Department of Computer Science

CS260
Data Structures

Fall 2021

Programming Assignment 5

1 Regulations

Due Date : 03/11/2021 - 11:59pm
Late Submission: 10% off for each late day, with at most 2 day late submission
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 Puzzle

A sliding puzzle is a puzzle that challenges a player to slide flat pieces along certain routes on a $k \times k$ board to establish a certain end-configuration. The fifteen puzzle is the oldest type of sliding block puzzle and it was invented by Noyes Chapman in 1880s. It is played on a 4-by-4 grid with 15 square blocks labeled 1 through 15 and a blank square. The goal is to rearrange the blocks so that they are in order, using as few moves as possible. Blocks are permitted to be slid horizontally or vertically into the blank square. Below you see two examples of 15-puzzle.

1	2	3	4
5	6	7	8
9	10	15	11
13	14		12

Initial state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal state

For the initial state we have illustrated, there are only three possible immediate successor states:

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

Move 15

1	2	3	4
5	6	7	8
9	10	15	11
13		14	12

Move 14

1	2	3	4
5	6	7	8
9	10	15	11
13	14	12	

Move 12

We will represent the moves that takes us to each of the board states above from the initial state as (15), (14), and (12) respectively. So, denoting a solution of the puzzle is to provide a sequence of such moves. Leftmost of the list is going to be the first move, i.e., the move that is applied to the initial state.

3 Solution using Breadth First Search

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 the class we have seen two basic algorithms that can serve that purpose: Depth First Search (DFS) and Breadth First Search (BFS). **In this assignment, you will implement BFS**. Below are some suggestions that might be helpful for you to get started:

- Before worrying about anything else, first take pencil and paper to draft what is needed to be done to tackle the problem. DO NOT start by copy pasting code from the slides. First understand what the problem is, how it will be solved, and what data structures can be used to achieve this goal!
- You need to determine how you are going to implement the graph data structure. Think about pros and cons of the two implementations that we covered in class: 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 order to implement BFS, think about which data structure you will need to use.
- Think about how you will store the correct path of steps that will take you from the start state to the goal state. Would you like to keep that information at each node? Or keep it at a separate array like structure?
- If you are still unsure where to start from, reach out to the instructor and/or TAs sooner rather than later!!!

4 Starter Package

- Along with this assignment instructions document, you will be provided with:
  - a **starter code** that has a sample function to read input from file and another sample function to write into file. You might need to change these functions to match the input/output specifications of this assignment. These functions are given to you to merely serve as a helper.
  - a few sample **input puzzles** along with their **solutions** where input is a shuffled k-puzzle and the output is the sequence of tiles to be replaced with the empty tile to reach to the goal state.
  - a simple **makefile**, that compiles the code to generate an executable
  - an **executable** *solve* **program**, which you can compare against your program to match input/output specifications. This program also generates puzzles and provides solutions for puzzles.
  - a starter **report file in LaTeX**, which you will fill with the analysis of your assignment.

5 Input/Output Specifications

- Your makefile needs to produce an executable named "solve", once we type "make" in command line.
- Your **executable** “solve” is going to take **two command line arguments**, first being the name of the input file and the second being the name of the output file. Thus, an example call to your program will be as follows:

  ./solve input.txt output.txt

- **Input file** is going to have the structure explained below. You do not need to make any error check for the correctness of the input file. You can assume that input specifications are always going to be satisfied by the test inputs.

#k
4
#initial state
1 2 3 4 5 6 7 8 9 10 15 11 13 14 0 12

- first and third line will be there for comment. You will ignore them while reading the file.
- second line will denote the **number of rows (or columns) of board**. i.e.,if $k = 4$, that means you will read 16 labels from the input file. **Your program can be tested for values of k where $k \leq 10$.** We're not going to test your program for crazy cases. Time is valuable for all of us :)
- fourth line consists of the **labels of the puzzle in reading order from left to right, top to bottom**. For instance, input example given above is the reading order of the board that was given as the example in previous page.
- empty block is going to be denoted by 0. The rest of the labels are going to be the integers from 1 to $k^2 - 1$.

• **Execution of the program:** Your program is going to calculate the moves necessary to make in order to arrive to the **goal state**. As stated earlier in the previous page, goal state is the state where labels are in the following order over the example board of 4×4:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

That is, all the labels are in sorted order from left to right, top to bottom, and the empty label (denoted by 0) is at the bottom right corner of the board.

A solution is going to be considered **invalid** if either one of the following occurs:

- **Invalid move:** In a board, you can only move the tiles that are around the empty block, and you can only move them orthogonally. Thus, diagonal moves, or moves that take a tile which is not in immediate vicinity of empty block is considered invalid.
- **Repeated state:** If the board state that is to be produced after a move is identical to a state which was previously created by your program, this will be considered as an invalid move.. (Our evaluation program will keep track of intermediate states, and will check if you are revisiting an already visited board state)

• **Output file** needs to strictly satisfy the following structure:

#moves
15 11 12

- first line is comment to be ignored.
- second line consists of **labels of the blocks to be moved** at a time, first move being the leftmost, and each move separated by a white space. For instance, the example output given above is a valid output to solve the initial state given in first page of this document. It stands for moving the blocks 15, 11, and finally 12 to arrive to the goal state.
- If the test case that is supplied does not have a solution, you should write "no solution" to the output file in the second line. It is important to note that, half of the possible inputs in a $k^2 - 1$ puzzle does not have a solution. You need to find put a way to determine whether a solution exists or not (google will definitely help on that).

• **Report file** should provide the following information:

- Explain the data structure that you used. How did you implement it?
- What is the space (memory) complexity of your algorithm? Could it be better? If so, what would be the trade off?
- Explain the algorithm that you used. What is its running time? Could there be a better algorithm? Does it give the optimal solution? If it does not, would there be an algorithm that gives optimal solution?
- How did you calculate whether a valid solution existed for the problem or not?

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**, a **makefile**, and a **report file written in pdf** that explains your approach for solving the problem. 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.
- You will provide your **self assessment** for your effort for the **30 point** portion of the assignment.
- **50 point** portion of the remaining part will be for the **test cases** of the assignment.
- The final **20 point** will be for the **report file**.
- 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.