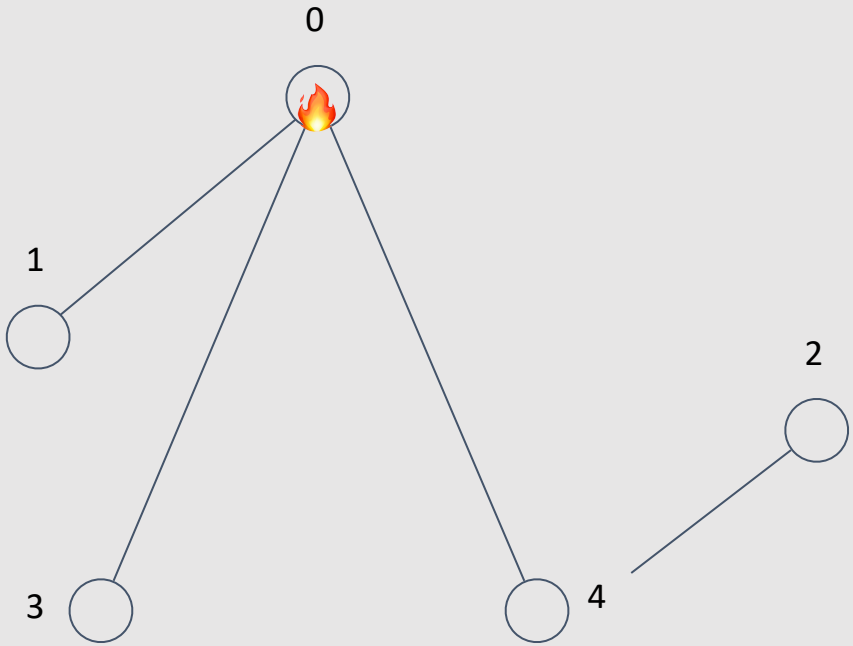
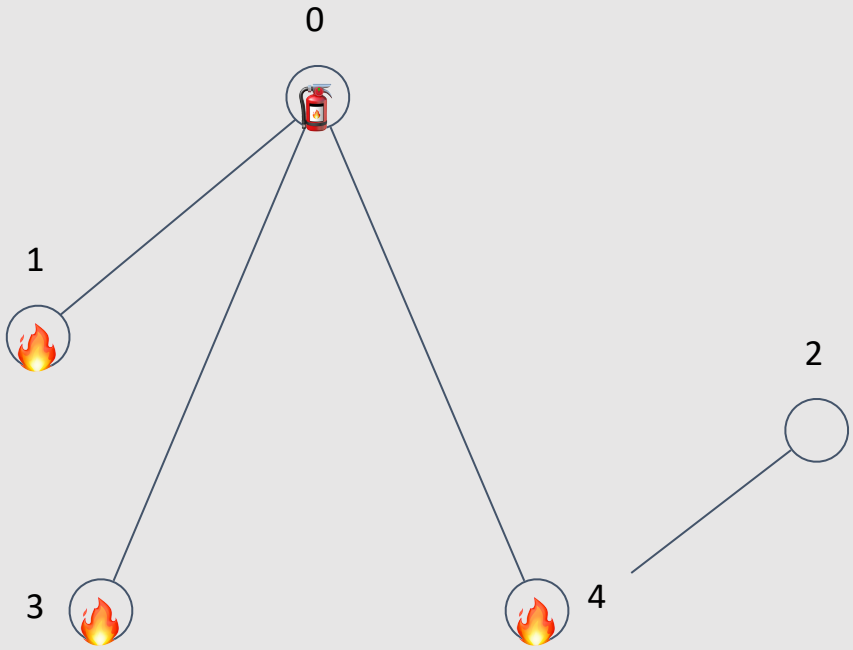
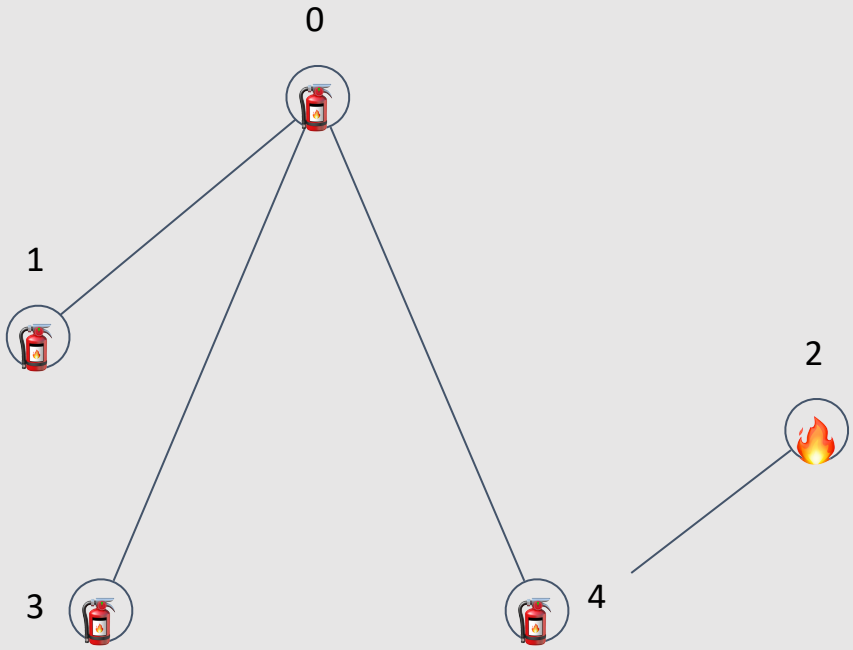
Graphs

BFS

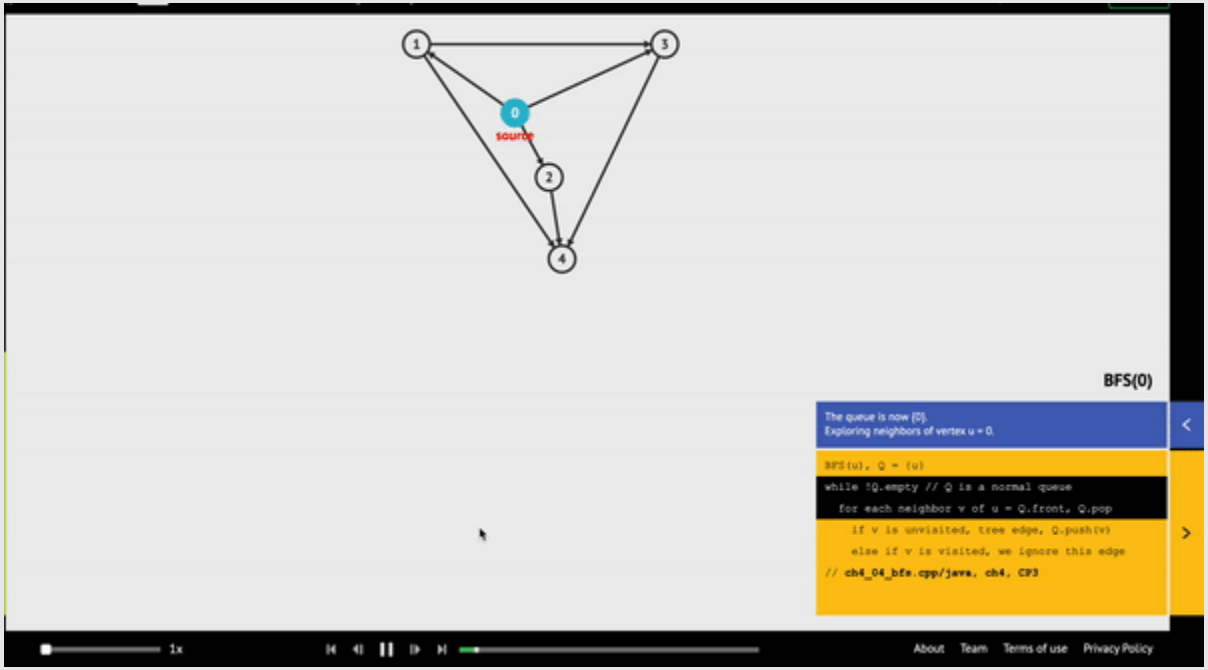
- We have graph <V, E>. We need to check all vertices.
- Let’s use the matches method. When you fire the match, the fire will go deeply.


0
1
2
3
4


0
1
2
3
4


0
1
2
3
4

Visualization



```cpp
#include <bits/stdc++.h>
using namespace std;

void bfs(int root, const vector<vector<int> > &g, vector<bool> &visited) {
    queue<int> vertices;
    vertices.push(root);
    while (!vertices.empty()) {
        auto u = vertices.front();
        visited[u] = true;
        vertices.pop();
        for (auto v: g[u]) {
            if (!visited[v]) {
                vertices.push(v);
                visited[v] = 1;
            }
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }
```

```
32.     }
33.
34.     bfs(0, graph, visited);
35.
36.     for (int i = 0; i < n; i++) {
37.         cout << visited[i] << " ";
38.     }
39.
40.     return 0;
41. }
```

Success #stdin #stdout 0.01s 5460KB

stdin

```
4 3
1 2
2 3
1 4
```

stdout

```
1 1 1 1
```

```python
def BFS(graph, start, visited):
    queue = deque([start])
    visited[start] = True

    while queue:
        vertex = queue.popleft()

        for neighbor in graph[vertex]:
            if not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True

n, m = map(int, input().split(' '))

Matrix = []
visited = [False] * n
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

BFS(Matrix, 0, visited)

print(visited)

```

Success #stdin #stdout 0.03s 9776KB comments (0)

stdin copy

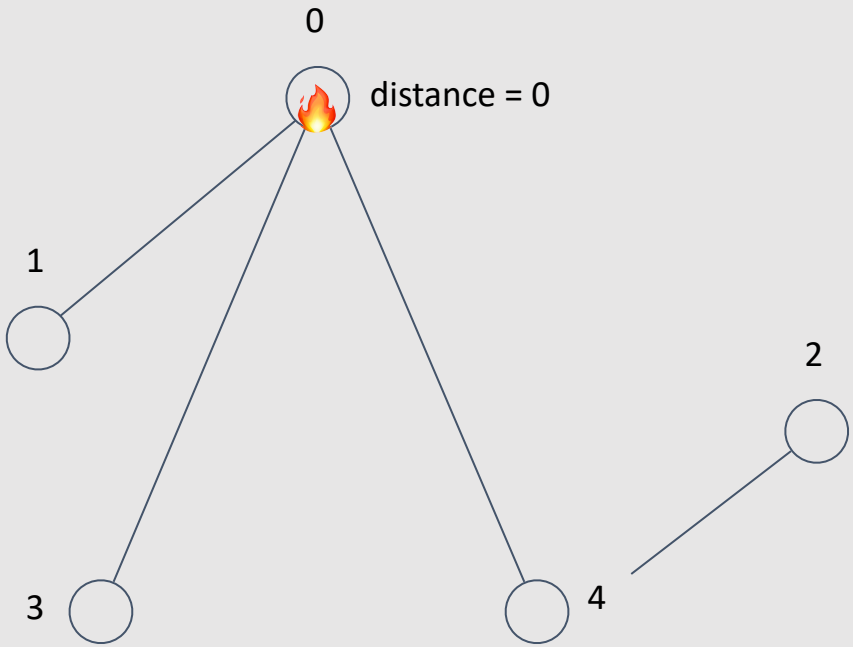
```
3 4
1 2
2 3
3 1
1 3
```

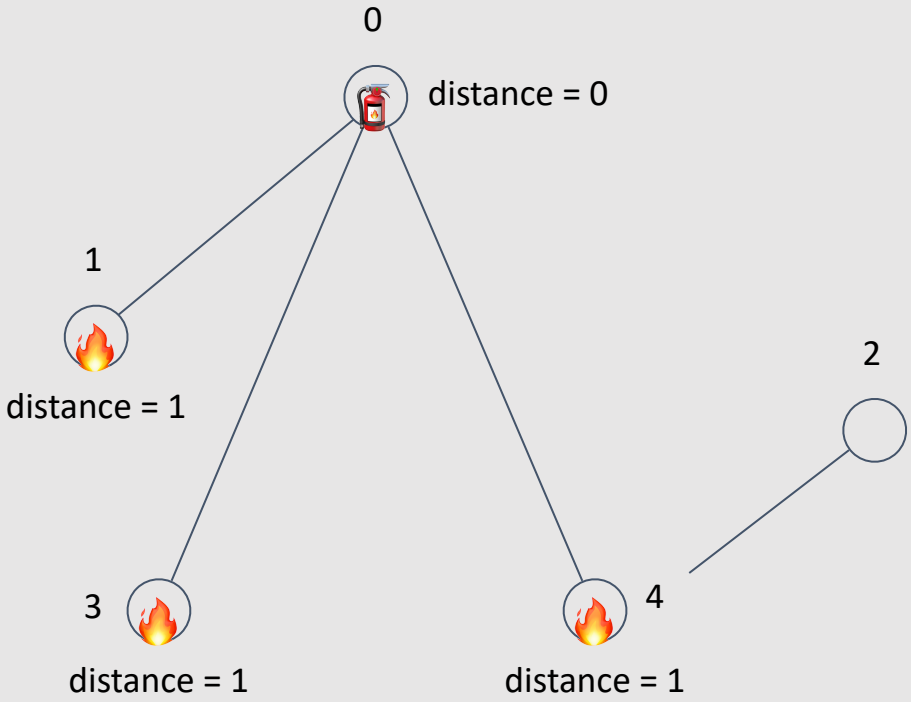
stdout copy

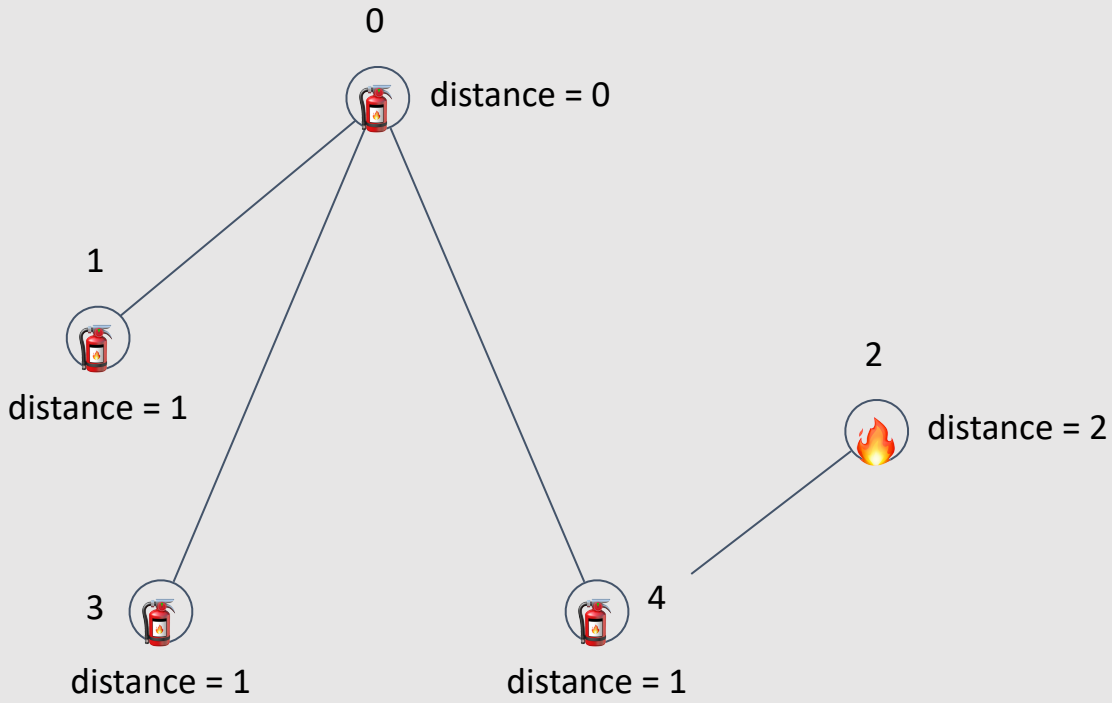
```
[True, True, True]
```

BFS

- We can find distance with BFS for example


0
distance = 0
1
2
3
4


0
distance = 0
1
distance = 1
2
3
distance = 1
4
distance = 1


0
distance = 0
1
distance = 1
2
distance = 2
3
distance = 1
4
distance = 1

```
5.  const int INF = 1e9;
6.
7.  void bfs(int root, const vector<vector<int> > &g, vector<int> &distance) {
8.      queue<int> vertices;
9.      vertices.push(root);
10.     distance[root] = 0;
11.     while (!vertices.empty()) {
12.         auto u = vertices.front();
13.         vertices.pop();
14.         for (auto v: g[u]) {
15.             if (distance[v] == INF) {
16.                 vertices.push(v);
17.                 distance[v] = distance[u] + 1;
18.             }
19.         }
20.     }
21. }
22.
23. int main() {
24.     int n, m;
25.     cin >> n >> m;
26.     vector<vector<int> > graph(n);
27.     vector<int> distance(n, INF);
28.
29.     for (int i = 0; i < m; i++) {
30.         int from, to;
31.         cin >> from >> to;
32.         from--;
33.         to--;
34.         graph[from].push_back(to);
35.     }
```

```
36.
37.     bfs(0, graph, distance);
38.
39.     for (int i = 0; i < n; i++) {
40.         cout << distance[i] << " ";
41.     }
42.
43.     return 0;
44. }
```

Success #stdin #stdout 0.01s 5392KB

stdin

```
4 3
1 2
2 3
1 4
```

stdout

```
0 1 2 1
```

```python
def BFS(graph, start, distance):
    queue = deque([start])
    distance[start] = 0

    while queue:
        vertex = queue.popleft()

        for neighbor in graph[vertex]:
            if distance[neighbor] == INF:
                queue.append(neighbor)
                distance[neighbor] = distance[vertex] + 1

n, m = map(int, input().split(' '))

Matrix = []
distance = [INF] * n
for i in range(n):
    Matrix.append([])

for i in range(m):
    from_, to = map(int, input().split(' '))
    Matrix[from_ - 1].append(to - 1)

BFS(Matrix, 0, distance)

print(distance)
```

Success #stdin #stdout 0.03s 9564KB

comments (0)

stdin

copy

```
3 4
1 2
2 3
3 1
1 3
```

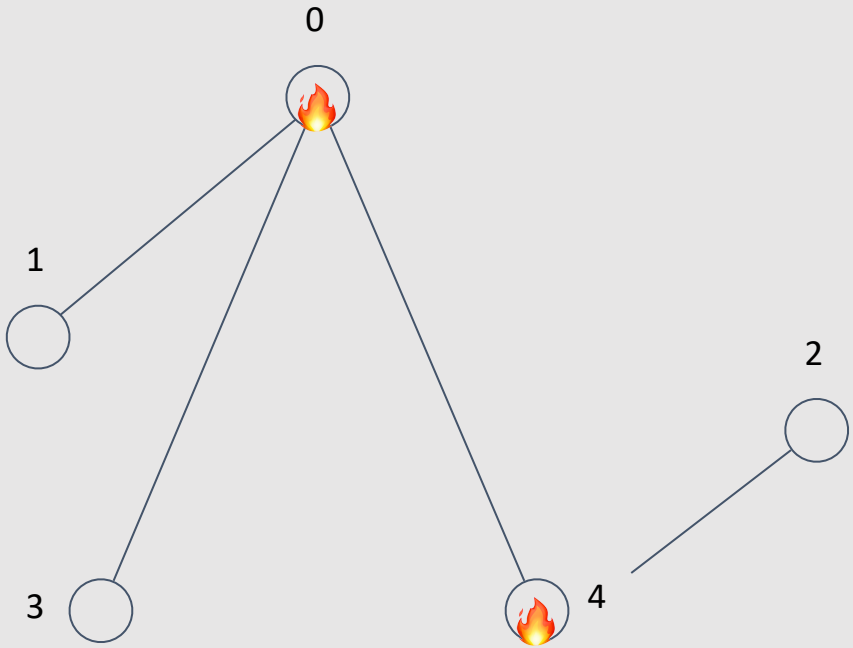
stdout

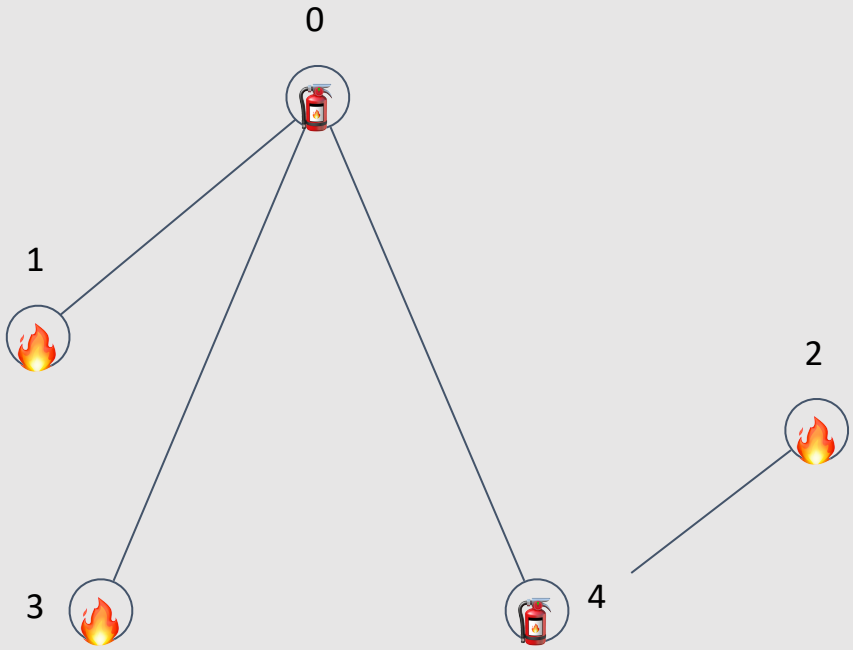
copy

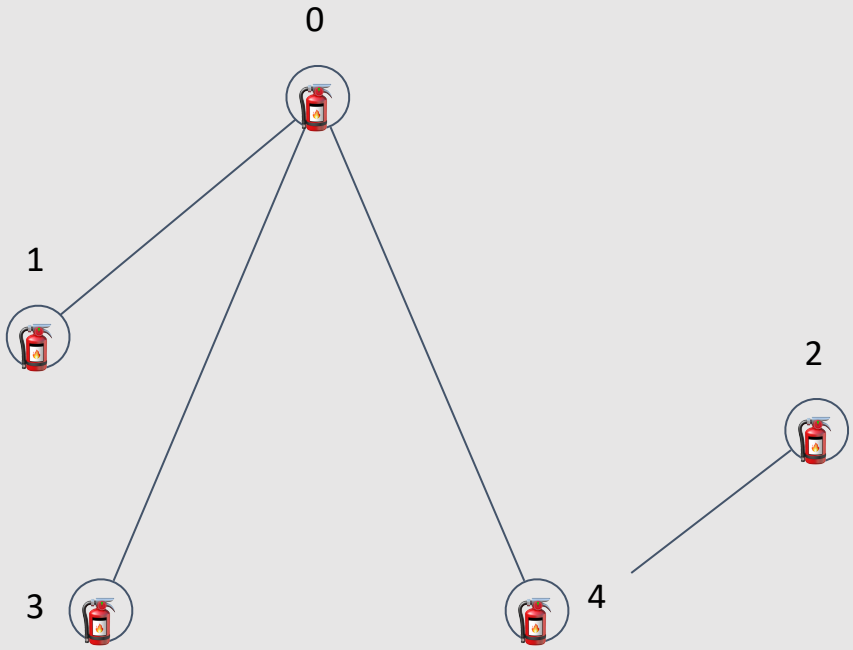
```
[0, 1, 1]
```

Task

- We now know that not just one vertex was set on fire, but several at once. We need to determine when each vertex will burn out.


0
1
2
3
4


0
1
2
3
4


0
1
2
3
4

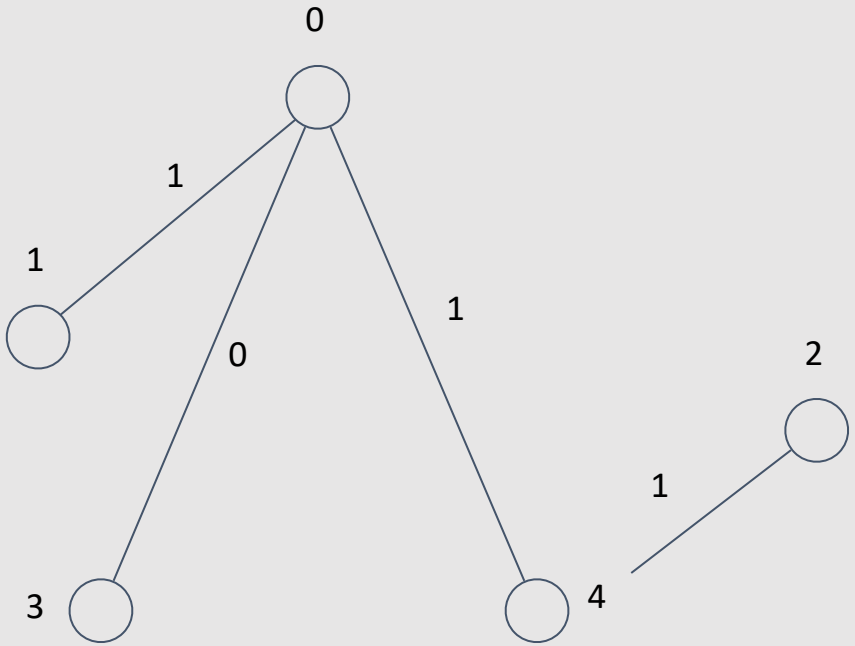
Solution

- We can add a fake vertex from which there are edges to all burnt vertices.
- We can add all burnt vertices to the queue.

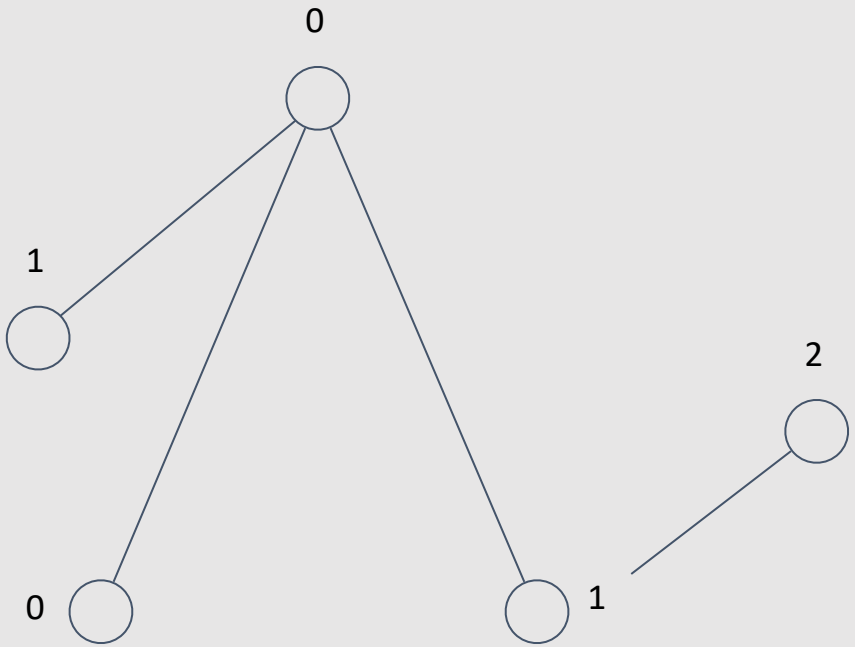
0-1 BFS

- Now the edges also have a weight - the cost of traveling along the edge.
- The weight can be either 0 or 1.
- It is required to calculate the sum of the numbers written on the edges, that is, for what total cost one can travel from vertex a to vertex b.

graph

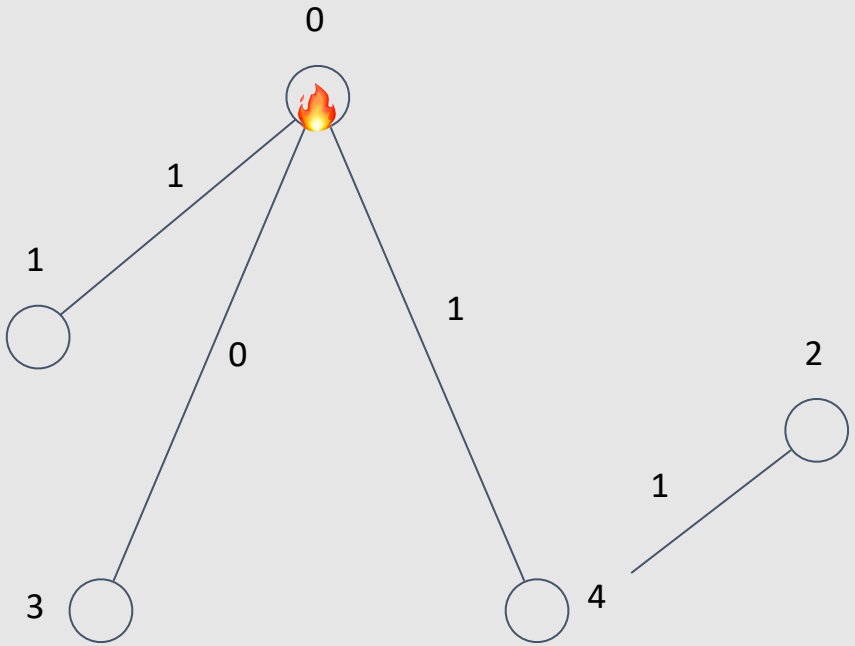


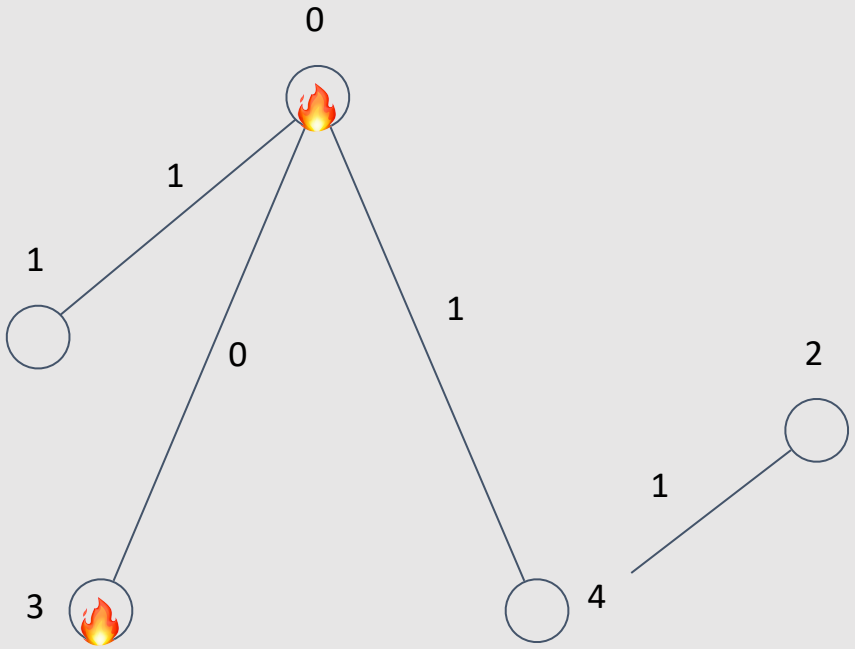
distance

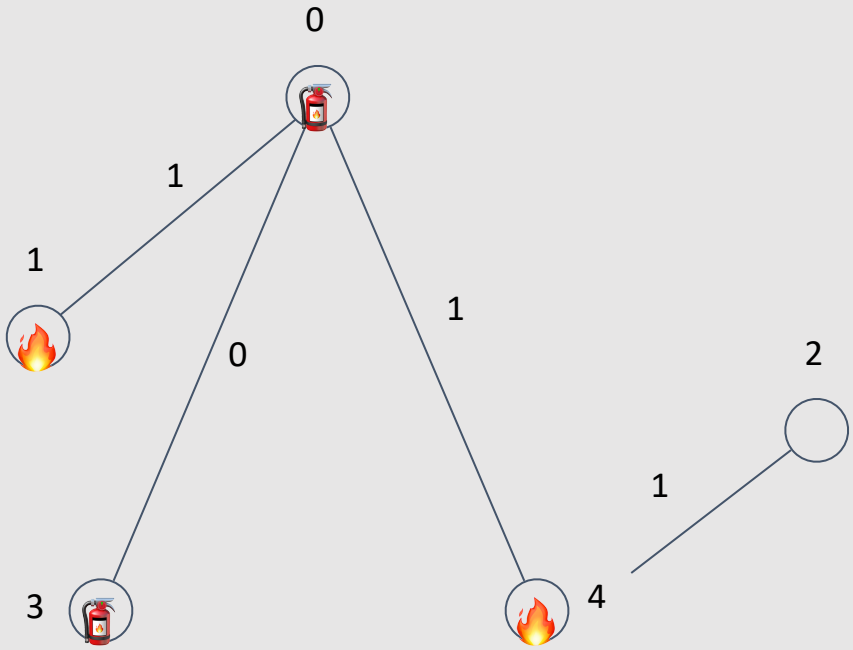


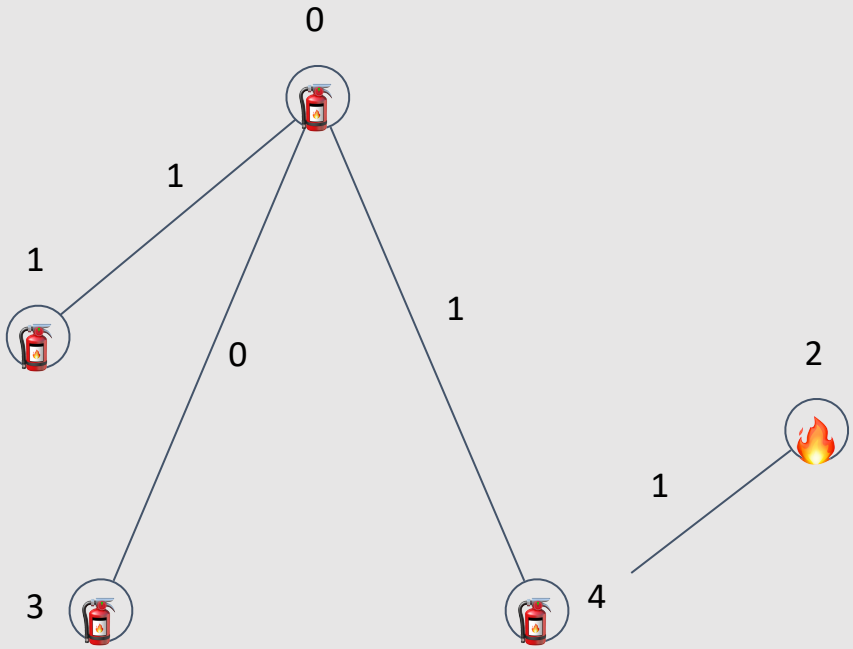
0-1 BFS

- Let’s use not a queue, but a deque.
- If the edge has a weight of 1, then, as before, add it to the end of the queue.
- However, if the edge has a weight of 0, it means that this edge can be added to the beginning of the queue, since it should be "burned" at the same time as the current vertex.


0
1
1
1
0
1
2
3
4


0
1
1
1
0
1
2
1
3
4


0
1
1
1
0
1
2
3
4


0
1
1
1
0
1
2
3
4

```cpp
const int INF = 1e9;

struct Edge {
    int to, w;
    Edge(int to, int w) {
        this->to = to;
        this->w = w;
    }
};

void bfs(int root, const vector<vector<Edge> > &g, vector<int> &dist) {
    deque<int> vertices;
    dist[root] = 0;
    vertices.push_back(root);
    while (!vertices.empty()) {
        auto u = vertices.front();
        vertices.pop_front();
        for (auto edge: g[u]) {
            if (dist[edge.to] > dist[u] + edge.w) {
                if (edge.w) {
                    vertices.push_back(edge.to);
                }
                else {
                    vertices.push_front(edge.to);
                }
                dist[edge.to] = dist[u] + edge.w;
            }
        }
    }
}

```

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<Edge> > graph(n);
    vector<int> dist(n, INF);

    for (int i = 0; i < m; i++) {
        int from, to, w;
        cin >> from >> to >> w;
        from--;
        to--;
        graph[from].push_back(Edge(to, w));
        graph[to].push_back(Edge(from, w));
    }

    bfs(0, graph, dist);

    for (int i = 0; i < n; i++) {
        cout << dist[i] << " ";
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5520KB

comments (0)

stdin

copy

```
5 4
1 2 1
1 4 0
1 5 1
5 3 1
```

stdout

copy

```
0 1 2 0 1
```

```python
from collections import deque

INF = 10**9

class Edge:
    def __init__(self, to, w):
        self.to = to
        self.w = w

def bfs(root, g, dist):
    vertices = deque([root])
    dist[root] = 0
    while vertices:
        u = vertices.popleft()
        for edge in g[u]:
            if dist[edge.to] > dist[u] + edge.w:
                dist[edge.to] = dist[u] + edge.w
                if edge.w:
                    vertices.append(edge.to)
                else:
                    vertices.appendleft(edge.to)

```

```
23. if __name__ == '__main__':
24.     n, m = map(int, input().split())
25.     graph = [[] for _ in range(n)]
26.     dist = [INF] * n
27. 
28.     for _ in range(m):
29.         from_vertex, to, w = map(int, input().split())
30.         from_vertex -= 1
31.         to -= 1
32.         graph[from_vertex].append(Edge(to, w))
33.         graph[to].append(Edge(from_vertex, w))
34. 
35.     bfs(0, graph, dist)
36. 
37.     print(*dist)
38. 
```

Success #stdin #stdout 0.04s 9708KB

comments (0)

stdin

copy

```
5 4
1 2 1
1 4 0
1 5 1
5 3 1
```

stdout

copy

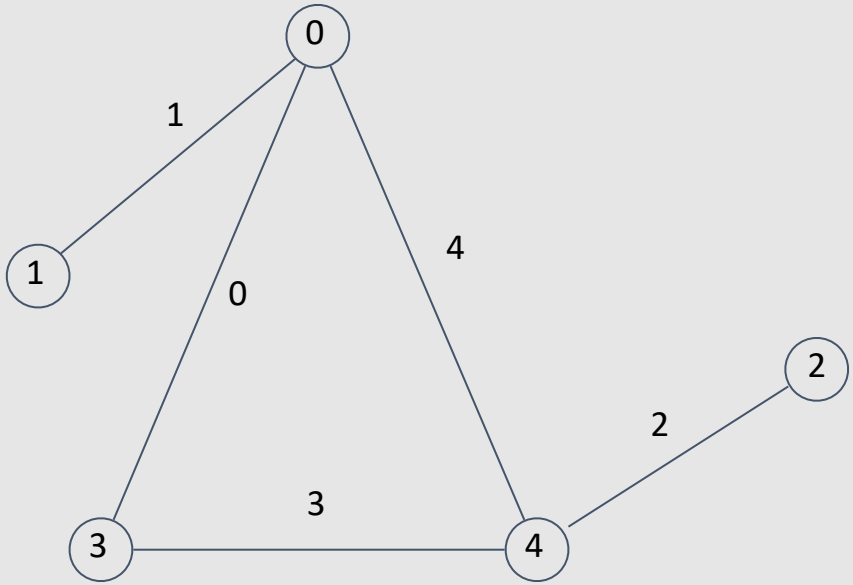
```
0 1 2 0 1
```

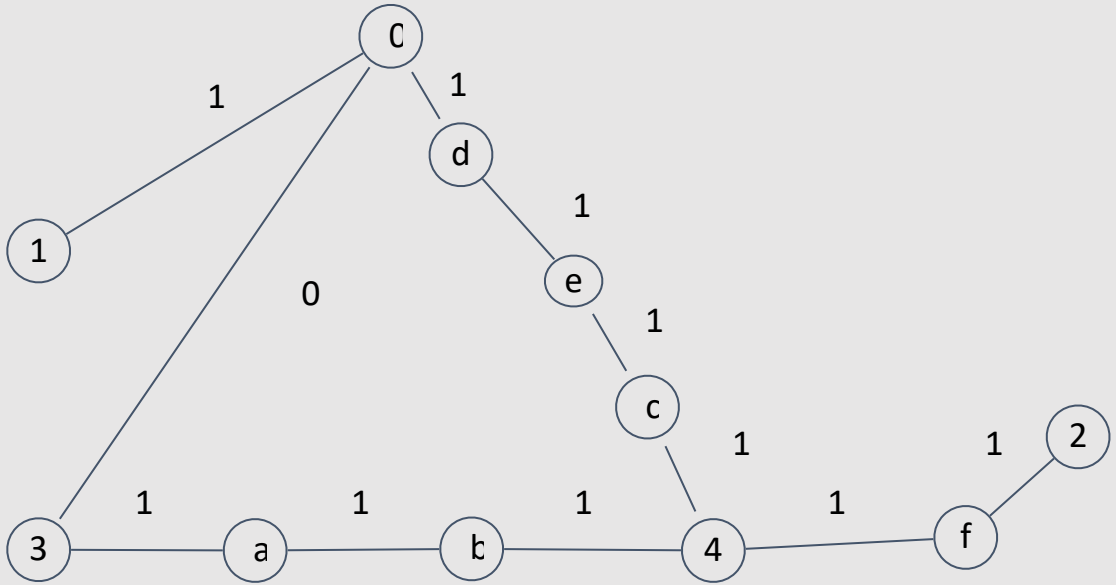
0-k BFS

- Now, let's assume we have distances from 0 to k. What should we do in this case?

Idea

- The simplest and most obvious solution would be to turn 1 edge into k edges and then run a standard BFS.


0
1
1
4
0
2
2
3
3
4

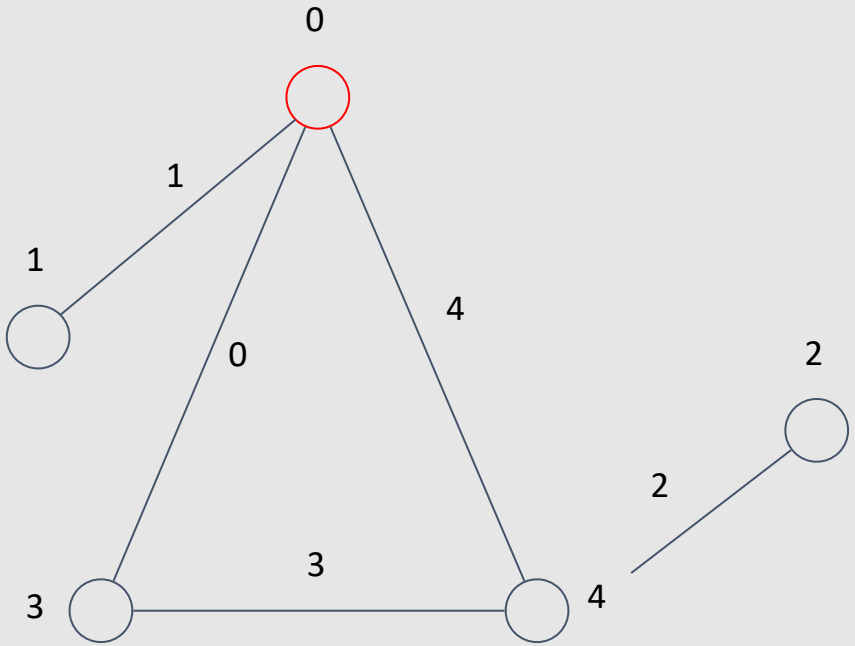

C
1
1
d
1
1
e
0
1
c
1
1
2
1
1
1
1
3
a
b
4
f

max distance = n * k

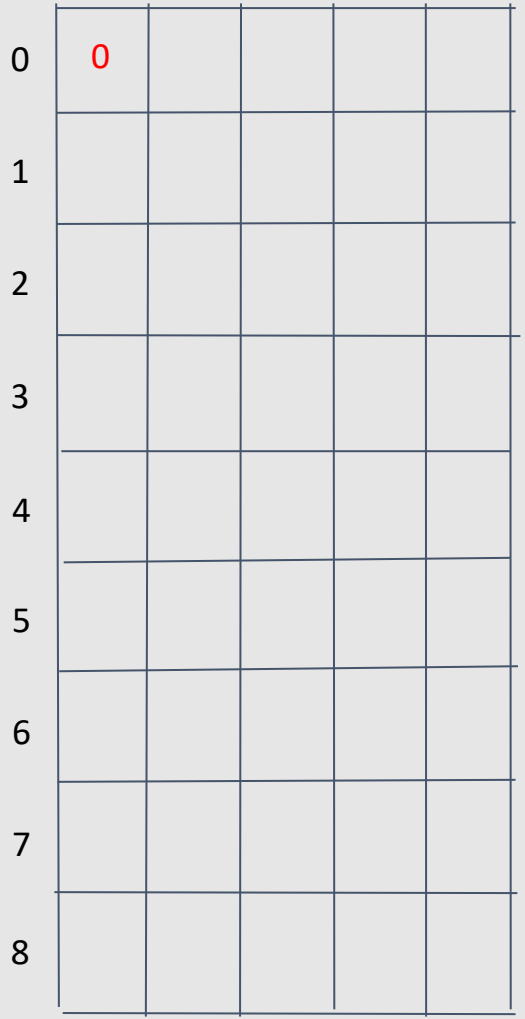
- The maximum distance in such a graph is (n - 1) * k, as the maximum number of edges is (n - 1), and the maximum edge weight is k.

Idea

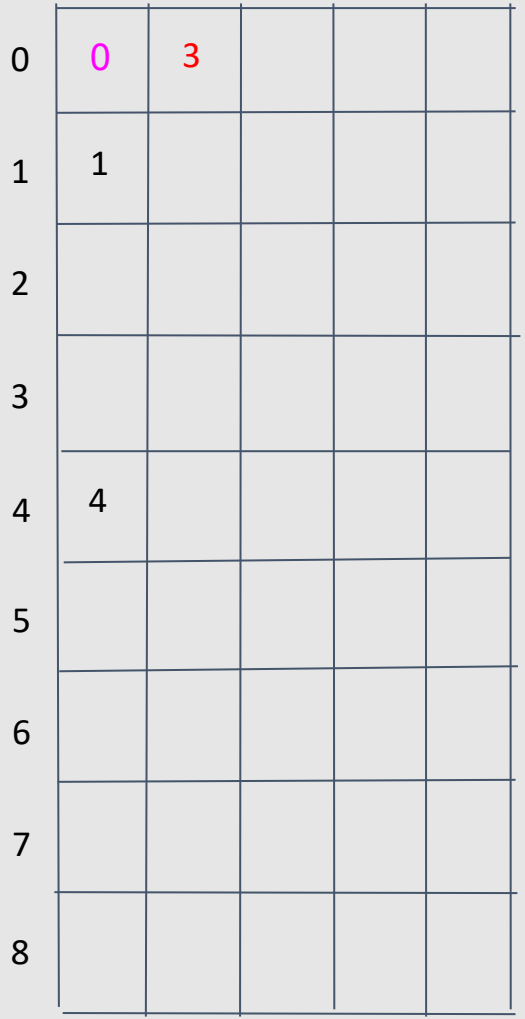
- Let's not have just one, but a whole n * k queues, where k is the maximum edge weight. Inside queue i, we will place all the vertices at that distance.


0
1
1
4
0
2
2
3
3
4

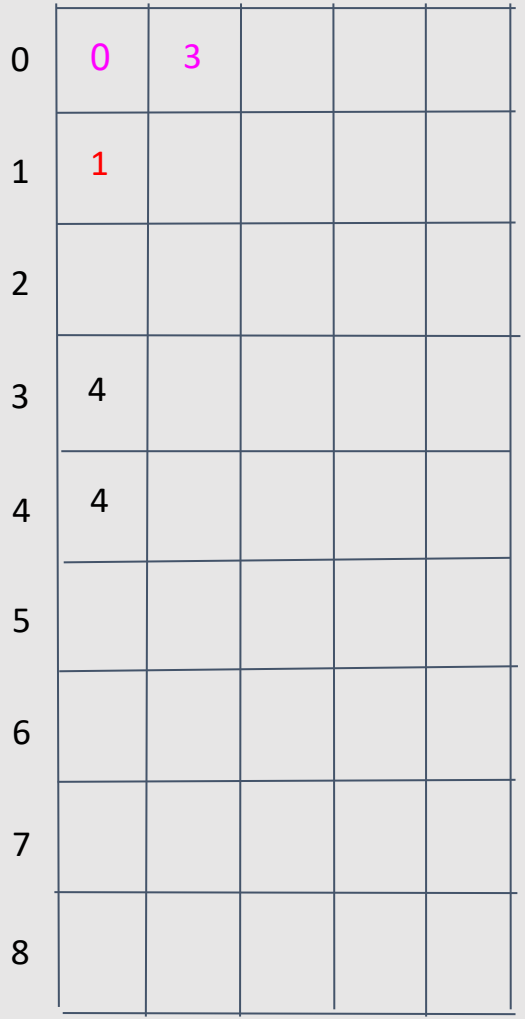
n = 5, k = 4



n = 5, k = 4



n = 5, k = 4



n = 5, k = 4

n = 5, k = 4

```python
from collections import deque

def bfs_0k(n, k, graph, start):
    queues = [deque() for _ in range(n * k)]
    distance = [float('inf')] * n
    distance[start] = 0
    queues[0].append(start)

    for d in range(n * k):
        while queues[d]:
            v = queues[d].popleft()
            for to, weight in graph[v]:
                next_d = d + weight
                if distance[to] > next_d:
                    distance[to] = next_d
                    queues[next_d].append(to)

    return distance

# Example Usage
n, m, k = map(int, input().split(' '))
graph = [[] for _ in range(n)] # Initialize graph
for _ in range(m):
    to, from_, weight = map(int, input().split(' '))
    to = to - 1
    from_ = from_ - 1
    graph[from_].append((to, weight))
    graph[to].append((from_, weight))
```

```
30.  # Call the BFS function
31.  start = 0 # Starting vertex
32.  distances = bfs_0k(n, k, graph, start)
33.  print(distances)
34.
```

Success #stdin #stdout 0.03s 9608KB

stdin

```
5 5 4
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

```
[0, 1, 5, 0, 3]
```

```cpp
void bfs_0k(int n, int k, const vector<vector<pair<int, int>>>& graph, int start) {
    vector<queue<int>> queues(n * k);
    vector<int> distance(n, INT_MAX);
    distance[start] = 0;
    queues[0].push(start);

    for (int d = 0; d < n * k; ++d) {
        while (!queues[d].empty()) {
            int v = queues[d].front();
            queues[d].pop();
            for (auto& edge : graph[v]) {
                int to = edge.first, weight = edge.second;
                int next_d = d + weight;
                if (distance[to] > next_d) {
                    distance[to] = next_d;
                    queues[next_d].push(to);
                }
            }
        }
    }

    // Print distances
    for (int dist : distance) {
        cout << dist << " ";
    }
    cout << endl;
}

int main() {
    int n, m, k;
    cin >> n >> m >> k;
    vector<vector<pair<int, int>>> graph(n);

    for (int i = 0; i < m; ++i) {
        int from, to, weight;
        cin >> from >> to >> weight;
        --from;
        --to;
```

```
36. int main() {
37.     int n, m, k;
38.     cin >> n >> m >> k;
39.     vector<vector<pair<int, int>>> graph(n);
40.
41.     for (int i = 0; i < m; ++i) {
42.         int from, to, weight;
43.         cin >> from >> to >> weight;
44.         --from;
45.         --to;
46.         graph[from].push_back({to, weight});
47.         graph[to].push_back({from, weight});
48.     }
49.
50.     int start = 0;
51.     bfs_0k(n, k, graph, start);
52.     return 0;
53. }
54.
```

Success #stdin #stdout 0.01s 5380KB

comments (0)

stdin

copy

```
5 5 4
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

copy

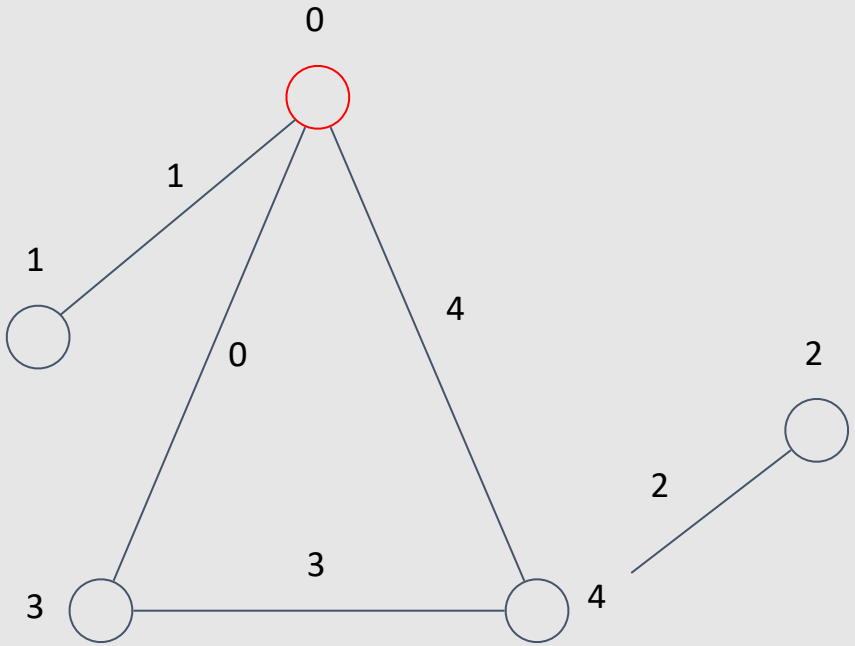
```
0 1 5 0 3
```

Complexity

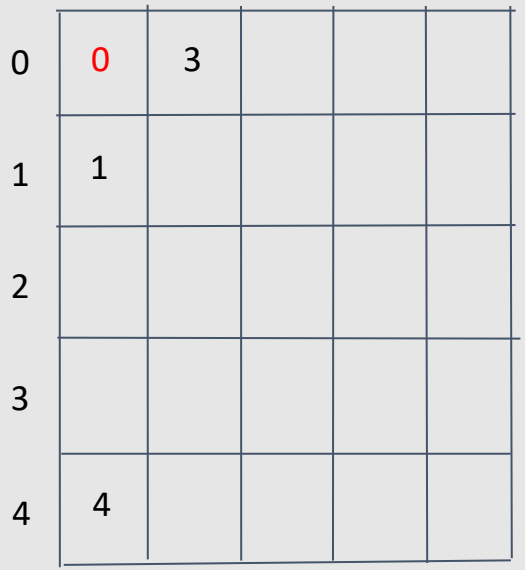
- We maintain k * n queues, and in total, we can store no more than k * n elements, because if an element is already in some queue, it can only be added to the next queue if the distance is strictly less. Also, we need m time to consider all the edges.
- The overall complexity is O(n * k) in terms of additional memory and O(n * k + m) in terms of time.

Idea

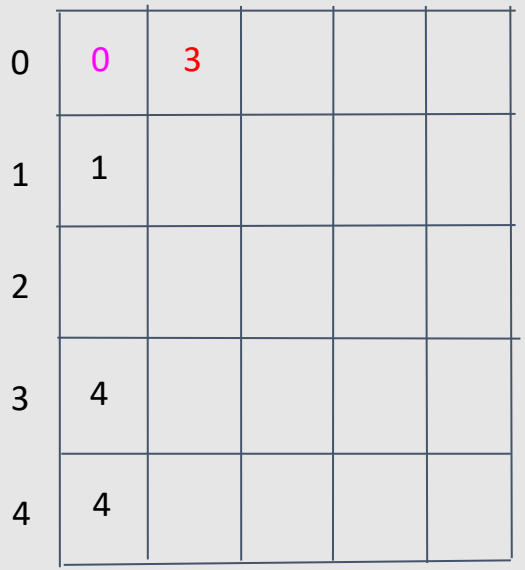
- This solution can be improved by storing only the next k + 1 queues.
- Moreover, it's easiest to do this cyclically, for example, if we are looking at an element in queue 2, and the edge weight is k, then we add the element to queue 1, because $(2 + k) \bmod (k + 1) = 1$


0
1
1
4
0
2
2
3
3
4

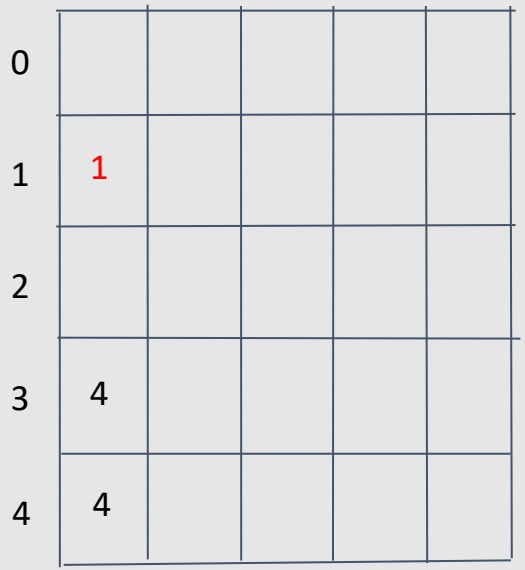
n = 5, k = 4



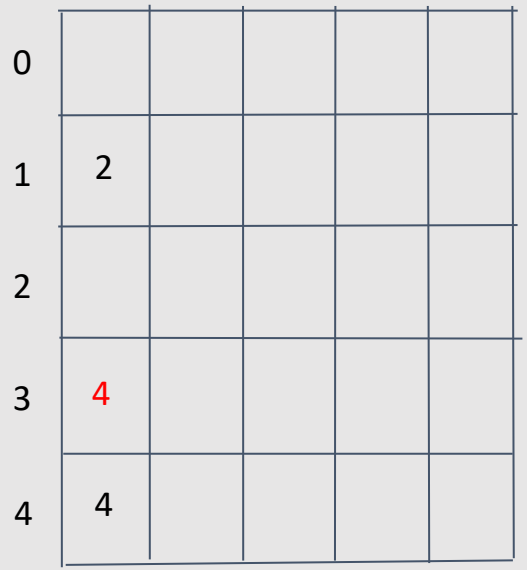
n = 5, k = 4



n = 5, k = 4

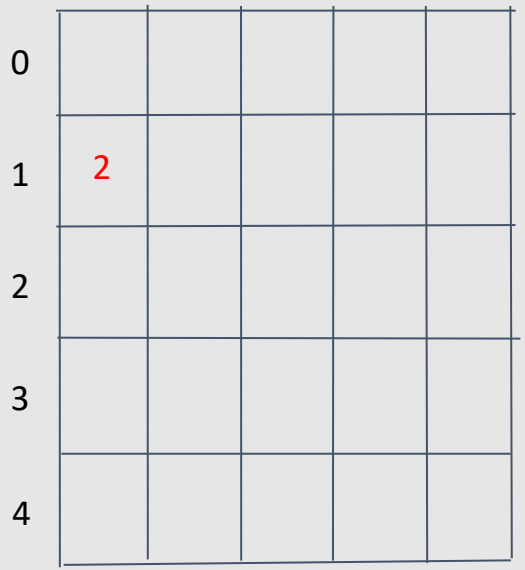


n = 5, k = 4



n = 5, k = 4

n = 5, k = 4



Complexity

- Complexity is the same

Djkstra

- Let's assume now that the weight of an edge can be any positive number.

Problem

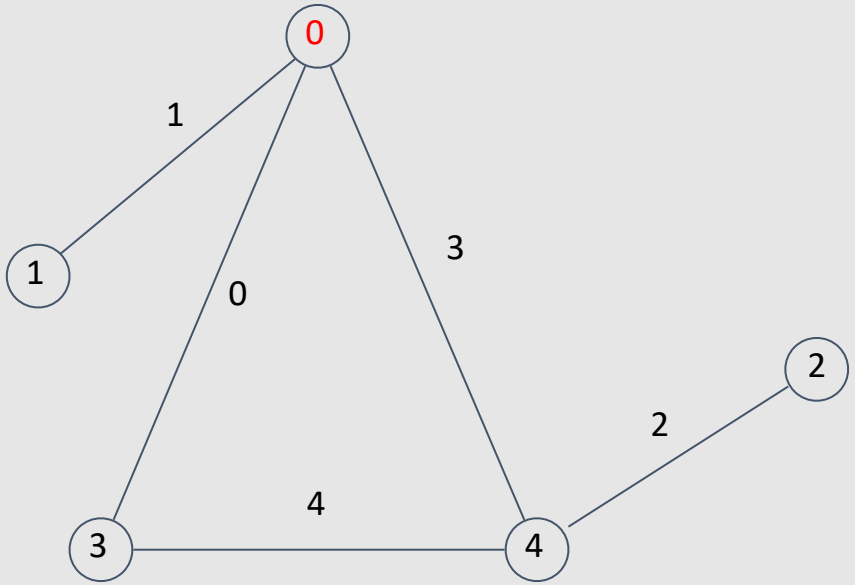
- In previous algorithms, we always inserted at the end of the queue and took from the beginning of the queue, but now we need to always take the smallest element, while we don't care about the insertion.

Idea

- We need a structure that can find the minimum and at the same time somehow manage to insert an element.
- We have already discussed such a structure in our language lectures - it is a heap or a priority queue.

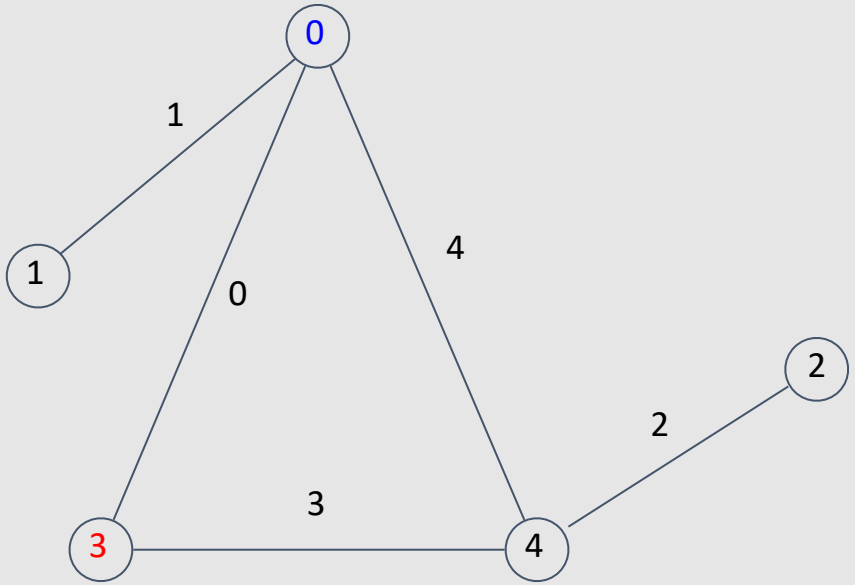
(0, 0)

0, inf, inf, inf, inf



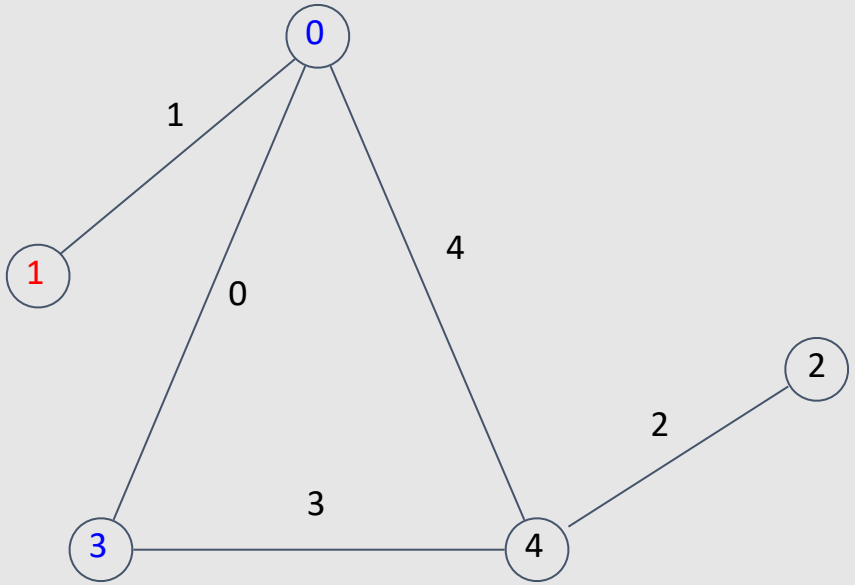
(0, 3), (1, 1), (4, 4)

0, 1, inf, 0, 4



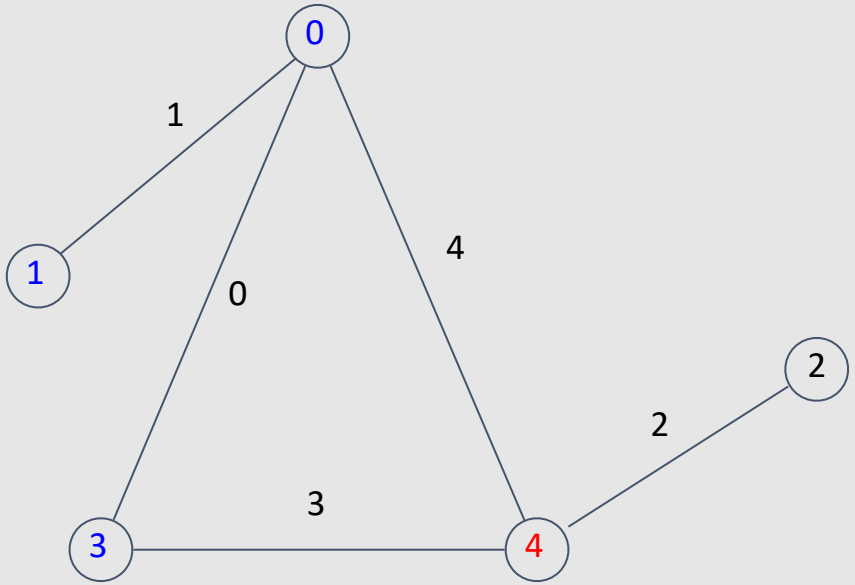
(1, 1), (3, 4)

0, 1, inf, 0, 3



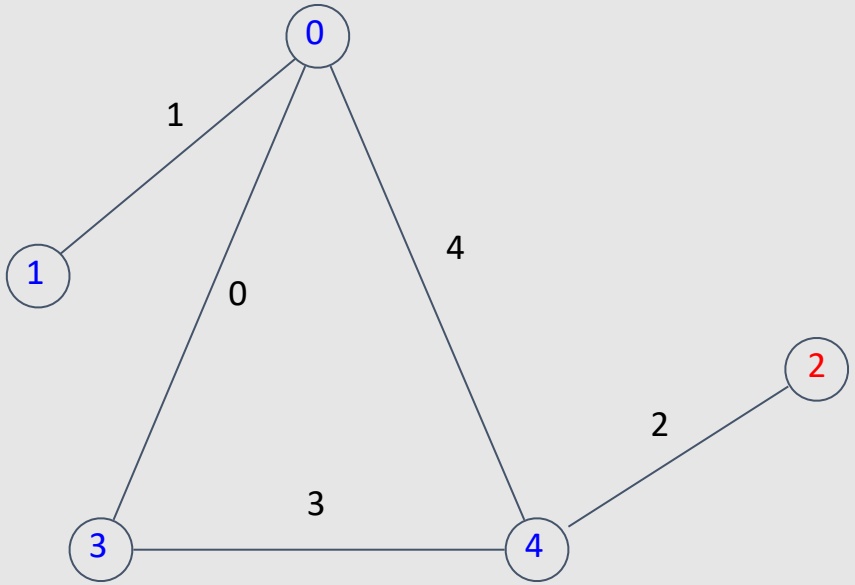
(3, 4)

0, 1, inf, 0, 3

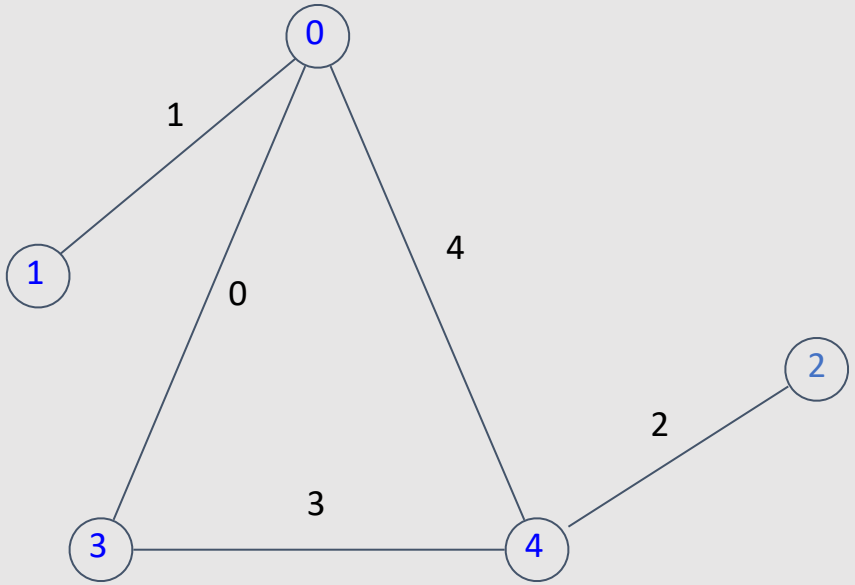


(5, 2)

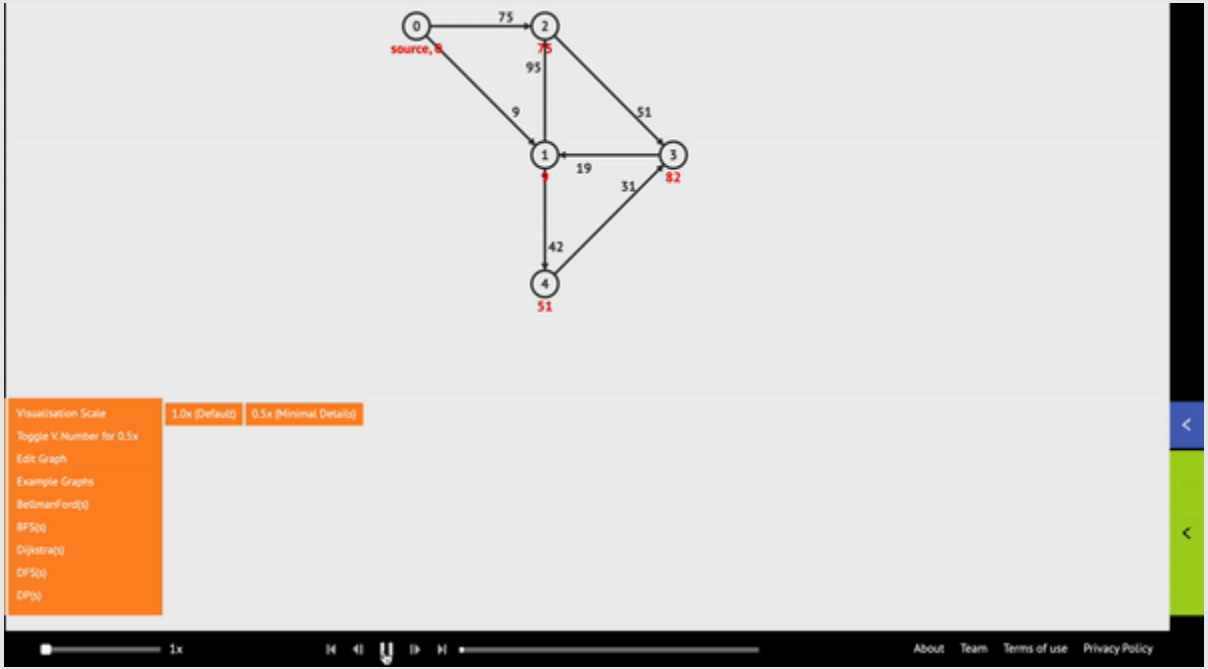
0, 1, 5, 0, 3



0, 1, 5, 0, 3



Visualization



```python
import heapq

def dijkstra(n, graph, start):
    distance = [float('inf')] * n
    distance[start] = 0
    queue = [(0, start)]

    while queue:
        current_dist, v = heapq.heappop(queue)

        if current_dist > distance[v]:
            continue

        for to, weight in graph[v]:
            if current_dist + weight < distance[to]:
                distance[to] = current_dist + weight
                heapq.heappush(queue, (distance[to], to))

    return distance

# Read data
n, m, k = map(int, input().split())
graph = [[] for _ in range(n)]

for _ in range(m):
    from_, to, weight = map(int, input().split())
    graph[from_ - 1].append((to - 1, weight))
    # If the graph is undirected, uncomment the following line
    # graph[to - 1].append((from_ - 1, weight))

# Call Dijkstra's function
distances = dijkstra(n, graph, 0) # starting vertex is 0
print(distances)
```

```
30.
31.  # Call Dijkstra's function
32.  distances = dijkstra(n, graph, 0) # starting vertex is 0
33.  print(distances)
34.
```

Success #stdin #stdout 0.03s 9844KB

stdin

```
5 5 4
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

```
[0, 1, 5, 0, 3]
```

```cpp
void dijkstra(int n, const vector<vector<pair<int, int>>>& graph, int start) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    vector<int> distance(n, INT_MAX);
    distance[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        int v = pq.top().second;
        int current_dist = pq.top().first;
        pq.pop();

        if (current_dist > distance[v]) continue;

        for (auto edge : graph[v]) {
            int to = edge.first;
            int weight = edge.second;

            if (distance[v] + weight < distance[to]) {
                distance[to] = distance[v] + weight;
                pq.push({distance[to], to});
            }
        }
    }

    // Print distances
    for (int dist : distance) {
        cout << dist << " ";
    }
    cout << endl;
}
```

```cpp
int main() {
    int n, m, k;
    cin >> n >> m >> k;
    vector<vector<pair<int, int>>> graph(n);

    for (int i = 0; i < m; ++i) {
        int from, to, weight;
        cin >> from >> to >> weight;
        --from; --to;
        graph[from].push_back({to, weight});
        // If the graph is undirected, uncomment the following line
        // graph[to].push_back({from, weight});
    }

    // Call Dijkstra's function
    dijkstra(n, graph, 0); // starting vertex is 0
    return 0;
}
```

Success #stdin #stdout 0.01s 5460KB

comments (0)

stdin

copy

```
5 5 4
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

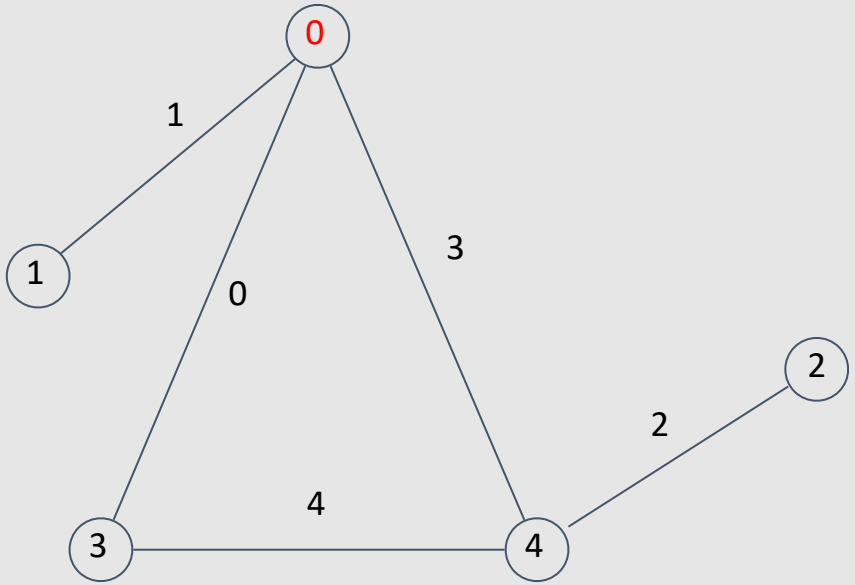
copy

```
0 1 5 0 3
```

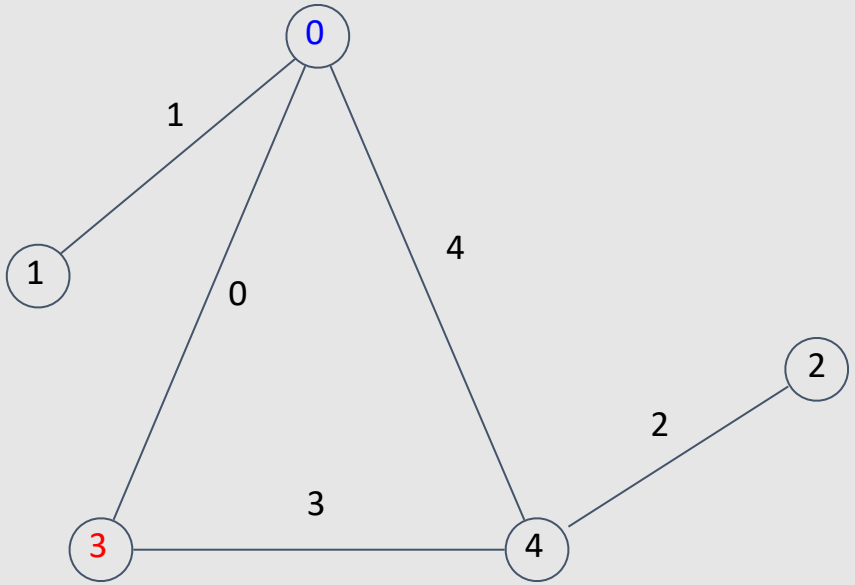
Another way

- We need to be able to find the vertex with the minimum distance and be able to update distances somehow.
- Let's now simply search for the minimum in the array, and for updates - just update the element.

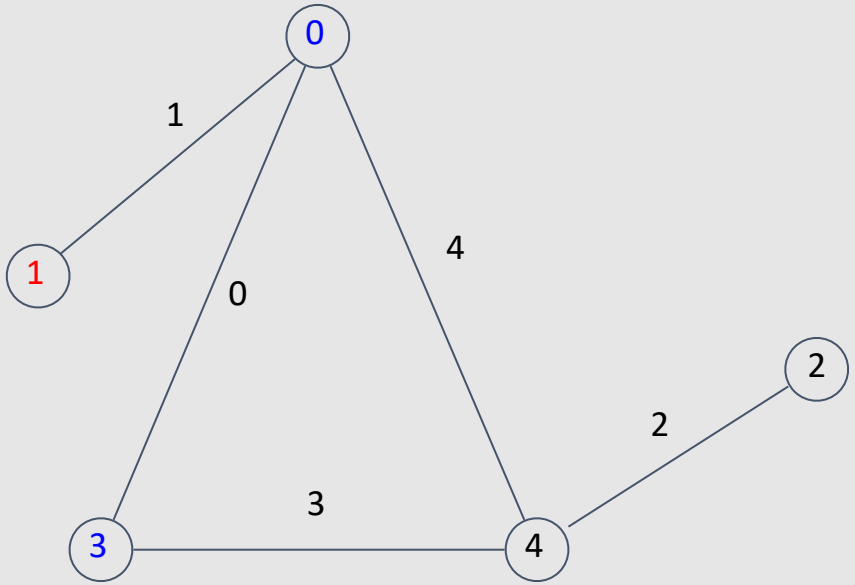
0, inf, inf, inf, inf



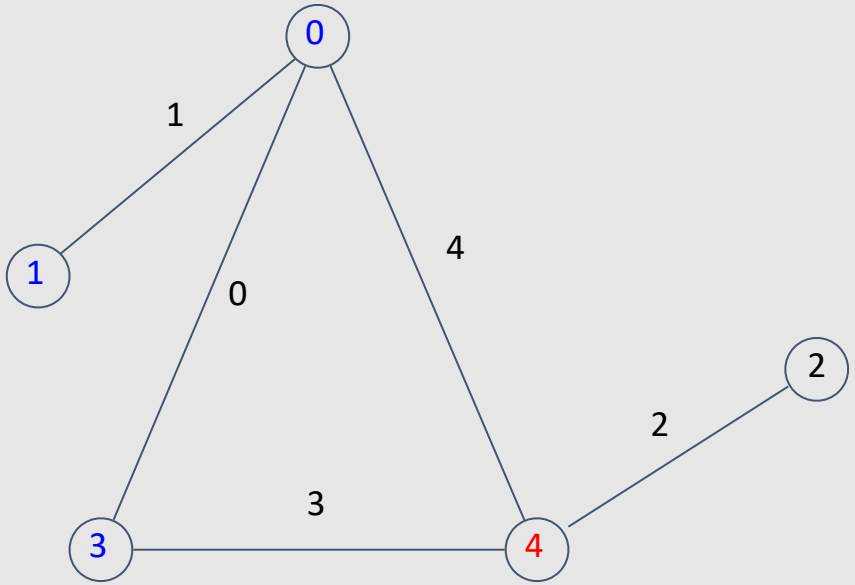
0, 1, inf, 0, 4



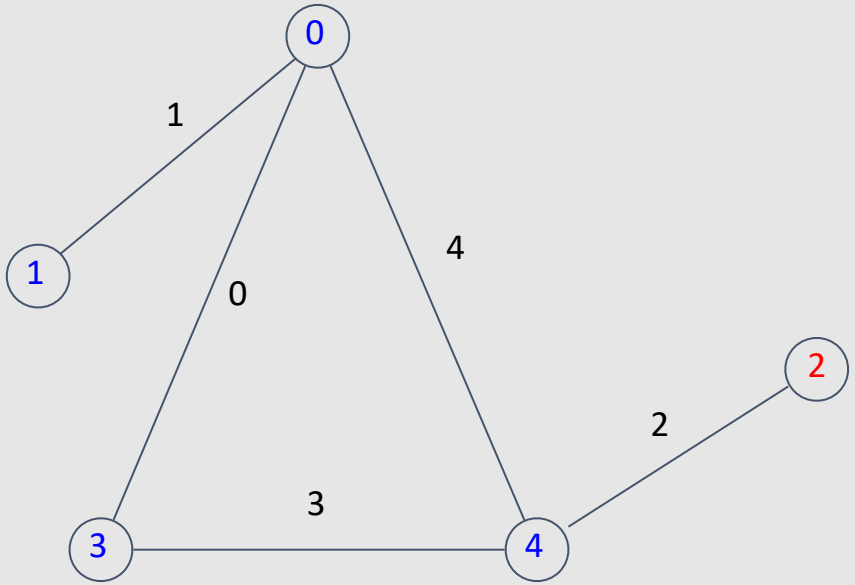
0, 1, inf, 0, 3



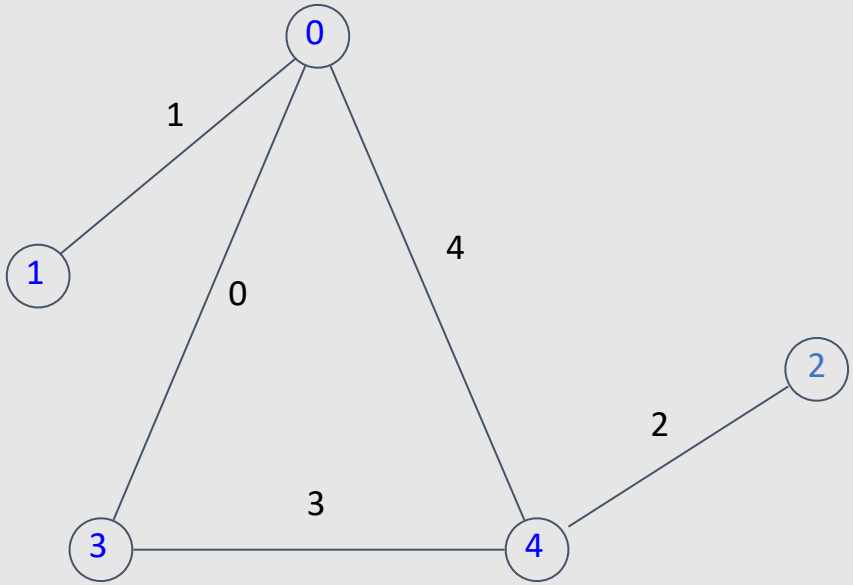
0, 1, inf, 0, 3



0, 1, 5, 0, 3



0, 1, 5, 0, 3



```python
def dijkstra(n, graph, start):
    dist = [float('inf')] * n
    dist[start] = 0
    used = [False] * n

    for _ in range(n):
        u = -1
        for v in range(n):
            if not used[v] and (u == -1 or dist[v] < dist[u]):
                u = v

        if dist[u] == float('inf'):
            break

        used[u] = True
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    return dist

n, m = map(int, input().split())
```

```
22.  n, m = map(int, input().split())
23.  graph = [[] for _ in range(n)]
24.  for _ in range(m):
25.      from_, to, weight = map(int, input().split())
26.      graph[from_ - 1].append((to - 1, weight))
27.      graph[to - 1].append((from_ - 1, weight))
28.
29.  start_vertex = 0
30.  distances = dijkstra(n, graph, start_vertex)
31.  print(distances)
32.
```

Success #stdin #stdout 0.04s 9824KB

stdin

```
5 5
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

```
[0, 1, 5, 0, 3]
```

```cpp
vector<int> dijkstra(int n, const vector<vector<pair<int, int>>>& graph, int start) {
    vector<int> dist(n, numeric_limits<int>::max());
    vector<bool> used(n, false);
    dist[start] = 0;

    for (int i = 0; i < n; ++i) {
        int u = -1;
        for (int j = 0; j < n; ++j) {
            if (!used[j] && (u == -1 || dist[j] < dist[u])) {
                u = j;
            }
        }

        if (dist[u] == numeric_limits<int>::max()) {
            break;
        }

        used[u] = true;
        for (const auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    return dist;
}
```

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<pair<int, int>>> graph(n);
    for (int i = 0; i < m; ++i) {
        int from, to, weight;
        cin >> from >> to >> weight;
        graph[from - 1].emplace_back(to - 1, weight);
        graph[to - 1].emplace_back(from - 1, weight);
    }

    vector<int> distances = dijkstra(n, graph, 0);
    for (int d : distances) {
        cout << d << ' ';
    }
    cout << endl;

    return 0;
}

```

Success #stdin #stdout 0.01s 5504KB comme

stdin

```
5 5
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

```
0 1 5 0 3
```

Floyd

- And what should we do if we want to calculate the distance not just between one vertex and the others, but immediately between all pairs of vertices?

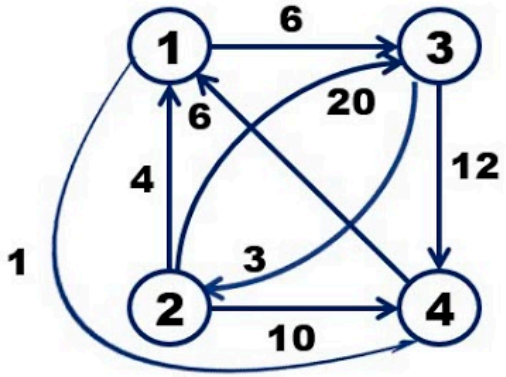
Floyd

- dp[i][u][v] - min distance from u to v using only i first vertices
- dp[0][u][u] = 0, dp[0][u][v] = w for all edges(u, v, w), dp[0][u][v] = inf otherwise
- any order
- dp[i][u][v] = min(dp[i - 1][u][v] (if we dont take i-th vertex), dp[i - 1][u][i] + dp[i - 1][i][v] (if we take ith vertex)

FLOYD'S ALGORITHM

0	1	2	3	4
1	0	∞	6	1
2	4	0	20	10
3	∞	3	0	12
4	6	∞	∞	0

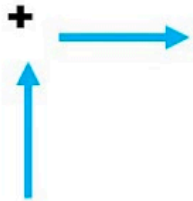
k=2	1	2	3	4
1	0	∞	6	1
2	4	0	10	5
3	7	3	0	8
4	6	∞	12	0



k=1	1	2	3	4
1	0	∞	6	1
2	4	0	10	5
3	∞	3	0	12
4	6	∞	12	0

k=3	1	2	3	4
1				

IDEA !!!



Idea

- In fact, we can observe that we don't actually need to store n^3 memory, as we don't need to keep track of i (the stage number), and can simply iterate.
- Note that in the worst case, a path without cycles will consist of n edges, and therefore, if we perform two operations in one stage of the loop, it won’t break anything.

Visualization

```python
INF = float('inf')

def floyd_warshall(graph):
    n = len(graph)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])

n, m = map(int, input().split())
graph = [[INF] * n for _ in range(n)]

for _ in range(m):
    u, v, w = map(int, input().split())
    graph[u - 1][v - 1] = w  # Adjusting indices to 0-based
    graph[v - 1][u - 1] = w  # For undirected graph

for i in range(n):
    graph[i][i] = 0  # Distance to self is 0

floyd_warshall(graph)

# Print shortest distances
```

```
22.
23.  # Print shortest distances
24.  for row in graph:
25.      print(' '.join(['INF' if x == INF else str(x) for x in row]))
26.
```

Success #stdin #stdout 0.04s 9748KB

comments (0)

stdin

copy

```
5 5
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

copy

```
0 1 5 0 3
1 0 6 1 4
5 6 0 5 2
0 1 5 0 3
3 4 2 3 0
```

```cpp
const int INF = 1e9; // Define infinity value (adjust if necessary)

void floydWarshall(vector<vector<int>>& graph, int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (graph[i][k] < INF && graph[k][j] < INF)
                    graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
            }
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<vector<int>> graph(n, vector<int>(n, INF));

    for (int i = 0; i < n; i++) {
        graph[i][i] = 0; // Distance to self is 0
    }

    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u - 1][v - 1] = w; // Adjusting indices to 0-based
        graph[v - 1][u - 1] = w; // For undirected graph
    }

    floydWarshall(graph, n);

    // Print shortest distances
```

```
    // Print shortest distances
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] == INF)
                cout << "INF ";
            else
                cout << graph[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}

```

Success #stdin #stdout 0.01s 5516KB

comments (0)

stdin

copy

```
5 5
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

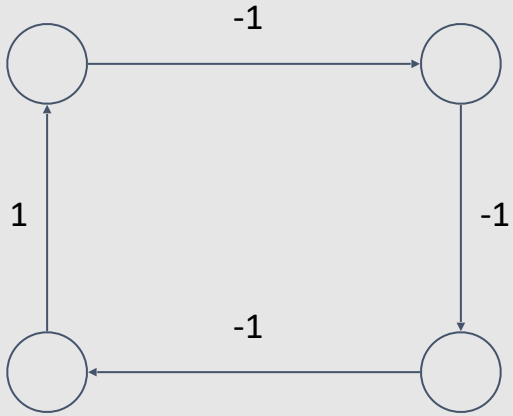
stdout

copy

```
0 1 5 0 3
1 0 6 1 4
5 6 0 5 2
0 1 5 0 3
3 4 2 3 0
```

Negative edges

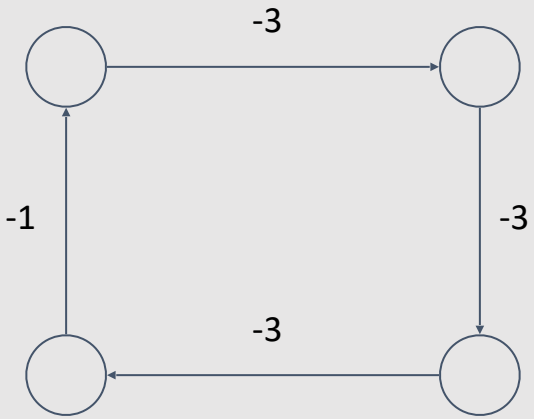
- In fact, edges with negative weights themselves do not break the Floyd algorithm.
- Only cycles with negative weight break something.
- But why do they break something, and how can this be fixed?


-1
1
-1
-1

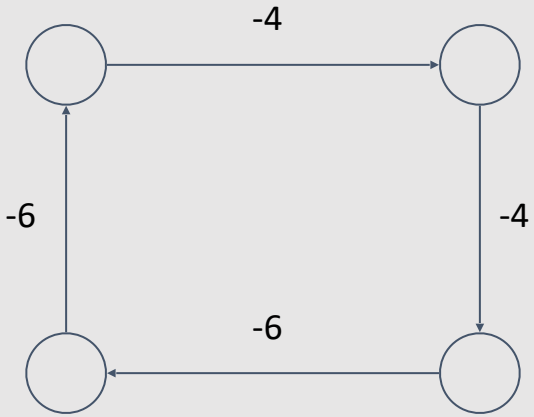
Negative cycles

- In fact, by using the Floyd algorithm, we can check if there is a cycle of negative weight and even find it.

after n steps



one extra step



Negative cycles

- Checking for the existence of a negative weight cycle is quite simple.
- Suppose we have completed n iterations of dynamic calculation.
- Let's do one more iteration; if after this iteration at least one element has changed, it means there is a negative weight cycle in the graph.

Why

- Let's prove that if something has changed, then a cycle exists.
- If there are no negative weight cycles, then the maximum length (number of edges) of any path is at most n - 1.
- Therefore, no distance should change after one more operation.
- We have found a contradiction

Why

- Let's prove that if there is a cycle, then something has changed.
- If there is a cycle with negative weight, it means that at some point at least one element will change, this can be written as F(...F(D)) = D’, where D != D’.
- But since we applied the transition F to the distance matrix D and nothing changed, then after another application nothing will change either, etc., because F(...F(D)) = F(..(D)) = F(D).
- And here we have found a contradiction.

Ford-Belman

- Let's again look for the distance from one vertex to all others, but this time we will have negative edges and, accordingly, negative cycles.

Ford-Belman

- dp[i][j] = distance from s to j, using only i edges.
- dp[0][s] = 0, dp[0][u, u != s] = inf
- order - first by amount of edges, then by vertices.
- dp[i][u] = min(dp[i - 1][v], dp[i][u]), for all (v, u)
- dp[n][v] = distance from j to v

Again

- We can again remove number of stages from dp.

Cycle

- We can check for the presence of a negative weight cycle in the same way as in the Floyd algorithm. We simply add an extra phase and verify that nothing changes.

```cpp
    int n, m;
    cin >> n >> m;

    vector<vector<int>> edges;

    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        edges.push_back({u - 1, v - 1, w});  // Adjusting indices to 0-based
    }

    // Bellman-Ford algorithm
    vector<int> distance(n, INF);
    distance[0] = 0; // Starting from vertex 0

    for (int i = 0; i < n - 1; i++) {
        for (const auto &edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int w = edge[2];

            if (distance[u] != INF && distance[u] + w < distance[v]) {
                distance[v] = distance[u] + w;
            }
        }
    }

    // Check for negative weight cycles
    for (const auto &edge : edges) {
        int u = edge[0];
        int v = edge[1];
        int w = edge[2];

        if (distance[u] != INF && distance[u] + w < distance[v]) {
            cout << "Graph contains a negative weight cycle" << endl;
            return 0;
        }
    }
```

```
    // Print distances
    for (int i = 0; i < n; i++) {
        cout << "Distance from 0 to " << i << " is ";
        if (distance[i] == INF) {
            cout << "INF";
        } else {
            cout << distance[i];
        }
        cout << endl;
    }

    return 0;
}
```

Success #stdin #stdout 0.01s 5380KB comments

stdin cc

```
5 5
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout cc

```
Distance from 0 to 0 is 0
Distance from 0 to 1 is 1
Distance from 0 to 2 is 5
Distance from 0 to 3 is 0
Distance from 0 to 4 is 3
```

```python
INF = float('inf')

n, m = map(int, input().split())
edges = []

for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u - 1, v - 1, w))  # Adjusting indices to 0-based

# Bellman-Ford algorithm
distance = [INF] * n
distance[0] = 0  # Starting from vertex 0

for _ in range(n - 1):
    for u, v, w in edges:
        if distance[u] != INF and distance[u] + w < distance[v]:
            distance[v] = distance[u] + w

# Check for negative weight cycles
for u, v, w in edges:
    if distance[u] != INF and distance[u] + w < distance[v]:
        print("Graph contains a negative weight cycle")
        break
else:
    # Print distances
    for i in range(n):
        print(f"Distance from 0 to {i} is {distance[i] if distance[i] != INF else 'INF'}")

```

Success #stdin #stdout 0.03s 9768KB

comments (0)

stdin

copy

```
5 5
1 2 1
1 4 0
1 5 4
4 5 3
5 3 2
```

stdout

copy

```
Distance from 0 to 0 is 0
Distance from 0 to 1 is 1
Distance from 0 to 2 is 5
Distance from 0 to 3 is 0
Distance from 0 to 4 is 3
```

Tasks

- Given a number X and a set of digits D, the task is to append the minimum number of digits from D to X, so that the resulting number is divisible by k. Additionally, the resulting number should be as small as possible.

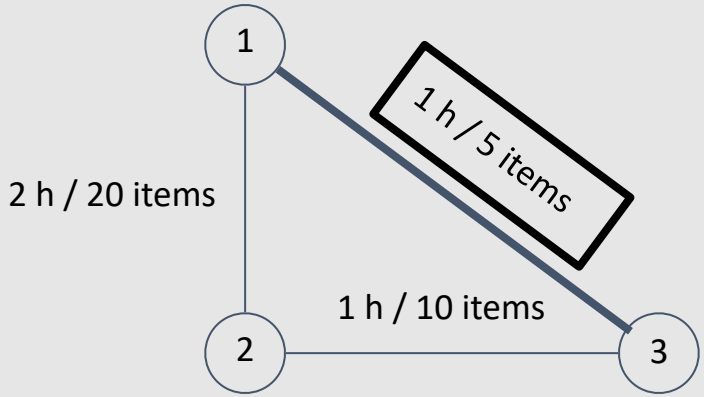
Example

1) Given the number 102 and the set of digits {1, 0, 3}, we need to find the smallest number, formed by appending the least number of these digits to 102, that is divisible by 101. Answer = 10201

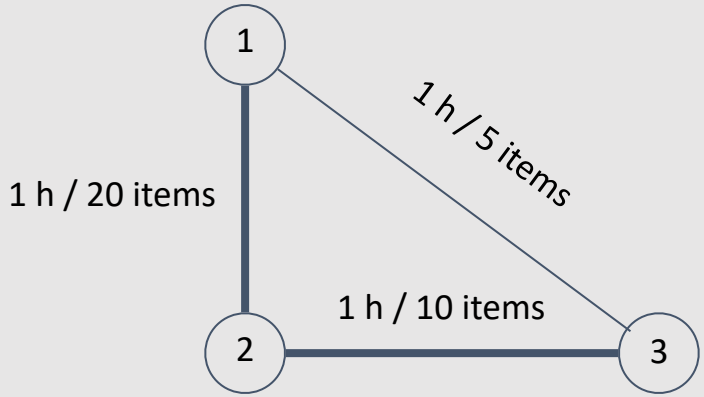
Tasks

- Determine the maximum number of mugs that can be loaded onto a truck in one trip to ensure they are delivered to the camp within X hours, considering weight restrictions on certain roads. The map can be represented as n cities and m roads between them.

n = 3, m = 3, X = 2



n = 3, m = 3, X = 2



Tasks

- Buses run between some cities. Since the passenger flow here is not very large, buses only run a few times a day.
- The goal is to get from city 0 to city n - 1 as quickly as possible (at time 0 we are in city 0).

Example

A flight from location 1 to 2 starts at time 0 and takes 5 seconds.

A flight from location 1 to 2 starts at time 1 and takes 3 seconds.

A flight from location 2 to 3 starts at time 3 and takes 5 seconds.

A flight from location 1 to 3 starts at time 1 and takes 10 seconds.

Task

Treasure hunter Vasya has found a map of an ancient dungeon. The dungeon is a maze of size N×M. Each cell of the maze is either empty and passable, or contains a wall. You can only move to a cell that shares a wall with the current cell (thus, each cell can have up to 4 adjacent cells).

In one of the cells lies the treasure that Vasya wants to find. There are K entrances in the maze from which Vasya can start his journey.

It is necessary to determine from which entrance Vasya should start in order to reach the treasure with the shortest possible distance traveled. If there are several such entrances, the entrance with the lowest number should be output.

Example

map:

00000

00000

10*00

01111

00000

Entrances - {(1, 1), (1, 5), (4, 1), (5, 5)}

All codes

edges(c++) - https://ideone.com/aBNXXJ

edges(python) - https://ideone.com/HhEJ0p

matrix(c++) - https://ideone.com/UdX0uQ

matrix(python) - https://ideone.com/aJn659

vector vector(c++) - https://ideone.com/ykU3d8

vector vector(python) - https://ideone.com/QzXRF9

dfs - tree(c++) - https://ideone.com/C7XSkH

dfs-tree(python) - https://ideone.com/RTNCRH

dfs(pyhton) - https://ideone.com/ytSMr8

dfs - graph(c++) - https://ideone.com/JRRG6b

depth(c++) - https://ideone.com/JKe2nr

depth(python) - https://ideone.com/JYnajS

dfs - without rec (c++) - https://ideone.com/8BO4S7

dfs - without rec(python) - https://ideone.com/UKIEsx

bfs(c++) - https://ideone.com/3vdOVj

bfs(python) - https://ideone.com/tOFCkm

All codes

distance(c++) - https://ideone.com/BXQ5Hq
distance(python) - https://ideone.com/rXQ2IM
euler tour(c++) - https://ideone.com/Wvg9Cx
euler tour(python) - https://ideone.com/Jl08mM
euler tour - 2(c++) - https://ideone.com/AxRLDa
euler tour - 2(python) - https://ideone.com/3frgTM
components(c++) - https://ideone.com/hcD1Na
components(python) - https://ideone.com/cpCOpZ
cycle(c++) - https://ideone.com/GOMrPO
cycle(python) - https://ideone.com/utO8VS
topsort(c++) - https://ideone.com/u5hsOk
topsort(python) - https://ideone.com/qYcbfF
cut points(c++) - https://ideone.com/Y5sL9r
bridges(c++) - https://ideone.com/veoWDG
bridge(python) - https://ideone.com/FuyXEJ
condensation(python) - https://ideone.com/Njq8Ys
condensation(C++) - https://ideone.com/VQVFpf
0-1 bfs(c++) - https://ideone.com/AZNd43

All codes

0-k bfs(python) - https://ideone.com/bfJYtn

0-k bfs(c++) - https://ideone.com/JgH6tb

djkstra(python) - https://ideone.com/ATtW40

djkstra(c++) - https://ideone.com/YqdNUI

djkstra n2(c++) - https://ideone.com/LKfWnQ

djksta n2(python) - https://ideone.com/7PtspL

floyd(python) - https://ideone.com/EnpxtB

floyd(c++) - https://ideone.com/A3rcCK

ford(c++) - https://ideone.com/PXOYdw

ford(python) - https://ideone.com/lJjeuS