CS2040

Tutorial 10: Shortest Paths (I)

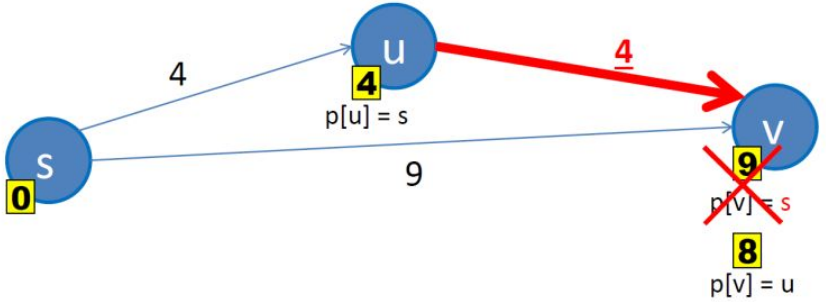
Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)


:D
SSSP

Relax Edges

Let e be an edge from u to v. Relax operation update distance to reach node v if distance to reach node u + weight of edge e is smaller than current distance to reach node v.

```
relax(u, v, w(u,v))
  if D[v] > D[u]+w(u,v) // if SP can be shortened
    D[v] ← D[u]+w(u,v) // relax this edge
    p[v] ← u // remember/update the predecessor
    // if necessary, update some data structure
```



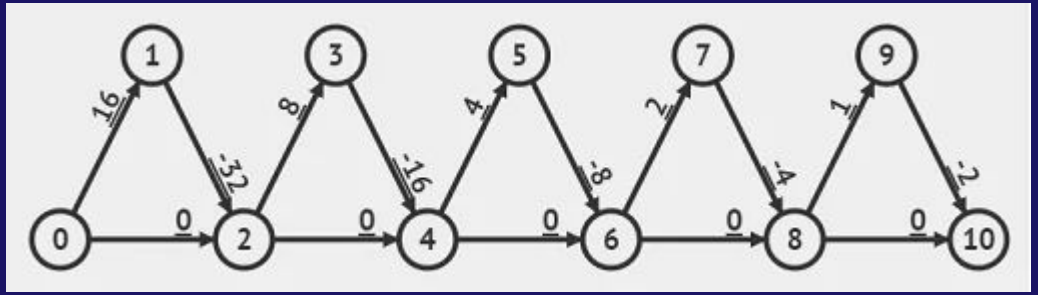
Shortest Path Algorithms

Single Source Shortest Path (SSSP)

1. BFS
2. Bellman-Ford
3. Dijkstra

All Pairs Shortest Path (APSP)

1. Floyd-Warshall



Bellman-Ford

Let V = the number of vertices, and E = the number of edges.

After we relax all E edges in arbitrary order, in the worst case, vertices that are one edge away from the starting point are guaranteed to have correct shortest distance value.

If we do the step above i times, vertices that are i edges away from the starting point are guaranteed to have correct shortest distance value.

Since the shortest path from any vertex to any vertex in a non-negative cycle graph must be a simple path (no cycle), doing the E edges relaxation V - 1 times guaranteed a shortest distance value for all vertex.

To check for negative cycle, relax E edges one more time (after V - 1 iterations), if there is still an update in the distance value then there is a negative cycle.

Complexity: O(VE)

Bellman-Ford

```
// Simple Bellman Ford's algorithm runs in O(VE)
for i = 1 to |V|-1 // O(V) here
  for each edge(u, v) ∈ E // O(E) here
    relax(u, v, w(u,v)) // O(1) here
```

Another variant

One-pass Bellman-Ford:

If the graph is DAG and the order of relaxing is the topological order, we only need to do one pass of Bellman Ford. So the complexity is O(E).

Bellman-Ford

```
// Simple Bellman Ford's algorithm runs in O(VE)
for i = 1 to |V|-1 // O(V) here
  for each edge(u, v) ∈ E // O(E) here
    relax(u, v, w(u,v)) // O(1) here
```

Another variant

One-pass Bellman-Ford:

If the graph is DAG and the order of relaxing is the topological order, we only need to do one pass of Bellman Ford. So the complexity is O(E).

OG Dijkstra

1. Declare a priority queue PQ that stores pair of distance and node (d, v) where d = D[v] (current shortest path estimate)
2. Enqueue (INF, i) for every vertex i in the graph except for source s. Enqueue (0, s) to PQ
3. Keep removing vertex u with minimum d from the PQ, add u to the Solved set and relax all its outgoing edges until the PQ is empty
4. If an edge (u,v) is relaxed find the vertex v it is pointing to in the PQ and “update” the shortest path estimate (decrease key)

Each vertex inserted/extracted into PQ once: O(V log V), each edges relaxed once: O(E), each relaxation decrease the key O(log V).
Total time complexity is O(VlogV + ElogV) = O((V + E) log V).

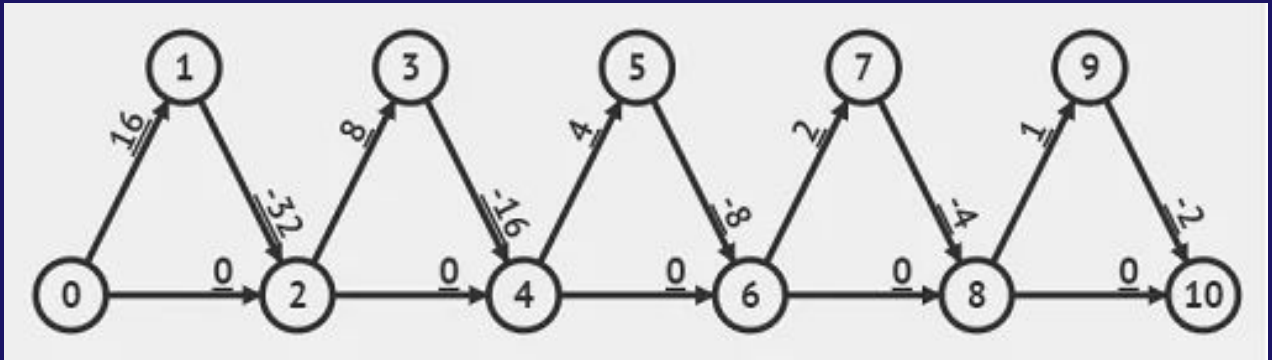
Modified Dijkstra (the cooler Dijkstra)

1. Declare a priority queue PQ that stores pair of distance and node (d, v) where d = D[v] (current shortest path estimate)
2. Only enqueue (0, s) to PQ
3. Keep removing vertex u with minimum d from the PQ, if d = D[u] we relax all edges out of u, else we discard (d,u)
4. If during edge relaxation, D[v] of a neighbor v of u decreases, enqueue a new (D[v], v)

Each vertex processed in PQ once (although multiple copies, d > D[v] check does not process outdated copies). Total number of insertions into PQ is O(E) and extractMin from PQ is O(log E). Total complexity is thus O(E log E).

Special Cases (choose wisely…)

1. Graph is a tree: O(V) BFS/DFS (V vertices, V - 1 edges). There is an unique path from any vertex to any vertex
2. Graph is unweighted: O(V + E) BFS
3. Graph is directed and acyclic: One-pass Bellman Ford, O(E)
4. Graph has no negative edge: Original Dijkstra, O((V + E) log V)
5. Graph has no negative weight cycle: Modified Dijkstra, O(E log E)



Floyd-Warshall

1. Use 2D matrix V x V
2. Initialise D[i][i] to 0 and D[i][j] to the weight of edge(i,j) if exists, INF otherwise
3. Run this $O(V^3)$ algorithm

```
for (int k = 0; k < V; k++) // remember, k first
  for (int i = 0; i < V; i++) // before i
    for (int j = 0; j < V; j++) // then j
      D[i][j] = Math.min(D[i][j], D[i][k]+D[k][j]);
```


If (D[i][k] + D[k][j] < D[i][j])
D[i][j] = D[i][k] + D[k][j]

Relaxation of D[i][j]

4. Once we run this $O(V^3)$ pre-process, future queries can be done in $O(1)$

Floyd-Warshall can be used to find transitive closure, minimax/maximin, and detecting +ve/-ve cycle. (See lecture 20 for more details)

01

UNLOCK THE LOCK

Unlock the lock

Abridged problem description:

- There is a 4-digit lock, possible values 0000 to 9999
- The starting number is L, the code is U
- There are R buttons that we can press, each button i is associated with a number R_i
- The resulting number after button i being pressed is $(L + R_i) \% 10000$ (take the last 4 digits)

What is the minimum number of button pressed to get number U?

Similar question at https://open.kattis.com/problems/buttonbashing

Thinking process

What do the nodes and edges in the graph represent?

Nodes: 0 to 9999
Edges: A → B if B = (A + R_i) % 10000 for some i

Which specific graph problem do we want to solve?

SSSP with just BFS!

How can we elaborate more on what algorithm should we use?

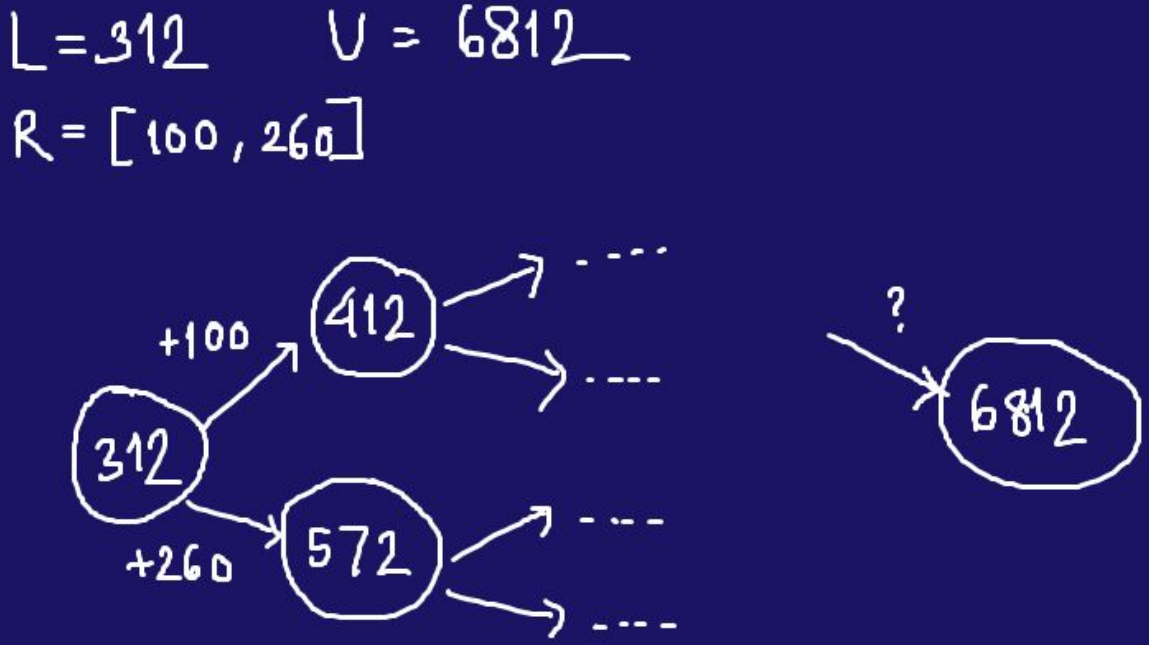
Initialize D with all values = INF except 0 at L.
Start from L, run BFS until we reach U, then output D[U].

Any edge cases I can consider further?

See next slides.

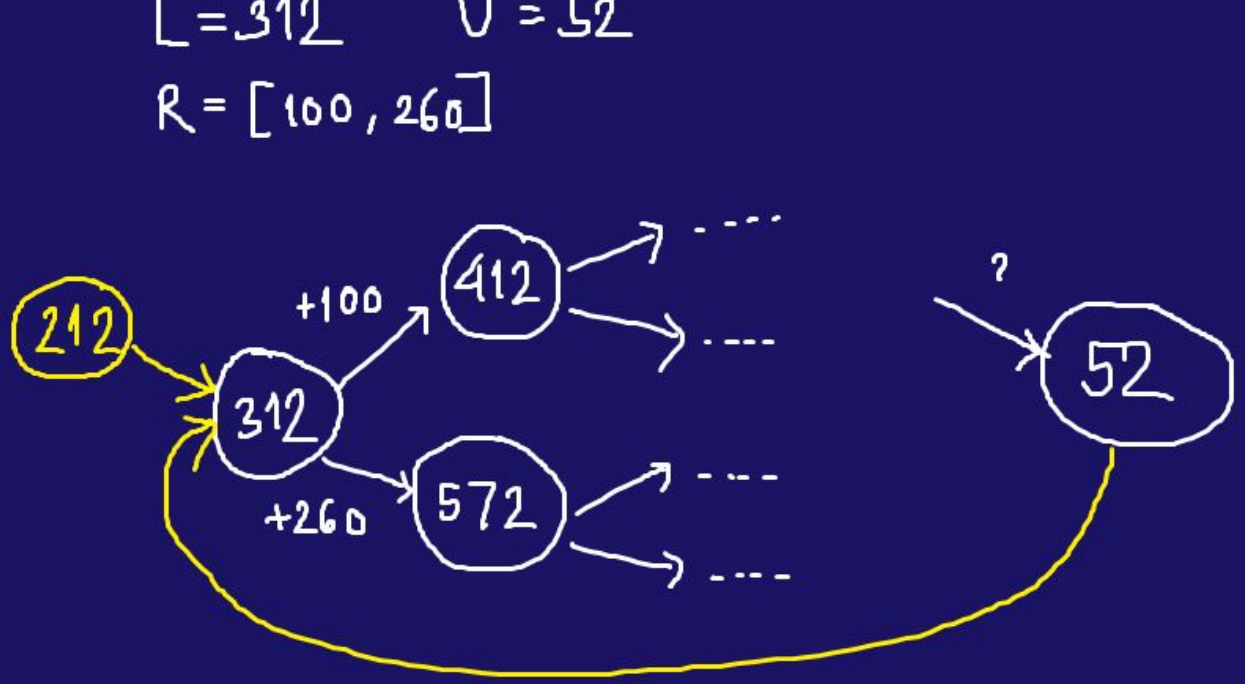
Proof without words

Sketching helps us visualize what kind of algorithm we will use.



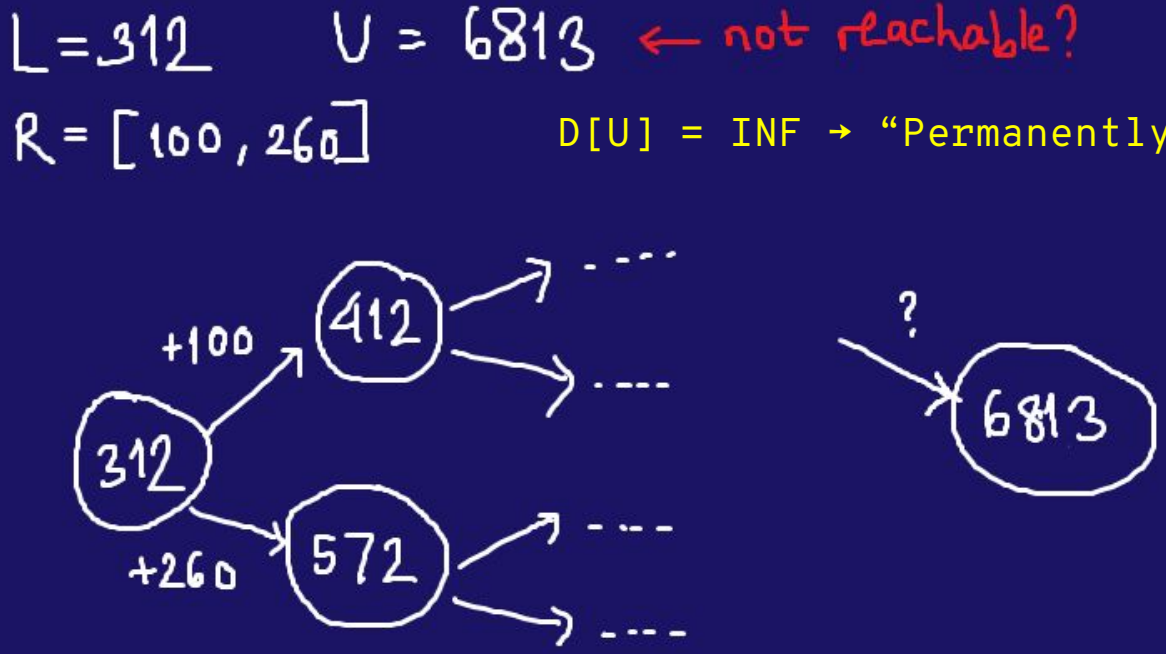
Proof without words

Sketching helps us visualize what kind of algorithm we will use.



Proof without words

Sketching helps us visualize what kind of algorithm we will use.



02

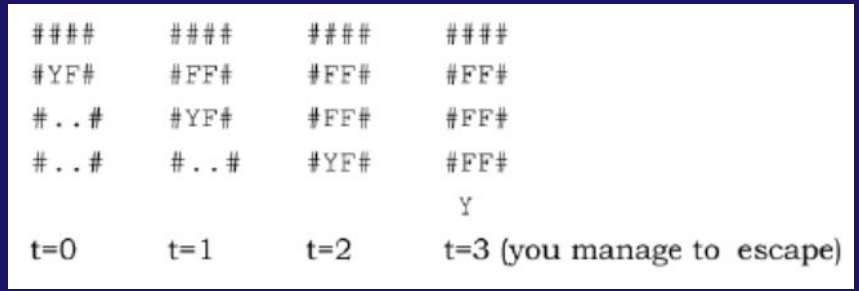
ESCAPE PLAN

Escape Plan

(2012/2013 CS2010 WQ2)
Abridged problem description:
Given an R x C grid.
: wall, . : passable square, F : fire, Y : you

You must escape through the border of the grid (not wall). Both you and the fire move 1 square per time unit. Fire moves in all 4 directions (up, right, left, down). Both you and fire cannot go through the walls.

You must determine whether you can get out of the maze safely before the fire reaches you; and if you can, how fast you can do it?



Thinking process

(Q2a) What do the vertices and edges in the graph represent?

Vertices: non-wall cells in the grid, up to RC of them
Edges: A → B if B is at A's NSEW direction and both are not walls, unweighted up to 4 per vertex

(Q2b) Which specific graph problem do we want to solve?

MSSP (Multi Source SPs) on unweighted graph! The sources are all the initial Y's and F's. (we're not restricted to just SSSP :D)

(Q2c) How can we elaborate more on what algorithm should we use?

See next slide.

Thinking process

(Q2c) How can we elaborate more on what algorithm should we use?

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 in the next slide.

Thinking process

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
```

Thinking process

(Q2c) How can we elaborate more on what algorithm should we use?

After running BFS algorithm from multiple sources, all dotted cells in the map are eventually replaced by a ‘Y’ or an ‘F’. This means each of the RC 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 4RC.

Total time complexity is O(RC).

Thinking process

(Q2c) How can we elaborate more on what algorithm should we use?

Alternatively, you can run BFS twice:

- Run BFS with Y as the single source. Store all shortest path estimates in an array D_Y.
- Run BFS with F as multiple sources. Store all shortest path estimates in an array D_F.
- Among all the boundary cells i where $D_Y[i] < D_F[i]$, find the one with the lowest D_Y, say x. Output x + 1.
- If there isn't any cell that satisfies the condition above, it is impossible to escape.

Kattis version of the exact same problem:
https://open.kattis.com/problems/fire2
https://open.kattis.com/problems/fire3
https://open.kattis.com/problems/slikar

03

MONEY CHANGER

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.

For example: n = 3 currencies and m = 3
- 1 USD gives us 0.8 Euro
- 1 Euro gives us 0.8 GBP (British pound sterling)
- 1 GBP gives us 1.7 USD

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. Profit!

Can you model this in a graph 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.

Thinking process

What do the nodes and edges in the graph represent?

Nodes: each currency
Edges: A → B if we can exchange from A to B
Weight: exchange rate

Which specific graph problem do we want to solve?

SSSP Bellman Ford!

How can we elaborate more on what algorithm should we use?

If we can multiply all edge in a cycle such that the result is > 1, then it is possible.
Bellman Ford only checks for negative sum of edge weights in a cycle. You want positive product of edge weights in a cycle. How to modify?

Money Changer

Say a profitable cycle has the path u_1, u_2, …, u_k, and back to u_1.
Thus we must have

$$w(u_1, u_2) \times w(u_2, u_3) \times \dots \times w(u_k, u_1) > 1$$

Equivalent to

$$\log w(u_1, u_2) + \log w(u_2, u_3) + \dots + \log w(u_k, u_1) > 0$$

Equivalent to

$$-\log w(u_1, u_2) - \log w(u_2, u_3) - \dots - \log w(u_k, u_1) < 0$$

→ negative sum of weights in a cycle!

Money Changer

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 $O(VE) = O(nm)$.

Similar question that needs this kind of transformation: https://open.kattis.com/problems/getshorty

04

HEIGHTS

Heights

A cat does not know the absolute height of a 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 $A \neq B$.

There are 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.

Thinking process

What do the nodes and edges in the graph represent?

Nodes: the areas
Edges: $A_i \rightarrow B_i$ if we have the information of the height difference
Weight: H_i
Note that you can also connect B_i to A_i with weight $-H_i$

Which specific graph problem do we want to solve?

See next slides.

How can we elaborate more on what algorithm should we use?

See next slides.

Simple Solution

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.

Thoughts?

The time complexity is thus O(NQ) time for Q queries, which is inefficient. Instead, notice that we need not run DFS multiple times.

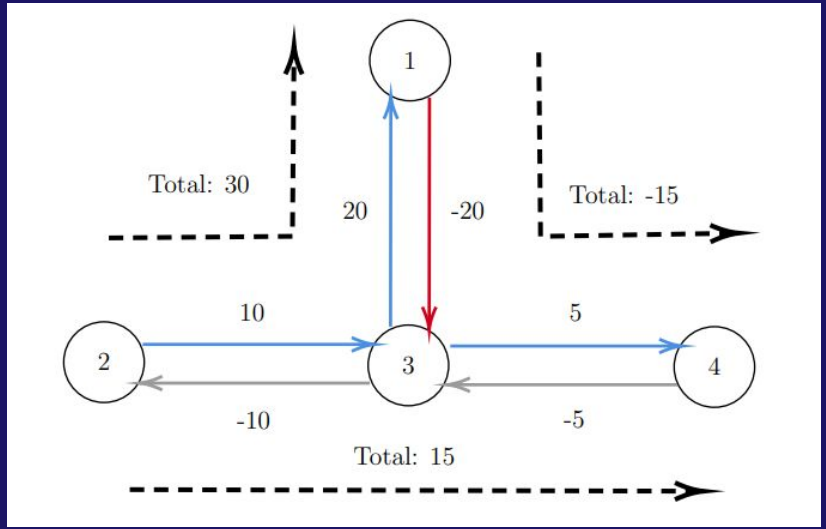
Better Solution

Observation:
Running BFS/DFS from X gives us:

- The distance from X to any vertex.
- The distance from any vertex to X (by negation).

This is because 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.

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.



Better Solution

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 → 1 and 1 → 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.

THE END!