UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science & Applied Mathematics

Introduction to Algorithms & Programming
Uninformed Search

1 Introduction

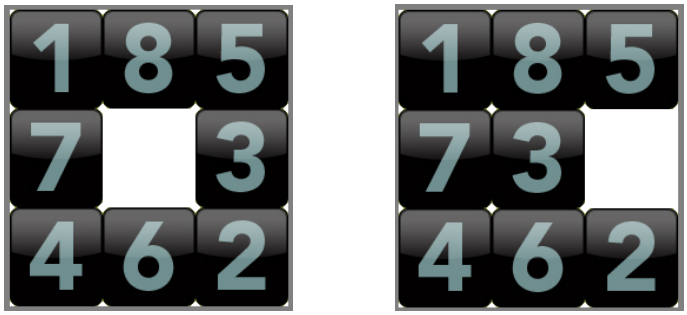
In this lab, we'll be implementing breadth-first and uniform-cost search on the 8-puzzle problem. Before starting, please make sure you have read Chapter 3 (Sections 3.1 – 3.5) in *Artificial Intelligence: A Modern Approach*.

2 8-Puzzle

The 8-Puzzle is a game in which a player is required to slide tiles around on a 3×3 grid in order to reach a goal configuration. If you have never played before, I recommend try it out here: `http://www.artbylogic.com/puzzles/numSlider/numberShuffle.htm?rows=3&cols=3&sqr=1`.

There is one empty space that an adjacent tile can be slid into. An alternative (and simpler) view is that we are simply moving the blank square around the grid. Note that not all actions can be used at all times. For example, the blank tile cannot be shifted up if it is alread on the top row!

The transition below illustrates the blank tile being moved to the `right`.



(a) (b)

Figure 1: Starting in the configuration (a), the agent executes the action `right`. This moves the blank tile to the right, exchanging its position with 3, as seen in (b).

Submission 1: Setting the board (10 marks)

The first step is to create our 8-puzzle game. Write a C++ program that reads in a single string, stores that information in an appropriate data structure, and then outputs it as a 3×3 square.

Input

Input consists of a single string of digits and the hash symbol. The position of the digits in the string is its location on the board, starting from top-left and moving to bottom right. The hash represents the blank space. For example, the board configuration given in Figure 1(a) would be represented by the string `1857#3462`.

Output

Print out the board as a 3×3 matrix, where each value is separated by a space.

Example Input-Output Pairs

Sample Input #1

```
1857#3462
```

Sample Output #1

```
1 8 5
7 # 3
4 6 2
```

Sample Input #2

```
18573#462
```

Sample Output #2

```
1 8 5
7 3 #
4 6 2
```

Sample Input #3

```
78651#432
```

Sample Output #3

```
7 8 6
5 1 #
4 3 2
```

Submission 2: Implementing moves (10 marks)

The next step is to implement action dynamics. Write a C++ program that takes in an initial configuration and an action, and outputs the resulting configuration and the associated cost. For all actions, the cost is 1.

Input

Input consists of two lines. The first line is a single string of digits and the hash symbol. The position of the digits in the string is its location on the board, starting from top-left and moving to bottom right. The hash represents the blank space. For example, the board configuration given in Figure 1(a) would be represented by the string `1857#3462`. The second line is the action taken, which is one of either `UP`, `DOWN`, `LEFT` or `RIGHT`.

Output

The first output line is the cost of the action. The next few lines are the resulting configuration as a 3×3 matrix, where each value is separated by a space.

Example Input-Output Pairs

Sample Input #1

```
1857#3462
LEFT
```

Sample Output #1

```
1
1 8 5
# 7 3
4 6 2
```

Sample Input #2

```
18573#462
UP
```

Sample Output #2

```
1
1 8 #
7 3 5
4 6 2
```

Sample Input #3

```
78651#432
DOWN
```

Sample Output #3

```
1
7 8 6
5 1 2
4 3 #
```

Submission 3: Implementing available actions (10 marks)

The next step is to determine which actions are available in which states. Write a C++ program that takes in an initial configuration, and outputs which actions are available at teh given state.

Input

Input consists of a single string of digits and the hash symbol. The position of the digits in the string is its location on the board, starting from top-left and moving to bottom right. The hash represents the blank space. For example, the board configuration given in Figure 1(a) would be represented by the string `1857#3462`.

Output

Output the list of available actions, each on their own line. The actions should be output in the order `UP`, `DOWN`, `LEFT`, `RIGHT`.

Example Input-Output Pairs

Sample Input #1	Sample Input #2	Sample Input #3
`1857#3462`	`18573#462`	`78651432#`
Sample Output #1	**Sample Output #2**	**Sample Output #3**

`UP`
`DOWN`
`LEFT`
`RIGHT`
	`UP`
	`DOWN`
	`LEFT`
		`UP`
		`LEFT`

Submission 4: Breadth-first search (50 marks)

Now that we've implemented the problem, we can now implement our search strategies. First, write a C++ program that accepts an initial and goal state, and returns the cost of the shortest path between the two, given that every action has a cost of 1.

Input

Input consists of two lines. The first line is a representation of the initial state, and the second line is the goal state.

Output

Output the cost of the optimal plan from the start to the goal state.

Example Input-Output Pairs

Sample Input #1	Sample Input #2	Sample Input #3
`78651#432` `12345678#`	`1857#3462` `185#73462`	`1857#3462` `78651432#`
Sample Output #1	**Sample Output #2**	**Sample Output #3**
`25`	`1`	`20`

Submission 5: Uniform-cost search (20 marks)

Now let's modify our code to implement UCS! First, modify the problem so that all actions have cost 1, except for `UP`, which now has a cost of 5. Then, write a C++ program that accepts an initial and goal state, and returns the cost of the shortest path between the two according to UCS.

Input

Input consists of two lines. The first line is a representation of the initial state, and the second line is the goal state.

Output

Output the cost of the optimal plan from the start to the goal state.

Example Input-Output Pairs

Sample Input #1	Sample Input #2	Sample Input #3
78651#432 12345678#	1857#3462 185#73462	1857#3462 78651432#
Sample Output #1	**Sample Output #2**	**Sample Output #3**
49	1	36

Extra (not for marks): Iterative deepening search

Implement depth-first search and iterative deepening. Compare the solutions found by these algorithms to those found by UCS and BFS in the previous examples!