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

Artificial Intelligence (COMS4033A/7044A)
Search

1 Introduction

In this lab, we'll be implementing some of the ideas on search we looked at in the lecture. Like all good AI researchers, we'll be applying this to a game called 15-Puzzle, which will require us to first create the game before applying our various search algorithms.

2 15-Puzzle

The 15-Puzzle is a game in which a player is required to slide tiles around on a 4×4 grid in order to reach a goal configuration. If you have never played before, I recommend try it out here: `https://15puzzle.netlify.app/`.

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 already on the top row!

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

A	B	E	H
C	D		F
G	I	L	M
J	K	O	N

(a)

A	B	E	H
C	D	F	
G	I	L	M
J	K	O	N

(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 F, as seen in (b).

The cost of each move is 1 and the goal of the game is to slide the tiles around to reach some goal configuration, which we will take to be the letters in alphabetical order, with the bottom right corner housing the blank slot as seen below:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	

Figure 2: The goal configuration

Submission 1: Setting the board (3 marks)

The first step is to create our 15-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 4×4 square.

Input

Input consists of a single string of characters and the hash symbol. The position of the characters 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 `ABEHCD#FGILMJKON`.

Output

Print out the board as a 4×4 matrix, where each value is separated by a space.

Example Input-Output Pairs

Sample Input #1

```
LAIB#KGCDOHENMJF
```

Sample Output #1

```
L A I B
# K G C
D O H E
N M J F
```

Sample Input #2

```
ELFIGHJONAKDMB#C
```

Sample Output #2

```
E L F I
G H J O
N A K D
M B # C
```

Sample Input #3

```
BH#NMJCELOIKAFGD
```

Sample Output #3

```
B H # N
M J C E
L O I K
A F G D
```

Submission 2: Implementing moves (4 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.

Input

Input consists of two lines. The first line is a single string of characters and the hash symbol. The position of a character in the string is its location on the board, starting from top-left and moving to bottom right. The hash represents the blank space. The second line is the action taken, which is one of either `UP`, `DOWN`, `LEFT` or `RIGHT`.

Output

Output the resulting configuration as a 4×4 matrix, where each value is separated by a space.

Example Input-Output Pairs

Sample Input #1

```
LAIB#KGCDOHENMJF
RIGHT
```

Sample Output #1

```
L A I B
K # G C
D O H E
N M J F
```

Sample Input #2

```
ELFIGHJONAKDMB#C
UP
```

Sample Output #2

```
E L F I
G H J O
N A # D
M B K C
```

Sample Input #3

```
BH#NMJCELOIKAFGD
LEFT
```

Sample Output #3

```
B # H N
M J C E
L O I K
A F G D
```

Submission 3: Implementing available actions (3 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 the given state.

Input

Input consists of a single string of characters and the hash symbol. The position of a character in the string is its location on the board, starting from top-left and moving to bottom right. The hash represents the blank space.

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
`LAIB#KGCDOHENMJF`	`ELFIGHJONAKDMB#C`	`BH#NMJCELOIKAFGD`
Sample Output #1	**Sample Output #2**	**Sample Output #3**

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

Submission 4: Breadth-first search (30 marks)

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

Input

Input consists of a single line representing the initial state.

Output

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

Example Input-Output Pairs

Sample Input

```
#AGCEBFDIJKHMNOL
```

Sample Output

```
8
```

Submission 5: A* with Misplaced Tiles (30 marks)

BFS works when the problem has a relatively small solution length, but takes a very long time otherwise. To solve much harder problems, we'll need to implement A* search. In this submission, we'll be using the heuristic labelled h_1 in the textbook. That is, the estimate of the cost-to-go is the number of tiles in the current configuration that are in the wrong location according to the goal.
Write a C++ program that accepts an initial state, and returns the cost of the shortest path between the start state and the goal state (`ABCDEFGHIJKLMNO#`) using A* with the above heuristic

Input

Input consists of a single line representing the initial state.

Output

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

Example Input-Output Pairs

Sample Input

```
EC#DBAJHIGFLMNKO
```

Sample Output

```
14
```

Submission 6: A* with Tile Distances (10 marks)

The previous submission used an admissible heuristic, but can we do better?
In this submission, we'll be using the heuristic labelled h_2 in the textbook. That is, the estimate of the cost-to-go is the sum of the distances of the tiles from their goal positions.
Write a C++ program that accepts an initial state, and returns the cost of the shortest path between the start state and the goal state (`ABCDEFGHIJKLMNO#`) using A* with the above heuristic

Input

Input consists of a single line representing the initial state.

Output

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

Example Input-Output Pairs

Sample Input

```
EABCM#GDKFILNOJH
```

Sample Output

```
30
```

Submission 7: All Together (20 marks)

We have now created three search algorithms. Let's see how efficient they are by counting how many nodes are expanded in the search!

To run this experiment, download the list of puzzle configurations on Moodle. For each of the puzzles, run BFS, A* with heuristic h_1 and A* with heuristic h_2. For each algorithm, count how many nodes are added to the frontier (i.e. each time you add a node to the frontier, increment a counter). Then, for each puzzle, record the length of the answer and the number of nodes expanded by each algorithm. An example of such results may look something like this:

Puzzle	**Solution**	**BFS**	**A*-h1**	**A*-h2**
ABCDEFG#IJKHMNOL	2	4	5	5
ABCDEFGHIJ#LMNKO	2	8	6	6
AB#CEFGDIJKHMNOL	4	50	8	8

Given this information produce a graph that plots, for each algorithm, the number of nodes expanded as a function of solution length. The y-axis should be plotted on a log scale. If there are entries in the table that have the same solution length (e.g. in the above graph, there are two entries with solution length 2), then average those to produce the average node expansion for each algorithm (e.g. in the above, the number of nodes BFS expands for solution length 2 would be 6).

Note that for the last few puzzles, it is likely that BFS may be too slow to produce an answer on your PC. If this is the case, then use a placeholder value of 1000000.

You may use any plotting library or tool (e.g. Excel) to generate the graph.

To Upload

Submit two files on Moodle: the first file should be a text file that contains the results for each algorithm on each of the provided puzzles (in a format similar to the above table). The second file should be an image of your graph as described above.