**CS2040S: Data Structures and Algorithms**

# Tutorial Problems for Week 12: Shortest Paths (I)

*For: 1 Nov 2021, Tutorial 10*

**Solution: Secret! Shhhh... This is the solutions sheet.**

## Problem 1. Unlock the lock

(UVa 12160 – `https://uva.onlinejudge.org/external/121/p12160.pdf`)

Mr. Ferdaus has created a special type of 4-digit lock named "FeruLock". It always shows a 4-digit value and has a specific unlock code $U$ (An integer value between 0000 and 9999). The lock is unlocked only when the unlock code is displayed. This unlock code can be made to appear quickly with the help of some of the special buttons available with that lock. There are $R$ ($1 \leq R \leq 10$) such special buttons and each button has a number (between 0000 and 9999) associated with it. When any of these buttons is pressed, the number associated with that button is added with the displayed value and so a new number is displayed. The display starts with some value $L$ ($0000 \leq L \leq 9999$). The lock always uses least significant 4 digits after addition. After creating such a lock, he has found that, it is also very difficult for him to unlock the Ferulock. As a very good friend of Ferdaus, your task is to create a program that will help him to unlock the Ferulock by pressing these buttons minimum number of times. If there is no way to unlock the FeruLock simply output "Permanently locked".

$L, U$ and the values of each special button will always be denoted by a 4 digit number (even if it is by padding with leading zeroes). Some examples are as follows:

- If $U = 9999, L = 0000, R = 1$ and the value of the button is 1000, then there is no way to unlock the FeruLock, so output "Permanently locked".
- If $U = 9999, L = 0000, R = 1$ and the value of the button is 0001, then the minimum number of button presses to unlock the FeruLock will be 9999.
- If $U = 1212, L = 5234, R = 3$ and the values of the buttons are 1023, 0101 and 0001 respectively, then the minimum number of button presses will be 48.

**Solution:** This problem can be modelled as a SSSP on an unweighted, directed graph. The vertices are numbers between [0000..9999]. An unweighted directed edge $A \rightarrow B$ exists between two numbers $A$ and $B$ if $B = (A + \text{certain button value})\%10000$. This is an example of implicit edges - they are not supplied by the input or represented as objects in memory, but rather determined algorithmically from the input.

We can use use BFS to compute the shortest path from the vertex with value $L$ as the source, to the vertex with value $U$. If $U$ is not reachable from $L$ even after completion of BFS (i.e. `dist[u] = infinity`), then output "Permanently locked'.

**Problem 2. Escape Plan** (2012/2013 CS2010 WQ2)

You work in a maze. Unfortunately, portions of the maze have caught on fire, and the owner of the maze neglected to create a fire escape plan. You need to escape QUICKLY. Given your location in the maze and which squares (can be zero, one, or more than one) of the maze that are initially on fire (when you first realized that there is a fire), you must determine **whether** you can exit the maze before the fire reaches you; and if you can, **how fast** you can do it.

Both you and any of the fire each move one square per minute, vertically or horizontally (not diagonally). The fire spreads in all four directions from each square that is on fire. You may exit the maze from *any square that borders the edge of the maze*. Neither you nor the fire may enter a square that is occupied by a wall.

You are given two integers $R$ and $C$ ($1 \leq R, C \leq 1000$) and a 2D character array $M$ with $R$ rows and $C$ columns that describes the starting condition of the maze. Each cell can contain either:

- '`#`', a wall. Neither you nor the fire may enter this cell.
- '`.`', a passable square (for you and the fire).
- '`Y`', your initial position in the maze, which is a passable square. There will be **exactly one** '`Y`' cell only in the test case.
- '`F`', a square that is currently on fire at this point of time. There can be **zero, one, or more than one** '`F`' cell(s) in the test case.

Your task is to implement the following function `EscapePlan` that takes in $R, C, M$ and simply returns -1 if you cannot exit the maze before the fire reaches you, or a positive integer that gives the earliest time that you can safely exit the maze, in minutes.

Example 1: R1 = 4, C1 = 4, maze M1 is as shown below

```
####    ####    ####    ####
#YF#    #FF#    #FF#    #FF#
#..#    #YF#    #FF#    #FF#
#..#    #..#    #YF#    #FF#
                         Y
t=0     t=1     t=2     t=3 (you manage to  escape)
```

The function `EscapePlan(R1, C1, M1)` will return 3 because by running downwards three steps (in three minutes), you can escape the maze while the fire cannot catch you.

Example 2: R2 = 3, C2 = 6, maze M2 is as shown below

```
######    ######
#Y....    #YF..F
#.F..F    #FFFFF

t=0       t=1 (you are already trapped)
```

The function `EscapePlan(R2, C2, M2)` will return -1 because after 1 minute, the nearest fire already surrounds you. Note that if you and the fire reach the same cell at the same time, you die.

Example 3: R3 = 3, C3 = 3, maze M3 is as shown below

```
###    ###    ###
#Y.    #.Y    #..Y
###    ###    ###

t=0    t=1    t=2
```

The function `EscapePlan(R3, C3, M3)` will return 2.

Example 4: R4 = 5, C4 = 5, maze M4 is as shown below

```
F...F     FF.FF     FFFFF     FFFFF
.....     F...F     FF.FF     FFFFF
..Y..     ....F     F..FF     FFFFF
....F     ..YFF     ..FFF     FFFFF
.....     ....F     ..YFF     ..FFF
                                 Y
 t=0       t=1       t=2       t=3 (a
                               narrow
                               escape)
```

The function `EscapePlan(R4, C4, M4)` will return 3, a narrow escape as shown above.

**Problem 2.a.** What do the vertices and the edges of the graph represent?

**Solution:** Vertices = '`.`' cells in the grid (we can replace the one cell of '`Y`' and a few cells of '`F`' to '`.`' too after recording their positions). There are up to $R \times C$ of them.

Edges = If there is another '`.`' cell in the N/S/E/W direction of the current '`.`' cell, then there is an edge between these 2 cells. These edges are unweighted. There are up to 4 edges per cells.

**Problem 2.b.** What is the graph problem that we want to solve?

**Solution:** This is a (Multi-Sources) Shortest Paths on unweighted graph. There are actually more than one source in this graph, the position of '`Y`' and the positions of all those '`F`'s. This is the key to the solution. We are not just restricted to Single-Source default variant at all times.

**Problem 2.c.** Give the most appropriate graph algorithm (or modified version of it) to solve this problem.

Use a BFS which will simulate the spreading of the fire and also you. The idea is that you are safe if you can reach the border before any fire, or you die otherwise (no more objects in the queue that represents you). Both you and fire spread at a rate of 1 cell at a time, which is what BFS computes.

We can run BFS by enqueuing all these sources ('`Y`' followed by the '`F`'s) first. We create an object that is to be enqueued with the following attributes:

1. Flag to indicate whether it is a '`Y`' or '`F`' object
2. Coordinate of the cell the source is in
3. Time stamp - set to 0 at the beginning and will indicate the time for an object (only necessary for '`Y`' objects) to reach a particular cell.

We keep a global count `numY` of the number of '`Y`'s in the queue. The dequeue operation (inside the loop step) of the BFS is modified as follows:

**Algorithm 1** Modified BFS dequeue

```
u = Q.dequeue()
M = cell given by the coordinate of u
if u is a 'Y' object then
    Decrement numY
    if M is marked as 'F' then                          ▷ Invalid 'Y' object, killed off by 'F'
        continue
    end if
    for each of the 4 cells u can move into do                                  ▷ 1 each for N,S,E,W
        if it is outside the grid boundary then
            return u.timestamp + 1
        end if
        if it is inside the grid boundary and no fire/wall in that cell (check against M) then
            Create a new 'Y' object v where v.timestamp = u.timestamp + 1
            Enqueue v and increment numY
            Mark M at the cell where v is with 'Y'
        end if
    end for
else if u is a 'F' object then
    for each of the 4 cells u can move into do                                  ▷ 1 each for N,S,E,W
        if it is outside the grid boundary then
            continue
        end if
        if it is inside the grid boundary and no fire/wall in that cell (check against M) then
            Create a new 'F' object v and enqueue into Q
            Mark M at the cell where v is with 'F'
        end if
    end for
end if
if numY == 0 then
    return 1                    ▷ no more 'Y' objects in the queue/grid, you cannot escape!
end if
```

After running BFS algorithm from multiple sources, all '`.`' cells in the map are eventually replaced by a '`Y`' or an '`F`'. This means each of the $R \times C$ cells has been processed at most once in the simulation. Since for each cell we at most consider all 4 directions, the total processing of cells is $4 \times R \times C$. Overall, this algorithms is $O(R \times C)$.

**Problem 3. Money Changer** (2011/2012 CS2010 WQ2)

Given $n$ currencies and $m$ exchange rates, determine if we can start with a certain amount of money in one currency, exchange this amount for other currencies, and end up at the same currency but with more money than what we had at first. Two examples are as follows:

- Suppose the money changer has n = 3 currencies and m = 3 exchange rates: 1 USD gives us

0.8 Euro; 1 Euro gives us 0.8 GBP (British pound sterling); and 1 GBP gives us 1.7 USD. So if we start with 1 USD, we can exchange it for 0.8 Euro, which can then be exchanged for 0.64 GBP, and if converted back to USD we have 1.088 dollars. We have just make a profit of 8.8 cents.

- Example 2: If the money changer has the following n = 2 currencies and m = 2 exchange rates: 1 USD gives us 0.8 Euro; and 1 Euro gives us 1.25 USD, then there is no way we can make a profit.

Can you model the $n$ currencies and their $m$ exchange rates as a graph problem and give an algorithm to report whether it is possible to start with any currency, exchange it with one (or more) other currencies, end with the same starting currency, and make a profit? State the time complexity of your algorithm.

**Solution:** Vertices will represent each currency. There is a directed edge from vertex $u$ to vertex $v$ if you can exchange the currency represented by vertex $u$ to the currency represented by vertex $v$. The edge weight is the exchange rate. In this graph, we want to find at least one "profitable" cycle, that is if we multiply all edge weight in such a cycle, the result should be greater than 1. If you understand this part, you can modify the standard Bellman Ford's algorithm to check for such "profitable cycle" this way. But if you are not sure, read on.

For any given profitable cycle $u_1 \sim u_1$,

$$weight(u_1, u_2) \times weight(u_2, u_3) \times \ldots \times weight(u_i, u_1) > 1$$

if and only if

$$\log(weight(u_1, u_2)) + \log(weight(u_2, u_3)) + \ldots + \log(weight(u_i, u_1)) > 0$$

Negating the logarithms, we have

$$-\log(weight(u_1, u_2)) - \log(weight(u_2, u_3)) - \ldots - \log(weight(u_i, u_1)) < 0$$

By transforming the exchange rate to the negative of its logarithm and use it to represent the edge weights, a "profitable" cycle will be represented by a negative cycle. Using the Bellman-Ford algorithm, we can easily detect whether such a cycle exists by running 1 more iteration after the $(V-1)$-th iteration and checking if any vertex has its shortest path distance relaxed. If there is, there is a profitable cycle. The time complexity is $O(VE) = O(nm)$.

*Note that in real life, it is not really possible to gain profit using this technique due to the difference of buying and selling rates of each currency. Doing this will just make the money changer happier (and you simply lose your money).*

**Problem 4. Heights** (2018/2019 Take-Home Lab 3)

Cats like to sit in high places. It is not uncommon to see cats climbing trees or furniture in order to lie on the top-most area within their feline reach. Rar the Cat is no exception. However, he

does not know how high is one area relative to another.

Height can be measured in centimeters (cm) above sea level but Rar the Cat does not know the absolute height of any place. However, he knows that area $B_i$ will be higher than area $A_i$ by $H_i$ centimetres because he needs to jump $H_i$ to get from area $A_i$ to $B_i$. There will be $N$ areas in total with $N-1$ such descriptions. Areas are labelled from 1 to $N$ and $0 < A, B \leq N$ where $A \neq B$.

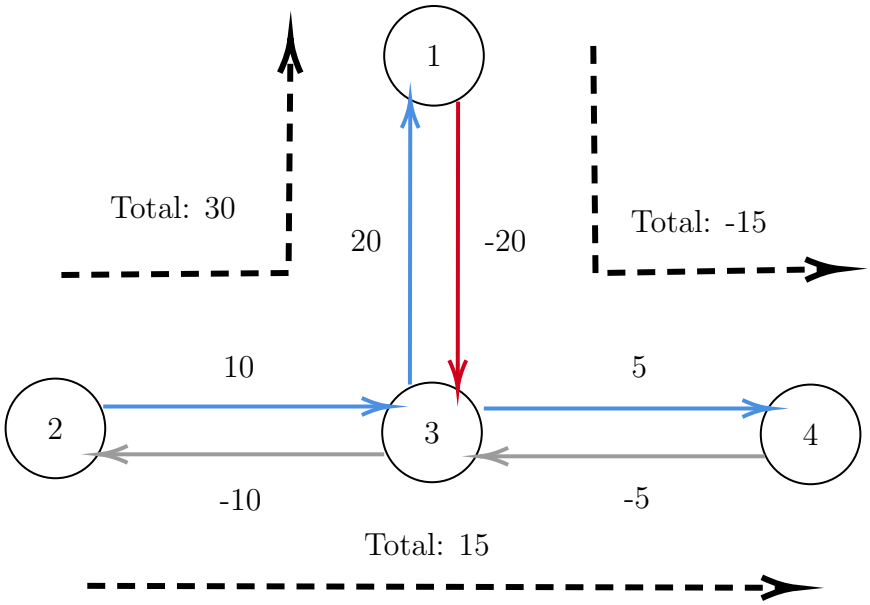
Rar the Cat also has $Q$ queries, each consisting 2 integers $X$ and $Y$. He wants to know the height of area $Y$ with respect to area $X$. Do note that $0 < X, Y \leq N$ but $X$ can be equal to $Y$. In the event that area $Y$ is lower than area $X$, please output a negative number. Otherwise, output a positive number.

It is guaranteed that the relative heights of all pairs of areas can be computed from the data provided in the input. **Design an algorithm that takes in the $N$ areas and $N-1$ descriptions, and outputs the correct relative height for all $Q$ queries efficiently**. Your time complexity should be dependent on both $N$ and $Q$.

**Solution:** Each area is represented by a vertex, and $(A_i, B_i, H_i)$ represents the directed edge from one area to the other, with the difference in height $H_i$. We can also create edges for the "reverse directions", which are also required to solve the problem. Note that since $H_i$ is always positive, we need to represent the reverse direction by $(B_i, A_i, -H_i)$.

A simple solution is to perform BFS/DFS from $X$ for each query $X, Y$ to find the distance of $Y$ from $X$, which takes $O(N)$ time since there are $N$ vertices and $2(N-1)$ edges. The time complexity is thus $O(N \cdot Q)$ time for $Q$ queries, which is inefficient. Instead, notice that we need not run DFS multiple times. There are two observations we can make here.

- Running BFS/DFS from $X$ gives us:
  1. The distance from $X$ to any vertex.
  2. The distance from any vertex to $X$ (by negation).
- The distance from $X$ to $Y$ is equal to the sum of the distance from $X$ to $Z$ and $Z$ to $Y$, where $Z$ can be any vertex in the graph, **even if the vertex $Z$ is not in the direct path from $X$ to $Y$**. An example is given in the diagram below.




For example, if we wanted to find the distance from vertex 2 to vertex 4, notice that it is the sum of the distances from vertex 2 to vertex 1, and vertex 1 to vertex 4. Notice that the 20 from $3 \to 1$ and the -20 from $1 \to 3$ will cancel out. Thus, the total distance from the path $2 \to 3 \to 1 \to 3 \to 4$ is the same as that if we took the path $2 \to 3 \to 4$ directly. The consequence of this is that as long as we know the distance from one vertex to all other vertices, we will know the distance between all pairs of vertices.

All that is required is to run BFS/DFS from any vertex, say vertex 1 for simplicity, to obtain the distance from any vertex 1 to all other vertices, stored in the distance array `dist`. Thus `dist[i]` will store the distance from vertex 1 to vertex $i$. Now, to answer the query for $X, Y$, we take `-dist[x] + dist[y]`, the sum of the distances from $X \to 1$ and $1 \to Y$. Note that we take `-dist[x]` instead of `dist[x]` because we want the distance from $X$ to 1 and not 1 to $X$. The time complexity of this algorithm is $O(N + Q)$ since we can process each query in $O(1)$ time.