Department of Computer Science

CS260 Data Structures

Fall 2021

Programming Assignment 6

(Extra Credit)

1 Regulations

Due Date : 03/18/2021 - 11:59pm
Late Submission: No late submission allowed.
Submission : via Gradescope.
Team : The homework is to be done and turned in individually. No teaming up.
Cheating : All parties involved in cheating get zero from assignment and will be reported to the University.

2 Sliding (or $k^2 - 1$) puzzle using A* algorithm

In programming assignment 5, you were introduced with the problem of solving the sliding puzzle by using Breadth First Search (BFS) algorithm. BFS is an undirected search algorithm that can take long to find the answer to the sliding puzzle problem, as it goes through any possible solution that is adjacent to the start state in arbitrary order. Considering that the search space is $k!$ for a $k \times k$ board, this can take long runtimes for large values of k as well as start states that are several steps ahead of the goal state.

Search time for the goal state can be shortened by adding a priority for the direction of the search to proceed. **A* algorithm** is commonly used as an efficient alternative to find an answer to the sliding puzzle with the minimum number of steps. In this assignment, you will implement A* algorithm to solve the sliding puzzle problem.

3 A* Algorithm

A* (pronounced A-star) is an informed search algorithm, or a best-first search, formulated in terms of weighted graphs starting from a specific starting node of a graph, and find a path to the given goal node having the smallest cost (least distance, shortest time, etc.). A* maintains a tree of paths originating at the start node and extends those paths one edge at a time until its termination criterion is satisfied.

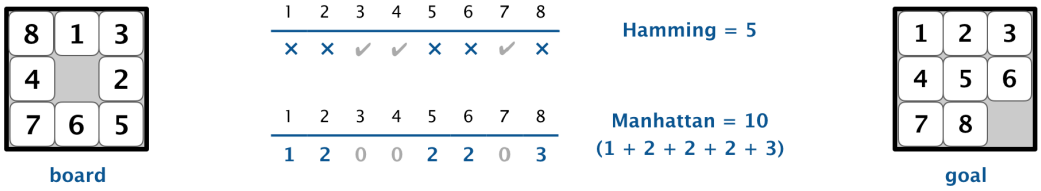
At each iteration of its main loop, A* needs to determine which of its paths to extend based on: i) the cost of the path taken so far and ii)an estimate of the cost required to extend the path towards the goal Thus, A* selects the path that minimizes $f(n) = g(n) + h(n)$, where

- n is the next node on the path,
- g(n) is the cost of the path from the start node to n, and
- h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal.

A **heuristic** is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow. The objective of using an algorithm with heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. This solution may not be optimal but is still valuable because finding it does not require a prohibitively long time.

In the implementation of A* for sliding puzzle, we need to define a heuristic for our search that is a lower bound on the distance of the current state (i.e., board) to the goal state. Then the goal of our search will be to follow the board with minimum distance. Two common measures for 8-puzzle are *the Hamming distance* and *the Manhattan distance*.

The Hamming distance is defined as the number of tiles in the wrong position while the Manhattan distance is defined as the sum of the Manhattan distances (sum of the vertical and horizontal distance) from the tiles to their goal positions. Examples for both distance types are provided in the example below.



4 Roadmap

In this assignment, you are going to write a C program to solve the general $k^2 - 1$ puzzle. Your program will read an initial state for a $k \times k$ table, calculate (preferably minimum number of) steps taking player from initial state to the goal state, and print the solution into an output file.

This is a graph search problem and can be solved using several different methods. In this assignment, you will implement A* to solve the problem. Below are some points to get you started:

- You first need to determine how you are going to implement the graph data structure. Think about pros and cons of both implementations. Adjacency matrix might be easier to implement, but might become very large to fit into memory if your program is tested with large boards. Adjacency list representation would be memory friendly, in the expense of dealing with pointers.
- In your adjacency list representation, consider that you will need to check if a node was already initiated, before generating a new node. Since the search space can be very large depending on the size of the board, this operation will be repeated several times, hence can become a performance bottleneck. Consider using an efficient data structure that can have constant asymptotic search and insert times.
- A* will require a priority queue for storing the nodes to be visited next. Thus, you should implement an efficient priority queue that would allow sub-linear access time for the unvisited node with minimum cost.
- You will need to implement the heuristic function that would estimate the cost of following a certain adjacent node.
- If you are still unsure where to start from, reach out to the instructor and/or TAs sooner rather than later!!!

5 Input/Output Specifications

Input/output specifications for this assignment is identical to programming assignment 5. Please refer to the descriptions there.

5.1 Submission

- Before submitting your code, compare the outputs of your program for various test data with that of the provided executable. Make sure you match the input/output specifications.
- Your submission package **must include** your **source code** and a **makefile** (nor report this time). Put all these material in a folder named with your userid, (example: abc123), compress it into a zip file named as your student id (example: abc123.zip). Submit this zip file through Gradescope.
- Your makefile needs to produce an executable named "solve", once we type "make" in command line.

5.2 Grading

- Assignment is going to be graded out of 100 points.
- There is no self assessment or report for this assignment. All 100 points will be for the test cases of the assignment.
- There are going to be several test cases. If the solution you supplied for a problem is "valid", you will get full credit for this test case, regardless of number of steps in your solution.