DAA Implementation Project

IMT2019003 - Aditya Vardhan
IMT2019012 - Archit Sangal

13 May 2021

1 Question

Consider an undirected graph containing N nodes and M edges. Each edge M_i has an integer cost, C_i , associated with it.

The penalty of a path is the **bitwise OR** of every edge cost in the path between a pair of nodes, A and B. In other words, if a path contains edges $M_1, M_2, \cdots, M_k$, then the penalty for this path is C_1 OR C_2 OR $\cdots$ OR C_k.

Given a graph and two nodes, A and B, find the path between A and B having the minimal possible penalty and print its penalty; if no such path exists, print -1 to indicate that there is no path from A to B.

Note: Loops and multiple edges are allowed. The **bitwise OR** operation is known as or in Pascal and as | in C++ and Java.

2 Constraints

- $1 \leq N \leq 10^3$
- $1 \leq M \leq 10^4$
- $1 \leq C_i < 1024$
- $1 \leq A, B \leq N$
- $A \neq B$

3 Explanation

Precursors to the algorithm: Trivial and intuitive but important to be noticed

1. Let $A_1, A_2, \ldots, A_m$ be m positive integers and let B be such that

$$A_1 \vee A_2 \vee \cdots \vee A_m = B$$

   Let $A_{max} = \max\{A_i \; \forall i = 1 \text{ to } m\}$ and let the minimum number of bits needed to store $A_{max} = p$, then:-

   (a) minimum number of bits needed to store maximum possible $B = p$

   (b) $B < 2^p$

   (c) $B \geq A_i \; \forall i = 1$ to m

2. Now, we'll try to get an upper bound for any possible path in the graph.

   According to question, the maximum possible cost of an edge $= C_{max} = 1023$ (from the constraints)
   $\rightarrow$ minimum number of bits needed to store $C_{max} = \lfloor \log_2 C_{max} \rfloor + 1 = 10$

   Consider any arbitrary path p, with the cost(weight) of the sequence of edges in that path being $C_1, C_2, \ldots, C_m$. Let weight of path p be denoted by w_p

$$\rightarrow C_1 \vee C_2 \vee \cdots \vee C_m = w_p$$

   From precursor 1(a) and 1(b) we've:-

   (a) minimum number of bits to store maximum possible $w_p = \lfloor \log_2 C_{max} \rfloor + 1 = 10$

(b) $w_p < (2^{10} = 1024)$

3. There should **not** exist any cycle along a shortest path(using **or**).

   Proof: For now, let us assume that there exists a shortest path p, with a cycle, from some source node s to some destination node d.

   Let the part of path p, excluding the cycle, be p_{simple} and let the cycle be represented as p_{cycle}. So,

   $$weight(p) = weight(p_{simple}) \vee weight(p_{cycle})$$

   But from precursor 1(c) $weight(p) \geq weight(p_{simple})$

   $$\rightarrow \exists \text{ a path } p_{simple} \text{ from } s \text{ to } d\text{, which is shorter(using } \textbf{or} \text{ operator) than path } p$$

   This contradicts our assumption of p being the shortest path.

   $$\rightarrow \text{ shortest paths shouldn't contain cycles (Q.E.D)}$$

Algorithm: As we need to calculate the shortest path using the **bitwise** OR operator, we look at the weight of the shortest path in terms of binary form, rather than just a simple decimal. From precursor 2(a), we see that 10 bits are enough to store any weight of a path from a source s to destination d.

So, we iterate over each of the 10 bits and try to find a path that minimizes the currently iterated bit(i.e. set to 0) if it is possible, using DFS(with small modification); keeping in constraint the previously iterated bits.

To understand this, let us say we've a binary form of the shortest path that we've yet to calculate, given as

$$x_1\ x_2 \ldots x_{i-1}\ x_i \ldots x_9\ x_{10}$$

where x_1 represents the most significant bit(MSB)
while x_{10} represents the least significant bit(LSB)

Let us say, we're currently iterating over the i^{th} bit. This means that the first $i-1$ MSBs are already set. We then **define an integer variable** num whose binary form will be as follows:-

1. first $i-1$ MSBs are $x_1\ x_2 \ldots x_{i-1}$ respectively
2. ith bit is 1
3. rest of the $(10-i)$ LSBs are 0

Now, to run the modified DFS with the given source node s as the starting node, we maintain a variable $DFSflag$ and 2 arrays $V[\,]$ and $dist[\,]$ such that:-

1. $V[v]$ is true if node v has been visited. We visit only the non-visited nodes to avoid cycles as mentioned in precursor 3.
2. This is the small modification in the DFS that we talked about. $dist[v] = dist[u] \vee weight(u, v)$ only if both the below conditions are satisfied:-

   (a) **condition P:** $dist[u] \vee weight(u, v) < num$

   (b) **condition Q:** $weight(u, v) \mid 2^{10-i} \neq weight(u, v)$

   This is equivalent to saying that we extend the path s to u by adding an edge v to it, only when binary form of the weight of path $s \sim> u- > v$ has:-

   (a) first $i-1$ MSBs = already set $x_1\ x_2 \ldots x_{i-1}$

   (b) i^{th} bit = 0

   Note:- dist[s] = 0

   Since u and v are arbitrary, it means the conditions should be followed **each time** we extend the path.

   In the code, what we've done is instead of doing:-

```
    if(P && Q){
        extend path;
    }
```

We've done:-

```
if(!P || !Q){
    continue;
} else {
    extend path;
}
```

Needless to say, both are equivalent.

3. For ith bit $DFSflag$ is set to true only if we can form a path from s to d by extensions starting from s; following the 2 conditions each time we extend the sub-path.

The DFS function returns the value of the $DFSflag$, indicating if d is reachable from s or not, given that we abide by the conditions and constraints.

Conclusion:-

1. Before iterating over the 10 bits, to find the shortest path, there should at least exist a path p connecting s to d, in the first place.

   We check this by passing $num = 1024$ to our algorithm. This is equivalent to saying that $w_p < 1024$, which is true, as we already know from precursor 2(b); if d is connected to s by **any** path. In this case, our modified DFS works exactly like a simple DFS and returns false if a path doesn't exist from s to d. We then print "-1" to the console and our algorithm ends. Otherwise, true is returned and our algorithm continues.

2. While iterating for the i^{th} bit, if at all there exists a path such that binary form of its weight retains the already set $i - 1$ bits with i^{th} MSB $= 0$, our algorithm will return true. Hence, we greedily minimize each bit of the 10 bits binary form of the shortest path(using 'or').

3. From conclusions 1 and 2, we can surmise our algorithm to be feasible.

4 Code With Comments

```java
import java.util.*;
import java.io.*;

public class min_penalty_path {

    // each "Edge" object is related to the its outgoing node and its corresponding edge-weight
    // for ex:- for edge e = (u,v) with weight 8, we'll have
    // 1) e.vertex = v and 2) e.weight = 8
    static class Edge {
        int vertex;
        int weight;

        // Constructor
        public Edge(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }
    }

    static class Graph {
        int vertices; // stores the number of vertices in the graph i.e. |V|
       Boolean DFS_flag;
        LinkedList<Edge> [] adjacencylist;

        // Constructor
        Graph(int vertices) {
            this.vertices = vertices;
            adjacencylist = new LinkedList[vertices];
            //initialize adjacency lists for all the vertices
            for (int i = 0; i <vertices ; i++) {
```

```java
            adjacencylist[i] = new LinkedList<>();
        }
    }

    // adds an edge to the graph
    // Since the graphs are undirected, we should have a 2-way edge
    // for each undirected edge added to the graph
    public void addEdge(int vertex1, int vertex2, int weight) {
        Edge edge1 = new Edge(vertex2, weight);
     adjacencylist[vertex1].addFirst(edge1);

     Edge edge2 = new Edge(vertex1, weight);
     adjacencylist[vertex2].addFirst(edge2);

    }

    // DFSvisit is the recursive DFS-subroutine called in "DFS" function
    /* "num" acts as an upper-bound for each bit that we perform DFS. How its value
    is decided, has been discussed in the pdf document */
  public void DFSvisit(int u, int destination, Boolean V[], int num, int dist[],int bit_index){
     V[u] = true;

     for(Edge e: adjacencylist[u]){
        if(!V[e.vertex]){ // if the current vertex hasn't been visited yet

              /**
              IF:-
              1. weight of the path(using "or" operator) from s(source node)
                 to node "u" is not less than "num"
                                    OR
              2. the ith most significant bit, for which we're currently
                 iterating, turns out to be "1", in the weight of the path(using "or")
                 from "s" to "u"
                           --------------X-------------
                 then continue
              **/
           if(((dist[u] | e.weight) >= num) || ((e.weight|(int)Math.pow(2,bit_index)) == e.weight)){
              continue;

           } else {
              dist[e.vertex] = (dist[u] | e.weight);
              if(e.vertex == destination){
                 DFS_flag = true;
              }
           }
           DFSvisit(e.vertex, destination, V, num, dist,bit_index);
        }
     }
  }

  public Boolean DFS(int s, int destination, int num, int bit_index){
     Boolean[] V = new Boolean[vertices]; // if node "u" has been visited then
                                    //  V[u]=true, false otherwise

        // dist[] stores the distance(calculated using the OR operator) of each node,
        // each time the DFS is called, to decide the value of each bit.

        /*
        NOTE:-

        1. Although, dist[] might seem analogous to the D[] array used for
           solving the general single source shortest path problem, it is actually NOT.
        2. At the end of the algorithm, dist[] does NOT store the shortest(using OR) path
           from the source node.
        3. dist[] is used only while the algorithm is running. Once the running
```

```
            of the algorithm is complete, dist[] is of no use.
         */
         int[] dist = new int[vertices];

         /*
         (DFS_flag=true) only when we get a path "p" from source node "s"
         to destination node "d" such that:-
         1. the weight of "p"(using "or") < num
                                AND
         2. the ith most significant bit, for which we're currently
            iterating, turns out to be "0", in the weight of the path(using "or")
            from "s" to "d"
         */
         DFS_flag = false;

      for(int i = 0; i < vertices; i++){
         V[i] = false;
      }

         // calls the recursive sub-routine
      DFSvisit(s,destination,V,num,dist,bit_index);
      return DFS_flag;
     }
}

public static void main(String[] args) {

    // reading the inputs
    FastScanner scanner = new FastScanner(System.in);
    int N, M;
    N = scanner.nextInt();
    M = scanner.nextInt();
    Graph graph = new Graph(N+1);
    int vertex1, vertex2, weight;

    for(int i = 0; i < M; i++){
        vertex1 = scanner.nextInt();
        vertex2 = scanner.nextInt();
        weight = scanner.nextInt();
        graph.addEdge(vertex1, vertex2, weight); // adding each undirected edge to the graph
    }

    int source, destination;
    source = scanner.nextInt();
    destination = scanner.nextInt();

    // if the destination node "d" is not reachable from source node "s"
    // We have taken 'upper-limit' as 1024, as max possible distance is 1023 when all the bits are '1'
        and so we have taken 11 th bit as zero
    if(!graph.DFS(source, destination, 1024,11)){
        System.out.println("-1");
        return;
    }

    int answer = 0; // stores the weight of the shortest path (using 'or')

    // minimizing each bit of the weight of path(using "or") from "s" to "d",
    // starting from the most significant bit(MSB) onwards
    for(int i=9;i>=0;i--){
        int num = (int)Math.pow(2,i);
        answer += num;

        if(graph.DFS(source, destination, answer, i)){
            answer = answer - num;
        }
```

```java
        }

        // printing the final weight of the shortest path(using "or") from "s" to "d"
        System.out.println(answer);

    }

    // this class is used only to read the
    // inputs "quickly" (faster than usual)
    static class FastScanner {
        BufferedReader br;
        StringTokenizer st;

        FastScanner(InputStream stream) {
            try {
                br = new BufferedReader(new
                    InputStreamReader(stream));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        String next() {
            while (st == null || !st.hasMoreTokens()) {
                try {
                    st = new StringTokenizer(br.readLine());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return st.nextToken();
        }

        int nextInt() {
            return Integer.parseInt(next());
        }
    }
}
```

5 Complexity (Time and Space)

1. **Space**

   (a) Adjacency list to store the graph takes $O(|V| + |E|)$ space

   (b) arrays $V[\ ]$ and $dist[\ ]$, which are created and destroyed in the iteration of each bit, both take $O(|V|)$ space.

$$\rightarrow \text{Total space used} = O(|V| + |E|) + O(|V|) + O(|V|) = O(|V| + |E|)$$

2. **Time**

   (a) For each bit we perform run our modified DFS. It is pretty easy to observe that the small modification we made to DFS won't have any onservable effect on the time complexity.

$$\rightarrow \text{Time to run the modified DFS } = O(|V| + |E|)$$

   (b) We run this modified DFS for each bit. Before doing this, we run it one extra time to check the existence of solution, to print "-1" in case it doesn't exist.

$$\begin{aligned}\text{No. of times we run modified DFS} &= (\text{minimum no. of bits to store the maximum possible weight}) + 1\\ &= (\lfloor \log_2 C_{max} \rfloor + 1) + 1\\ &= O(log\, C_{max})\end{aligned}$$

$$\therefore \text{Total time complexity of the whole algorithm } = O(\log C_{max}\,(|V| + |E|))$$

6 Proof

For proving we will use **Proof By Exchange Method**. Let the $G\{g_1, g_2, \cdots, g_{10}\}$ be the greedy solution given by our algorithm and $A\{a_1, a_2, \cdots, a_{10}\}$ be any other arbitrary (or optimal) feasible solution.

Assuming that optimal solution A is the same as our greedy solution G then we are done. If the optimal solution A is not the same as our greedy solution G then, without lose of generality, let i^{th} bit be the first point of difference between A and G, i.e. $a_1, a_2, \cdots, a_{i-1}$ is same as $g_1, g_2, \cdots, g_{i-1}$ and $a_i \neq g_i$. As a_i and g_i are bits so they can take only values 0 or 1. Therefore, there can only be these 4 cases -

- **Case I** : $a_i = 1$ and $g_i = 1$

  This is a contradiction as we have assumed that i^{th} bit be the first point of difference between A and G. Therefore, no need of exchanging a_i with g_i as they are same.

- **Case II** : $a_i = 0$ and $g_i = 0$

  This is a contradiction as we have assumed that i^{th} bit be the first point of difference between A and G. Therefore, no need of exchanging a_i with g_i as they are same.

- **Case III** : $a_i = 1$ and $g_i = 0$

  By definition of algorithm and as stated in conclusion, G is a feasible solution. As $g_i = 0$, therefore the maximum value G can take is 1 less than the minimum solution given by A when $a_i = 1$. As when $a_1, a_2, \cdots, a_{i-1}$ is same as $g_1, g_2, \cdots, g_{i-1}$ and $a_{i+1} = a_{i+2} = \cdots = a_k = 0$; $g_{i+1} = g_{i+2} = \cdots = g_k = 1$, hence -

  $$\text{Minimum Solution of } A = \text{Maximum Solution of } G + 1.$$

  Therefore, A is **not** the optimal solution, which is a contradiction. Therefore, if we exchange a_i (if A is any arbitrary solution) with g_i then we will get a better solution.

- **Case IV** : $a_i = 0$ and $g_i = 1$

  num is defined using $a_1, a_2, \cdots, a_{i-1}$ which is same for both A and G so ***num*** is also same for both A and G. By definition of algorithm, if g_i is 1, then g_i of the any path which exist between source and sink will not be 0 with weight of the path upper bounded by ***num***. Therefore, such a optimal solution will not exist with $a_i = 0$. Therefore, this will not be a feasible case.

Hence, eventually all the a_i are either same as g_i or will get exchanged so the solution G is indeed the optimal solution.

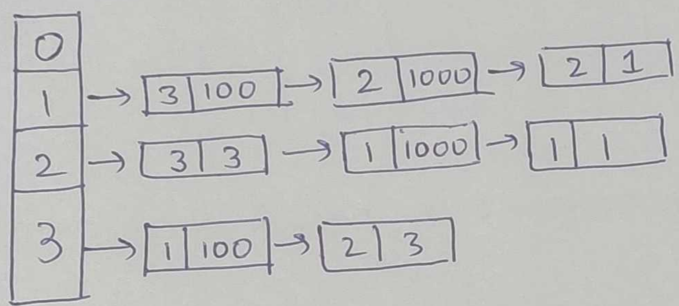
7 Dry Run

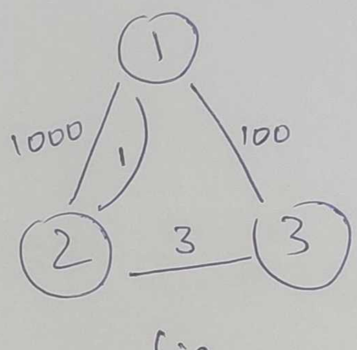
For input :-

3 4
1 2 1
1 2 1000
2 3 3
1 3 100
1 3

S = 1   d = 3 |V| = 3 |E| = 4

Adjacency list will be :-





fig

$$\underbrace{x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0}_{2^9 \; - - - - - - \; 2^1 \; 2^0}$$ → Binary form of the shortest path

∵ the check for existence of a solution works like a simple DFS, we're skipping that part. And as we can see from the fig that the graph is connected, so solution exists.

NOTE:-
We'll dry run only for a single iteration

Starting from ~~item~~ bit x_9

$\boxed{\text{Iteration 1}}$ $i = 9$; $num = 2^9 = 512$

answer = 0 + 512 i.e. $x_9 = 1$

Now, we start the modified DFS

DFS_flag = false

ITERATION-01

	0	1	2	3
V[]	false (F)	~~F~~ T(i)	~~F~~ T(vi)	~~F~~ T(iv)
dist[]	0	0	~~0~~ (v) 103	~~0~~ 100(ii)

DFS_flag = ~~F~~ T (iii)

V[1] set to true ; e = adjacencyList[1] = edge (1,3) with weight $\underline{\underline{100}}$

V[3] = false

((dist[1] | e.weight) = 100) < (num = 512)

$\underline{\underline{\text{and}}}$ $(\underbrace{\text{e.weight} \mid \text{num}}_{612}) \neq \underbrace{\text{e.weight}}_{100}$

$\Rightarrow$ dist[3] = (dist[1] | e.weight) = 100 & DFS_flag set to true

Now $S_1^1 = 3$

V[3] set to true; e = (3,1) with weight 100

But V[1] = true, so we see the next edge in the adjacency_list[3]

$\Rightarrow$ e = (3,2) with weight 3

V[2] = false

$(dist[3] | e.weight) < num$ <u>and</u> $e.weight | num \neq e.weight$

$\Rightarrow dist[2] = 103$ (updated)

$S_1^2 = 2$

$V[2]$ set to true ; $e = (2,3)$ with weight 3

But $V[3] = true \Rightarrow$ skip

Now, $e = (2,1)$ with weight 1000

but $V(1] = true \Rightarrow$ skip

$e = (2,1)$ with weight 1 but $V(1] = true \Rightarrow$ skip

this recursive inline fu^n ends

we're back to $S_1^1 = 3$ whose each edge has been visited. This recursive inline function also ends

We're back to $s = 1$

Now $e = (1,2)$ with weight 1000

but $V[2] = true \Rightarrow$ skip

then $e = (1,2)$ with weight $= 1$ but

$V[2] = true \Rightarrow$ skip

"DFS" returns (DFS_flag = true)

$\Rightarrow$ answer = answer − num = 0 i.e. x_9 set to 0

↓

We were able to minimize bit x_9

Similarly, we perform 9 more iterations & try to minimize $x_8, x_7, \ldots, x_1, x_0$ if possible

Finally our answer = 3 i.e. $x_9 = x_8 = \ldots = x_2 = 0$; $x_1 = x_0 = 1$