Department of Computer Science

CS260 Data Structures

Programming Assignment V

1 Regulations

Due Date : *Check Blackboard*

Late Submission Policy : *The assignment can be submitted at most 2 days past the due date. Each late submission will be subjected to a 10 point per day penalty.*

Submission Method : *The assignment will be submitted via Blackboard Learn/Gradescope*

Collaboration Policy : *The assignment must be completed individually.*

Cheating Policy : *Do not use code from sources except lecture slides. "Borrowing" code from sources such as friends, web sites, AI tools for any reason, including "to understand better" is condiered cheating. All parties involved in cheating get a 0 for the 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!!!
- You are strongly encouraged to use online visualizers for debugging your code such as this or this website, where starting from a board, you can see how the algorithm should proceed step by step.

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.
  - 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.

5 Input/Output Specifications

- Your **executable** (named "solve" for representation in this document) 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:** The 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).

- **Testing your own program:** Before submitting your code, compare the outputs of your program for various test data with that of the provided executable. If there exists differences in the outputs, that would lead to your program failing in test cases while grading. Make sure you match the input/output specifications. If your program is passing certain test cases over tux yet failing on autograder, it is most probably either due to cases that you didn't test yourself, or your program has a memory leak which causes segmentation fault on autograder, which might be compensated by the compiler on tux. To ensure that you do not leak memory, use valgrind.

- **Test cases and grading:** Your program will be tested across several test cases, and grading for each test case will be as follows:

  1. You get full credit from a test case if your output is both **valid** and is **produced within at most 2× time** to produce the output compared to the provided executable.
  2. You get half the credit from a test case if your output is valid but it took longer than the above mentioned time.
  3. You get no credit from a test case if your output is invalid and/or your program takes too long to run (i.e., timeout for the test case occurs, which is way longer than 2 times the runtime of provided executable).

6 Submission

- Even if you develop it elsewhere, your code must run on **tux**. So, make sure that it compiles and runs on tux prior to submitting it.

- Your submission will consist of two separate files (do not compress/zip the files).

  1. A single C file named **main.c**, which includes your code for the assignment.
  2. A single file named **self_evaluation**, without any file extension, which contains answers to the following questions. Your answers can be as long or as short as you would like.
     (a) How many hours did you spend on this assignment?
     (b) What did you struggle with the most?
     (c) What did you learn from this assignment?

- Submit your assignment by following the Gradescope link for the assignment through Blackboard Learn.

7 Grading

- Assignment will be graded out of 100 points.

- You will earn 10 points for submitting a file with your self-evaluation.

- The remaining 90 points will be awarded according to how many test cases your program is able to pass.