McGill University

COMP 424: Artificial Intelligence

Felix Simard (260865674)

February 8th, 2021

Assignment 1

**Question 1: Six-Puzzle**

1. Show the solution path (i.e., the sequence of puzzle states from the initial to the goal state) found by each of the following algorithms, assuming transitions have unit cost. You must ensure that puzzle states that have been explored are not added to the search queue. Given multiple states to explore that are otherwise equivalent in priority, the algorithm should prefer the state that involves moving a lower-numbered piece.

Breadth First Search (BFS)

A picture containing text

Description automatically generated

Uniform Cost Search (UCS)

Note: UCS with a cost of 1 ends up being the same as running BFS.

Text

Description automatically generated with medium confidence

Depth First Search (DFS)

Since this algorithm visits a lot more states the others, we will represent the board configurations as a list reading the board values from top-left to top-right, bottom-left to bottom-right.

For example: our initial state:

Text

Description automatically generated with medium confidence would transalte to the list: [1, 4, 2, 5, 3, 0]

[1, 4, 0, 5, 3, 2] -> [1, 0, 4, 5, 3, 2] -> [0, 1, 4, 5, 3, 2] -> [5, 1, 4, 0, 3, 2] -> [5, 1, 4, 3, 0, 2] -> [5, 0, 4, 3, 1, 2] -> [5, 4, 0, 3, 1, 2] -> [5, 4, 2, 3, 1, 0] -> [5, 4, 2, 3, 0, 1] -> [5, 4, 2, 0, 3, 1] -> [0, 4, 2, 5, 3, 1] -> [4, 0, 2, 5, 3, 1] -> [4, 2, 0, 5, 3, 1] -> [4, 2, 1, 5, 3, 0] -> [4, 2, 1, 5, 0, 3] -> [4, 0, 1, 5, 2, 3] -> [4, 1, 0, 5, 2, 3] -> [4, 1, 3, 5, 2, 0] -> [4, 1, 3, 5, 0, 2] -> [4, 0, 3, 5, 1, 2] -> [4, 3, 0, 5, 1, 2] -> [4, 3, 2, 5, 1, 0] -> [4, 3, 2, 5, 0, 1] -> [4, 3, 2, 0, 5, 1] -> [0, 3, 2, 4, 5, 1] -> [3, 0, 2, 4, 5, 1] -> [3, 2, 0, 4, 5, 1] -> [3, 2, 1, 4, 5, 0] -> [3, 2, 1, 4, 0, 5] -> [3, 0, 1, 4, 2, 5] -> [3, 1, 0, 4, 2, 5] -> [3, 1, 5, 4, 2, 0] -> [3, 1, 5, 4, 0, 2] -> [3, 0, 5, 4, 1, 2] -> [0, 3, 5, 4, 1, 2] -> [4, 3, 5, 0, 1, 2] -> [4, 3, 5, 1, 0, 2] -> [4, 3, 5, 1, 2, 0] -> [4, 3, 0, 1, 2, 5] -> [4, 0, 3, 1, 2, 5] -> [4, 2, 3, 1, 0, 5] -> [4, 2, 3, 0, 1, 5] -> [0, 2, 3, 4, 1, 5] -> [2, 0, 3, 4, 1, 5] -> [2, 1, 3, 4, 0, 5] -> [2, 1, 3, 0, 4, 5] -> [0, 1, 3, 2, 4, 5] -> [1, 0, 3, 2, 4, 5] -> [1, 3, 0, 2, 4, 5] -> [1, 3, 5, 2, 4, 0] -> [1, 3, 5, 2, 0, 4] -> [1, 3, 5, 0, 2, 4] -> [0, 3, 5, 1, 2, 4] -> [3, 0, 5, 1, 2, 4] -> [3, 2, 5, 1, 0, 4] -> [3, 2, 5, 0, 1, 4] -> [0, 2, 5, 3, 1, 4] -> [2, 0, 5, 3, 1, 4] -> [2, 1, 5, 3, 0, 4] -> [2, 1, 5, 0, 3, 4] -> [0, 1, 5, 2, 3, 4] -> [1, 0, 5, 2, 3, 4] -> [1, 5, 0, 2, 3, 4] -> [1, 5, 4, 2, 3, 0] -> [1, 5, 4, 2, 0, 3] -> [1, 5, 4, 0, 2, 3] -> [0, 5, 4, 1, 2, 3] -> [5, 0, 4, 1, 2, 3] -> [5, 2, 4, 1, 0, 3] -> [5, 2, 4, 0, 1, 3] -> [0, 2, 4, 5, 1, 3] -> [2, 0, 4, 5, 1, 3] -> [2, 1, 4, 5, 0, 3] -> [2, 1, 4, 5, 3, 0] -> [2, 1, 0, 5, 3, 4] -> [2, 0, 1, 5, 3, 4] -> [0, 2, 1, 5, 3, 4] -> [5, 2, 1, 0, 3, 4] -> [5, 2, 1, 3, 0, 4] -> [5, 0, 1, 3, 2, 4] -> [0, 5, 1, 3, 2, 4] -> [3, 5, 1, 0, 2, 4] -> [3, 5, 1, 2, 0, 4] -> [3, 5, 1, 2, 4, 0] -> [3, 5, 0, 2, 4, 1] -> [3, 0, 5, 2, 4, 1] -> [0, 3, 5, 2, 4, 1] -> [2, 3, 5, 0, 4, 1] -> [2, 3, 5, 4, 0, 1] -> [2, 0, 5, 4, 3, 1] -> [0, 2, 5, 4, 3, 1] -> [4, 2, 5, 0, 3, 1] -> [4, 2, 5, 3, 0, 1] -> [4, 2, 5, 3, 1, 0] -> [4, 2, 0, 3, 1, 5] -> [4, 0, 2, 3, 1, 5] -> [4, 1, 2, 3, 0, 5] -> [4, 1, 2, 0, 3, 5] -> [0, 1, 2, 4, 3, 5] -> [1, 0, 2, 4, 3, 5] -> [1, 2, 0, 4, 3, 5] -> [1, 2, 5, 4, 3, 0] -> [1, 2, 5, 4, 0, 3] -> [1, 0, 5, 4, 2, 3] -> [0, 1, 5, 4, 2, 3] -> [4, 1, 5, 0, 2, 3] -> [4, 1, 5, 2, 0, 3] -> [4, 0, 5, 2, 1, 3] -> [0, 4, 5, 2, 1, 3] -> [2, 4, 5, 0, 1, 3] -> [2, 4, 5, 1, 0, 3] -> [2, 4, 5, 1, 3, 0] -> [2, 4, 0, 1, 3, 5] -> [2, 0, 4, 1, 3, 5] -> [0, 2, 4, 1, 3, 5] -> [1, 2, 4, 0, 3, 5] -> [1, 2, 4, 3, 0, 5] -> [1, 0, 4, 3, 2, 5] -> [0, 1, 4, 3, 2, 5] -> [3, 1, 4, 0, 2, 5] -> [3, 1, 4, 2, 0, 5] -> [3, 0, 4, 2, 1, 5] -> [0, 3, 4, 2, 1, 5] -> [2, 3, 4, 0, 1, 5] -> [2, 3, 4, 1, 0, 5] -> [2, 3, 4, 1, 5, 0] -> [2, 3, 0, 1, 5, 4] -> [2, 0, 3, 1, 5, 4] -> [0, 2, 3, 1, 5, 4] -> [1, 2, 3, 0, 5, 4] -> [1, 2, 3, 5, 0, 4] -> [1, 0, 3, 5, 2, 4] -> [0, 1, 3, 5, 2, 4] -> [5, 1, 3, 0, 2, 4] -> [5, 1, 3, 2, 0, 4] -> [5, 0, 3, 2, 1, 4] -> [5, 3, 0, 2, 1, 4] -> [5, 3, 4, 2, 1, 0] -> [5, 3, 4, 2, 0, 1] -> [5, 3, 4, 0, 2, 1] -> [0, 3, 4, 5, 2, 1] -> [3, 0, 4, 5, 2, 1] -> [3, 2, 4, 5, 0, 1] -> [3, 2, 4, 5, 1, 0] -> [3, 2, 0, 5, 1, 4] -> [3, 0, 2, 5, 1, 4] -> [3, 1, 2, 5, 0, 4] -> [3, 1, 2, 5, 4, 0] -> [3, 1, 0, 5, 4, 2] -> [3, 0, 1, 5, 4, 2] -> [0, 3, 1, 5, 4, 2] -> [5, 3, 1, 0, 4, 2] -> [5, 3, 1, 4, 0, 2] -> [5, 3, 1, 4, 2, 0] -> [5, 3, 0, 4, 2, 1] -> [5, 0, 3, 4, 2, 1] -> [5, 2, 3, 4, 0, 1] -> [5, 2, 3, 4, 1, 0] -> [5, 2, 0, 4, 1, 3] -> [5, 0, 2, 4, 1, 3] -> [5, 1, 2, 4, 0, 3] -> [5, 1, 2, 0, 4, 3] -> [0, 1, 2, 5, 4, 3] -> Goal state reached in 163 moves

Iterative Deepening Search

The solution path is also basically identical to BFS since we increment by a depth of 1 at each iteration of depth-limited search.

Text

Description automatically generated

1. Suppose now that transitions have differing costs. In particular, the cost of a transition is equal to the number of the piece that is moved (e.g., moving the “4” costs 4). If we employ the Manhattan distance heuristic for the original unit cost version of the eight-puzzle presented in class (Lecture 4, slide 11, h2), would this heuristic still be an admissible heuristic for A\* search in the new variant? Justify your answer.

Let the cost of a transition be equal to the number of the piece that is moved.

We defined an “admissible heuristic” for A\* search as being an optimistic heuristic which always gives an underestimate of the true cost for all paths from n to a goal state.

As seen in the lecture slides, the Manhattan distance heuristic estimates the true cost of getting to the goal state by summing the distances of each tile from the current position to where they should end up being in the goal state.

With our variant cost based on the piece number scheme, the Manhattan distance heuristic value is still independent of the cost of moving a piece. Therefore, the actual cost would always be greater than, or equal to 1.

Hence, our modified scheme would still provide an admissible heuristic as it would be optimistic and always underestimate the true cost to the actual shortest paths, for all states.

1. Design an admissible heuristic that dominates the heuristic from part b, under the same cost scheme as part b.

An admissible heuristic which would dominate the heuristic from part b:

Heuristic: using our cost variant approach as shown in b), one could add each piece’s value to their Manhattan distance. For example, say tile “3” has a Manhattan distance of 2, then its adjusted Manhattan distance using out heuristic would be 2+3. This will provide a more accurate representation of how far away a certain piece is to its desired end position (i.e.: by proportionally weighting pieces that are further away from their goal state).

Why is it admissible and why does it dominate the heuristic in part b)?

Since that the lowest piece has value 1, and by using the newly designed heuristic, we will increment the Manhattan distance of all pieces, then our new heuristic is guaranteed to dominate the previous heuristic. In addition, for a goal state, both heuristics will be equal to zero. As a result, our heuristic will always require less moves than the actual solution which makes it admissible.

Reference (for code inspiration): <https://github.com/speix/8-puzzle-solver>

**Question 2: Search Algorithms**

1. Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, 𝑂(𝑛^2) vs 𝑂(𝑛)).

Recall that Depth First Search (DFS) will traverse nodes until a leaf is reached, adding neighboring nodes to a stack at each node being visited and popping that stack to find the next leaf.

On the other hand, Iterative Deepening Search (IDS) performs a depth-limited search at each depth increment, meaning during the first iteration, the child of the root is visited and on the second iteration, the child of root is visited at depth 2.

Hence, iterative deepening will perform worst if there is a goal state at depth n in a domain where every state has a single successor, n being the number of nodes, and with an incrementing depth factor of 1. In fact, IDS will take 1+2+3+...+n = O (n^2) steps, unlike DFS which will take O (n) steps.

Reference:

<http://web.eecs.utk.edu/~leparker/Courses/CS594-fall04/Homeworks/HW-1-soln.pdf>

<https://cpentalk.com/203/describe-iterative-deepening-search-performs-search-example>

1. Prove each of the following statements, or give a counterexample:
2. *Breadth-first search is a special case of uniform-cost search.*

Indeed, when all step costs are equal (and assume equal to 1), then uniform cost search essentially ends up storing nodes in the same order and priority as in BFS’s queue (inserted in queue based on traversal, no priority arrangement because all costs are 1). In other words, let g(n) be the past cost to a node n from a start state, then g(n) is a multiple of depth n of a node. Hence, given a uniform cost of 1 across all nodes, g(n) = 1 \* (depth of n) which is the optimal cost found by both BFS and uniform-cost search with cost equal 1 for all nodes.

1. *Uniform-cost search is a special case of A\* search.*

Recall for A\* search: f(n) = g(n) + h(n)

For uniform-cost search: f(n) = g(n)

Thus, for a heuristic function, h(n), equal to 0 (uninformed), uniform-cost search will produce the same result as A\* search.

1. *Best-first search is optimal in the case where we have a perfect heuristic (i.e., h(𝑛) = h∗(𝑛), the true cost to the closest goal state).*

False, even with a heuristic function which is exactly right, best-first search is never guaranteed to be optimal. A counterexample:

Diagram

Description automatically generated

In the graph above, green numbers are the actual costs, and red numbers are the exact heuristic function values. A path from S (start) to G (goal) using Best-First Search would give S 🡪 A 🡪 G following the heuristic function while we clearly see that the path S 🡪 B 🡪 C 🡪 G has a lower cost (5 < 6), hence is optimal.

Reference: <https://stackoverflow.com/questions/53311457/is-best-first-search-optimal-and-complete>

1. *Suppose there is a unique optimal solution. Then, A\* search with a perfect heuristic will never expand nodes that are not in the path of the optimal solution.*

Based on slide 32 of lecture 4 (Informed Search), we analyzed the optimal nature of A\* search given an admissible heuristic.

Diagram

Description automatically generated

In fact, we saw that A\* uses a priority to expand nodes in order of increasing f value, meaning A\* would expand node n before node G2. Now, since n was picked arbitrarily in the first place, this tells us that A\* with a perfect heuristic should end up expanding only nodes on the path of the optimal solution.

**Question 3: Travelling Salesman Problem**

As noted, and referenced in my code (see file *tsp.py* under *Q3* code folder), I used the following Python library which was designed specifically to facilitate the setup and analysis of random TSP instances:

Reference: <https://pypi.org/project/opentsp/>

To use this library, simply activate a python virtual environment, make sure the *pip* command line tool is installed on your machine, and run: *pip install opentsp*

Here is a sample TSP instance created by this library:

Diagram

Description automatically generated

*Note: the node values were restricted to the range [0, 1] for answering the questions below.*

For the following sub-questions, please refer to the *tsp.py* files under the *Q3* directory in my code submission folder.

Note: please refer to the comments in the code to see which portion of code is intended to solve which of the sub-questions below. I have attached a partial screenshot of each code section under the questions to pinpoint those sections even more.

(see next page)

1. Solve each TSP exactly by brute-force search over all possible tours. Record and save these costs. In your report, simply state where in your code this is implemented, and state the mean, min, max, and standard deviation of the optimal tour lengths across the 100 instances.

Note: for this question, we were not asked to develop our own custom “brute force” evaluation of the tour lengths. Hence, I used the “brute force” solver that was included in the referenced TSP library.

Text

Description automatically generated

Output:

Text

Description automatically generated

1. As a baseline, generate a random tour for each of the 100 instances. What is the mean, min, max, and standard deviation of the tour lengths found? Report the number of instances for which the random tour happens to be the optimal solution (may be zero).

Text

Description automatically generated

Output:

Graphical user interface, text, application

Description automatically generated

1. Implement and apply hill climbing/greedy local search using the 2-change neighbourhood function described in class on the 100 instances, using the randomly sampled start state from part b). What is the mean, min, max, and standard deviation of the tour lengths found? Also report the number of instances for which the algorithm found the optimal solution.

For this question, the code provided is quite more complex since I implemented the algorithm fully myself. Please refer to this section of the *tsp.py* file under the *Q3* directory for the code details:

Text

Description automatically generated

Overall, the steps which were achieved to answer this question were the following:

1. Generate a random tour for the 100 TSP instances, as in part B. Use as the sampled start state.
2. Construct the list of edges for the above path.
3. Find all 2-change combinations using the edge list from step 2.
4. For each 2-change combination, invert the order of the corresponding vertices.
5. Re-construct the node path sequence after having swapped the order of vertices in step 4.
6. At this point, we have all our possible neighbours for this specific random TSP instance.
7. Perform hill climbing/greedy local search using the list from step 6.
8. Record the lowest cost and the path associated with it.
9. Record if the algorithm indeed found the optimal solution or not.
10. Repeat steps 1-8 for all the 100 TSP instances.
11. Compute the mean, min, max, standard deviation metrics.

We see that using the 2-change neighbourhood function, the hill climbing/greedy local search performed more efficiently than the mere random tour generation algorithm used in the previous question.

Text

Description automatically generated

1. Scale up your experiments! Repeat parts b) and c) using 100 random TSP instances involving 100 cities. Since the number of possible tours is factorial with respect to the number of cities, it may no longer be feasible to compute the costs of the optimal solutions in a reasonable amount of time, so simply omit that part of the table.

To scale up our experiments, setting 100 cities for each TSP instance. I adjusted my code allowing it to terminate in a reasonable amount of time.

In fact, I suspended the computation of the optimal tour for TSP instances with 100 cities. Note this will also restrain us from knowing how many tours from our random algorithms actually computed the optimal tour cost, but that is fine.

For repeating part b), we construct 100 random TSP instances, each comprised of 100 cities, and compute the tour lengths for each, returning our usual metrics as shown below:

Graphical user interface, text, application

Description automatically generated

For repeating part c), we construct 100 random TSP instances (each with 100 cities), for each instance we generate the 2-change neighbours apply hill climbing/greedy local search to find the minimal cost tour our algorithm can find for that specific TSP instance. The metrics are shown again below:

Text

Description automatically generated