \to D \to D \to E \to E \to E \to E \to F \to F \to G$

The total cost of the found path is 14, which is the optimal path.

b. The order of nodes expanded by the greedy search algorithm are:

$S \to D \to E \to F \to C \to G$

The total cost of the found path is 18, which is not the optimal path.

c. The order of nodes expanded by the A* search algorithm are:

$S \to A \to D \to E \to F \to B \to C \to G$

The total cost of the found path is 14, which is the optimal path.

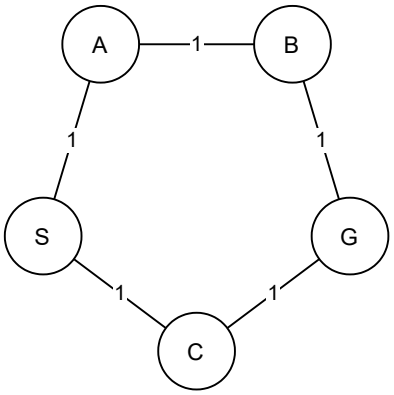
## Problem 2. Bi-directional A* search proofs

a. Bi-directional A* search consists of two independent A* search algorithms running concurrently, one starting at the initial state and searching for the goal state (here referred to as the *forward process*), while the other starts at the goal state and searches for the initial state (the *backward process*). If the forward and backward processes visit the same state, their respective paths to that state can be merged into a solution path. Therefore, we can analyze the bi-directional A* algorithm as a relaxation of the goal condition in a standard A* search problem, such that the goal condition is met 1) if the goal state is visited by the forward process, 2) if the initial state is visited by the backward process, or 3) if any intermediate state is visited by both the forward and backward processes. To demonstrate completeness, it is sufficient to show that any of these three conditions are necessarily satisfied by the algorithm.

Let us now consider the worst-case scenario, where no intermediate states are visited by both processes, but a solution path still exists. Since we know that A* search is complete on its own, and both the forward and backward processes are running A* independently, either the forward process will reach the goal state before the backward process completes, or the backward process will reach the initial state before the forward process completes. Either of these two conditions results in a solution path. Thus, a solution path is found even in the worst case scenario that the forward and backward processes visit no states in common. In the alternative scenario, where they do visit an intermediate state in common, their paths to this state can be merged to form a solution. Thus, one of the three possible goal conditions are necessarily satisfied if a solution path exists, so bi-directional A* search is complete.

b. Continuing in a similar vein as above, we can now investigate the optimality of bi-directional A* by viewing it as a relaxation of standard A* search. Since there are three possible ways to satisfy the goal condition, the question of whether the found solution is optimal can be broken down similarly. Suppose that a solution is found by the forward process reaching the goal state. Since the forward process is running A* search, the solution found will be identical to the one found by simply running A* search normally. Since A* search is optimal, the solution found by the forward process reaching the goal state will be optimal. In the second scenario, where a solution is found by the backward process reaching the initial state, the same analysis holds. That is, since the backward process is running A* search, which is optimal, if the backward process finds a solution by reaching the initial state than that solution will be optimal.

When a solution is found by the forward process and the backward process visiting the same intermediate node, it is possible to return a sub-optimal solution. This is best seen with a specific example, shown in Figure 1. Let $S$ be the initial state, $G$ be the goal state, and let $h_f$ and $h_b$ be admissible heuristic functions for the forward and backward processes, with values listed in Figure 1b. Starting with the forward process, we expand $S$ to yield successors $A$ and $C$, with evaluated values of 1 and 2 respectively. The backward process then expands $G$ to produce predecessors $B$ and $C$, with values 1 and 2. Switching again, the forward process expands $A$ to generate $B$ and $S$, both valued at 2. The backward process then visits $B$, which is tied for the minimal value node in the forward process queue. A possible tie-breaking rule like alphabetical order would then have the forward process visit $B$ next too, satisfying the goal condition and producing the merged solution path $S \to A \to B \to G$, which is sub-optimal. This demonstrates that bi-directional A* search with admissible heuristics is not optimal.




(a) Graph search space.

| State | $h_f$ | $h_b$ |
|---|---|---|
| $S$ | 0 | 0 |
| $A$ | 0 | 0 |
| $B$ | 0 | 0 |
| $C$ | 1 | 1 |
| $G$ | 0 | 0 |

(b) Heuristic function values for the forward ($h_f$) and backward ($h_b$) search processes.

Figure 1: An example where bi-directional A* search with admissible heuristics would not necessarily find the optimal path.

## Problem 3. Heuristic search for Puzzle 8

a. Implemented `main3a.py`.

b. Implemented `main3b.py`.

c. Implemented `heuristic1.py` and used in `main3c.py`.

d. Implemented `heuristic2.py` and used in `main3d.py`.

e. First we analyze the effect of eliminating state repeats from uniform cost search (UCS), which was the difference between `main3a.py` and `main3b.py`. Both algorithms found the optimal solution for the first three examples, but examples 4 and 5 were only solved by UCS after eliminating state repeats. The `main3a.py` method generated and expanded the most nodes, and had the largest queue length, of any of the search methods. Example 3 was the hardest problem that it solved, and it generated and expanded over 10 times as many nodes as `main3b.py` in doing so, seen in Figure 4. This difference consisted entirely of nodes in paths to states for which there was a known lower-cost path, so it was wasted computation.

   The next algorithmic improvement was introducing an admissible heuristic as a term in the evaluation function, which turns UCS into A* search. In `main3c.py`, which used the misplaced tile heuristic, this greatly improved the performance on every example. On the first three examples, the number of expanded nodes was reduced to one less than the optimal solution length, which is the minimum possible number of expanded nodes while reaching a solution (since only the goal node was not expanded). On the hardest two examples, there were over 20 times fewer nodes expanded and generated using `main3c.py` than `main3b.py`.

   The final comparison is between the two different heuristics for A* search, the second of which was Manhattan distance, used in `main3d.py`. There was no noticeable improvement in the first three examples, since both A* search methods solved these as efficiently as possible in terms of number of expanded nodes. On Example 4, `main3d.py` found the optimal solution of length 11 in just 12 expansions, which is nearly perfect and twice as efficient as `main3c.py`. It also performed better on Example 5, seen in the six fold decrease in number of expanded nodes, generated nodes, and maximum queue length in Figure 6.

   Because of its perfect performance on Examples 1-3, near-perfect performance on Example 4, and significant improvement over every other method on Example 5, it is clear that `main3d.py` was the best search algorithm. This is due to the combination of A* search, which greatly outperformed UCS, with the slight edge granted by using the superior Manhattan distance instead of the misplaced tile heuristic. It is notable that we did not consider A* search without checking for state repeats, which raises the question of whether this would still work (i.e. be optimal). A* search expands nodes

according to the sum of their current path cost and their heuristic value. Any repeated states would have the same heuristic value, and any nodes on sub-optimal paths to repeated states would have higher path cost than the optimal path. Therefore, they would have higher total evaluated cost, and have lower priority in the queue. As a result, A* search would still find the optimal solution, if it exists, without checking for state repeats.

Between the two tested heuristics, I would suggest using Manhattan distance, which has superior performance because it *dominates* over the misplaced tile heuristic–that is, it provides a tighter lower bound on the remaining path cost to the solution. This can be seen by the fact that the Manhattan distance of a misplaced tile is always at least one, while the misplaced tile heuristic just assigns 1 to any tile that is misplaced, no matter the distance. There is no tile configuration where the total Manhattan distance is less than the number of misplaced tiles, so Manhattan distance always provides the same or higher value, and is therefore dominant. On a related note, a heuristic $h_3$ which averages the values of Manhattan distance and misplaced tile heuristic would be admissible, since its value would never be more than Manhattan distance, which is admissible. Manhattan distance dominates over the misplaced tile heuristic, so the average of the two values would always be less than or equal to Manhattan distance. As a consequence, Manhattan distance would dominate over $h_3$ as well.

| | # expanded | # generated | Max. queue length | Solution length |
|---|---|---|---|---|
| `main3a.py` | 29 | 81 | 53 | 4 |
| `main3b.py` | 16 | 44 | 22 | 4 |
| `main3c.py` | 3 | 10 | 8 | 4 |
| `main3d.py` | 3 | 10 | 8 | 4 |

Figure 2: Search statistics for Example 1.

| | # expanded | # generated | Max. queue length | Solution length |
|---|---|---|---|---|
| `main3a.py` | 500 | 1382 | 883 | 7 |
| `main3b.py` | 84 | 234 | 105 | 7 |
| `main3c.py` | 6 | 15 | 10 | 7 |
| `main3d.py` | 6 | 15 | 10 | 7 |

Figure 3: Search statistics for Example 2.

| | # expanded | # generated | Max. queue length | Solution length |
|---|---|---|---|---|
| `main3a.py` | 1962 | 5622 | 3661 | 9 |
| `main3b.py` | 173 | 490 | 216 | 9 |
| `main3c.py` | 8 | 22 | 15 | 9 |
| `main3d.py` | 8 | 22 | 15 | 9 |

Figure 4: Search statistics for Example 3.

| | # expanded | # generated | Max. queue length | Solution length |
|---|---|---|---|---|
| `main3a.py` | | | | |
| `main3b.py` | 723 | 1950 | 775 | 11 |
| `main3c.py` | 25 | 69 | 38 | 11 |
| `main3d.py` | 12 | 34 | 21 | 11 |

Figure 5: Search statistics for Example 4.

| | # expanded | # generated | Max. queue length | Solution length |
|---|---|---|---|---|
| `main3a.py` | | | | |
| `main3b.py` | 31591 | 84812 | 30611 | 19 |
| `main3c.py` | 1465 | 3989 | 1576 | 19 |
| `main3d.py` | 252 | 676 | 262 | 19 |

Figure 6: Search statistics for Example 5.