CPSC3620 Programming Project

February 14, 2020

Deadline for submission: By April 01, 2020, 23:55

1 Sliding Tiles

In this programming project you are to write a program that solves a simpler version of a popular board game called *sliding tiles*. In this game, a square board is filled with numbered tiles with one position left empty. Your program will solve a 3×3 problem which has 8 numbered tiles indicated by labels $1, 2, \ldots, 8$ and a blank space indicated by label 0. The location of the

1	2	3
7	4	5
0	8	6

Figure 1: A 3×3 sliding tile board

tiles in a 3×3 board is called a **configuration**. For example, the configuration of the board in Figure 1 can be represented by the text string `"1 2 3 7 4 5 0 8 6"` (also known as row-oriented layout). Tiles adjacent to the blank space (label 0) can **move** left (L), right(R), up(U), or down(D). Thus, from a given configuration, a tile adjacent to the blank space can slide into the blank space to give a *new* configuration. The sliding puzzle game problem can be stated in the following way.

"Given an *initial* configuration and a *goal* configuration, find a sequence of **moves** that gives the *goal* configuration, starting from the *initial* configuration, using the least number of moves".

Suppose we are given the initial configuration as in Figure 1 and let the goal configuration be

1	2	3
4	5	6
7	8	0

.

There are two possible moves from the configuration of Figure 1 to obtain a new configuration:

1. the blank space can be moved one position to the right (indicated by R in Figure 2) or
2. the blank space can be moved one position up (indicated by U in Figure 2).

$$\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 7 & 4 & 5 \\ \hline \mathbf{0} & 8 & 6 \\ \hline \end{array} \overset{R}{\Longrightarrow} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 7 & 4 & 5 \\ \hline 8 & \mathbf{0} & 6 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 7 & 4 & 5 \\ \hline \mathbf{0} & 8 & 6 \\ \hline \end{array} \overset{U}{\Longrightarrow} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \mathbf{0} & 4 & 5 \\ \hline 7 & 8 & 6 \\ \hline \end{array}$$

Figure 2: Configurations obtained from tile board of Figure 1

A natural question at this point is: which direction should we move the blank space? Clearly, we can try all permissible directions. In this way, in the worst case, we may have to explore all possible board configurations – a very large number. Indeed, for an $n \times n$ board where n is a parameter to the problem, the sliding tile problem is computationally intractable or NP-hard. Fortunately, the 3×3 problem can be solved in a reasonable amount of time and space. We use an algorithm called A^* **search** to guide us in choosing a direction to reach the next configuration. In A^* **search** we **minimize** a function which is the sum of the number of moves taken to reach the current configuration (from the initial configuration) and an estimate of the number of moves necessary to reach the goal configuration from the current configuration. Of course, if the minimum number of steps to the goal configuration is already known the problem is solved! Since we do not have that information, we make a guess regarding the number of steps to reach the goal configuration. We use a measure called **Manhattan Distance** or **Taxi Cab Distance**. The Manhattan distance heuristic (the guess!) is used for its simplicity and also because it is actually a good lower bound on the number of moves required to bring a given configuration to the goal configuration. The Manhattan distance is the sum of the distances of each tile in a given configuration from its location in the goal configuration, completely ignoring all the other tiles. To illustrate the computation of Manhattan distance consider the configuration of Figure 1 and the goal configuration shown earlier. Tiles 1, 2, 3, and 8 are already in the required final location of the goal configuration. Tiles $4, 5, 6, 7$ are just one move away from their goal. Blank position is not included in the calculation. Thus, the Manhattan distance of the configuration displayed in Figure 1 from the goal configuration is $0 + 0 + 0 + 1 + 1 + 1 + 1 + 0 = 4$. In Figure 2 the configuration reached using the R move has Manhattan distance $0 + 0 + 0 + 1 + 1 + 1 + 1 + 1 = 5$ and the one reached using a U move has the Manhattan distance $0 + 0 + 0 + 1 + 1 + 1 = 3$. The following function on number of moves is minimized to select the next move,

$$D(C) = A(C) + E(C)$$

where C is a board configuration, $D(C)$ is the distance function that we minimize, $A(C)$ is the number of moves needed to reach configuration C from the start configuration, and $E(C)$ is an estimate of the number of moves needed to reach the goal from configuration C. In our implementation we use Manhattan distance measure for $E(C)$. For the example in Figure 1, the sequence (of moves) shown in Figure 3 gives the goal configuration which is also the minimum number of moves needed. You should verify this.

The A^* search algorithm is efficient in that it avoids exploring all possible next configura-

$$\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 7 & 4 & 5 \\ \hline 0 & 8 & 6 \\ \hline\end{array} \xrightarrow{U} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 0 & 4 & 5 \\ \hline 7 & 8 & 6 \\ \hline\end{array} \xrightarrow{R} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 0 & 5 \\ \hline 7 & 8 & 6 \\ \hline\end{array} \xrightarrow{R} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 0 \\ \hline 7 & 8 & 6 \\ \hline\end{array} \xrightarrow{D} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 0 \\ \hline\end{array}$$

Figure 3: A solution to the example 3×3 sliding tile puzzle

tions. A priority queue is an appropriate ADT to find the minimum value of D. You can use `minHeap` data structure to implement a priority queue.

To solve the problem you can maintain a priority queue of configurations based on the value of function D as key. Each time a new configuration is reached it should be inserted with its D value into the priority queue. The algorithm chooses a configuration with the minimum D value to explore next. Furthermore, your implementation must avoid inserting duplicate configurations in the priority queue. You also need to find a way to test whether the configuration that you are about to insert in the queue already exists in the queue, to avoid duplication. This way you save both running time and space.

Use a `minHeap` data structure available in the Standard Template Library (STL) or use the one given in the textbook.

2 What to submit

1. Define and implement a class named `Board_Tile` to represent 3×3 tile boards. The class should have a data member `config` to represent a 3×3 tile board configuration as a `C++` string, a second data member `movesFromStart` of type `C++` string representing the moves or steps that led to this configuration from a given start configuration. You need to define and implement the following member functions in the class.

   (a) A constructor `Board_Tile(const string&)` that takes a string parameter representing a tile board configuration which is the `initial configuration` and creates an object of the class type.

   (b) A member function `nextConfigs()` that returns a list of `Board_Tile` at most 4 objects that are one move away from the current object.

   (c) A member function `int numMoves()` that returns the number of moves it took from the initial board to reach the current configuration.

   (d) A member function `int Manhattan_Distance(const Board_Tile& goalconfig)` that takes a `Board_Tile` object `goalconfig` representing the goal configuration and returns the value of the Manhattan distance of the object.

   Feel free to add additional data members and member functions if needed for your implementation.

2. Define and implement a class called `Sliding_Solver` as specified below that solves a 3×3 sliding tile puzzle using A^* search.

   (a) It contains a data member called `tileQueue` representing a minHeap of `Board_Tile` objects.

(b) Define and implement a constructor to create an object of type `Sliding_Solver` given a string representing a start configuration.

(c) Define and implement a member function called `Solve_Puzzle` that solves the puzzle using A^* search.

Feel free to add additional data members and member functions if needed for your implementation.

3. Write a test program which contains the `main` function. Your test program should allow users to specify a start and goal configuration and output the solution with relevant information. Given below is a sample output.

```
Start Board         Goal Board         Number of Moves      Solution

123745086           123456780             4                  URRD

436871052           123456780            18                  URRULDDRULDLUURDRD
```

If there are multiple solutions, you need to output only one of them.

4. A pdf document `Read_me` containing instructions for compiling and running your program.

3 Grading Method

First and foremost, the code submitted by you must compile and run on the Linux lab machines of the department. You must also have an error-free make file which compiles your code using the `make` command from the command line.

Create a directory and put all your code (.h and .cc files), a `make` file, and a `README` file in it. Create a tar archive and name it with your login id for CPSC3620 computer account. Submit the tar file using the link provided on `Moodle` page for CPSC3630. Points will be deducted for poor design decisions and uncommented or unreadable code as determined by the grader.

Here is the points breakdown:

1. The program compiles without compile error and runs without crashing: 20 points. **If your program does not compile or the program crashes, the grader reserves the right to not award any credit under the following categories.**

2. The design and implementation of the classes follow the specification given (make sure that the name of each class and its members are as indicated): 15 points.

3. A^* search algorithm is correctly implemented: 40 points.

4. **Manhattan Distance** calculation is correct: 15 points

5. The program avoids inserting duplicate `Board_Tile` object as specified in Section 2: 10 points