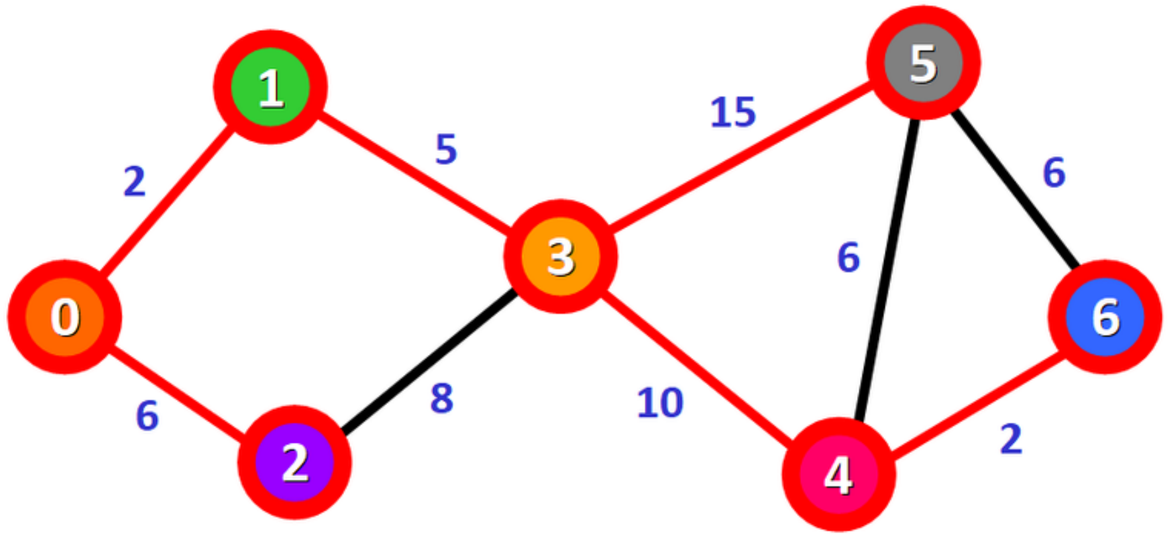
Table of Contents

What is Dijkstra's

Dijkstra's algorithm is also known as the **shortest path algorithm**. It is an algorithm used to find the shortest path between nodes of the graph. The algorithm creates the tree of the shortest paths from the starting source vertex from all other points in the graph.



starting node : 0

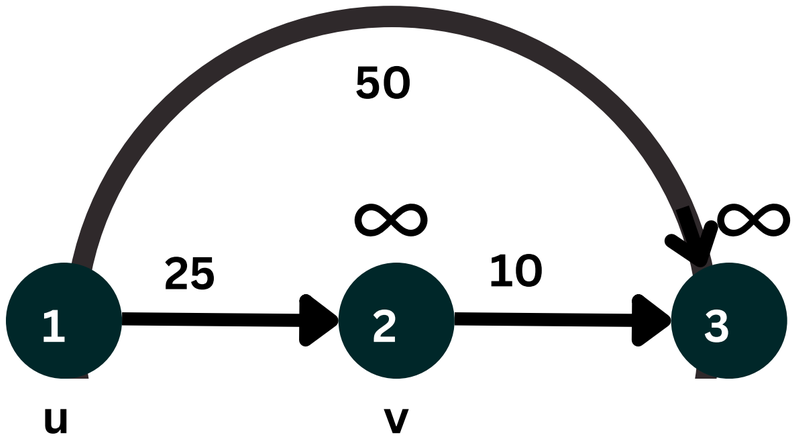
destination : 6

0 -> 1 -> 3 -> 4 - > 6

Applications of Dijkstra's Algorithm

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

Example of Dijkstra's Algorithm



1. **Greedy Algorithm**
2. **Relaxation**

if d(u) + c(u,v) < d(v)
d(v) = d(u) + c(u,v)

DFS vs BFS vs Dijkstra's

Depth-first search :
Time Complexity: O(E+V)
Data Structure used : Stack
Graph type:

- Undirected Graph
- Directed Acyclic Graphs(DAG) without weigth

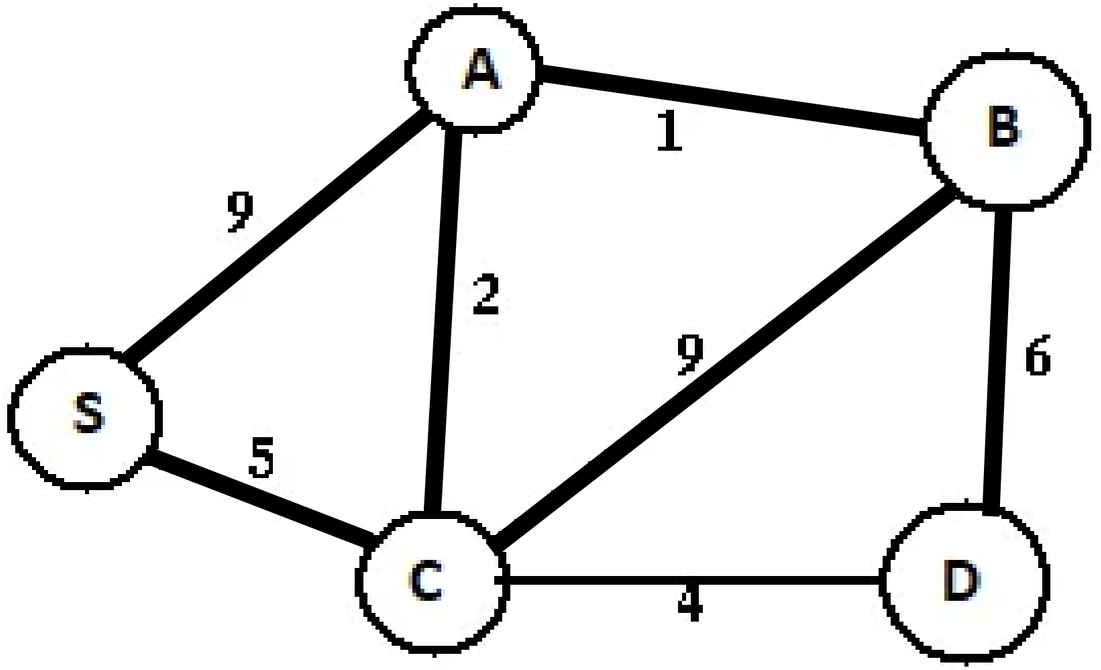
Breadth-first search
Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It uses a queue.
Time Complexity: O(E+V)
Data Structure used : Queue
Graph type:

- Undirected Graph
- Directed Acyclic Graphs(DAG) without weigth

Dijkstra's algorithm
Time Complexity: O(E+V log V)
Data Structure used : Priority Queue
Graph type:

- non-negative weighted DAG

Example of Dijkstra's algorithm

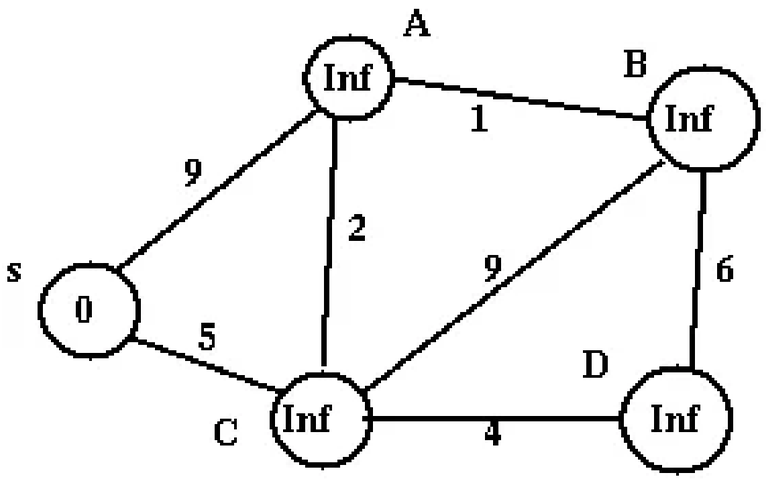


Example of Dijkstra's algorithm

Step - 1 :

Initialize the distance array (dist) using the following steps of algorithm –

Set dist[s]=0, S=ϕ // u is the source vertex and S is a 1-D array having all the For all nodes v except s, set dist[v]= ∞

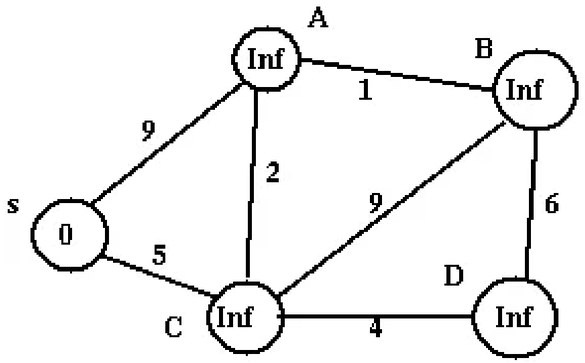


Set of visited vertices (S)	S	A	B	C	D
	0	∞	∞	∞	∞

Example of Dijkstra's algorithm

Step 1: Set dist[s]=0, S=ϕ // s is the source vertex and S is a 1-D array having all the visited vertices
Step 2: For all nodes v except s, set dist[v]= ∞
Step 3: find q not in S such that dist[q] is minimum // vertex q should not be visited
Step 4: add q to S // add vertex q to S since it has now been visited
Step 5: update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited dist[r]=min(dist[r], dist[q]+cost[q][r]) //Greedy and Dynamic approach
Step 6: Repeat Steps 3 to 5 until all the nodes are in S // repeat till all the vertices have been visited
Step 7: Print array dist having shortest path from the source vertex u to all other vertices
Step 8: Exit

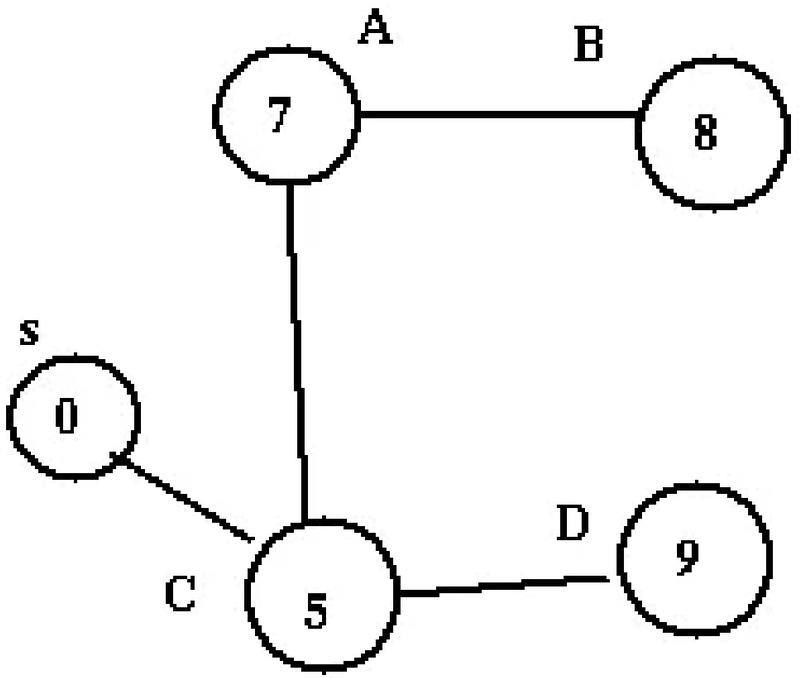
Example of Dijkstra's algorithm



Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9
[s, C, A]	0	7	8	5	9
[s, C, A, B]	0	7	8	5	9
[s, C, A, B, D]	0	7	8	5	9

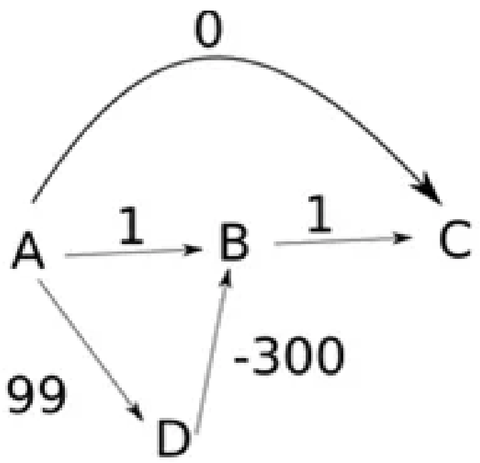
Example of Dijkstra's algorithm

Resultant graph



Disadvantages of Dijkstra's Algorithm-

Dijkstra's Algorithm cannot obtain correct shortest path(s)with weighted graphs having negative edges.



Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99
[A, C]	0	1	0	99
[A, C, B]	0	1	0	99
[A, C, B, D]	0	1	0	99

we got A->C->B->D not correct

we can have another path from A to C, A->D->B->C having total cost= -200 which is smaller than 0.

Time Complexity Analysis-

Case1- When graph G is represented using an adjacency matrix the complexity of Dijkstra's algorithm is O(n2). Please note that n here refers to total number of vertices in the given graph

Case 2- When graph G is represented using an adjacency list - The time complexity, in this scenario reduces to O(|E| + |V| log |V|) where |E|represents number of edges and |V| represents number of vertices in the graph. Priority Queue is used here.


Thank
you

Table of Contents

What is Bellman Ford

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.
It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

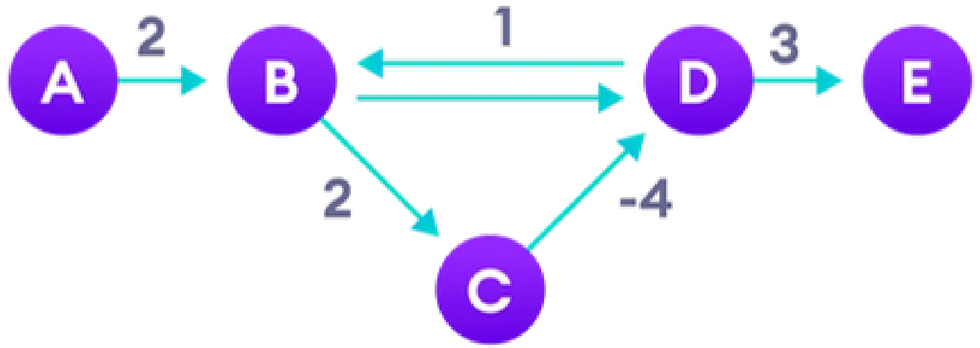
The Bellman-Ford algorithm uses the **bottom-up approach.**

Bellman Ford in real life

- Identifying negative weight cycles
- In a chemical reaction, calculate the smallest possible heat gain/loss.
- Identifying the most efficient currency conversion method.
- Using negative weights, find the shortest path in a graph.

Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.
Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.



How Bellman Ford's algorithm works
Assumptions

- Graph does not contain a negative circle
  - Shortest path exist, but Bellman-Ford algorithm can’t handle this case
  - However, no shortest walk exists

- Graph is directed
  - If undirected → replace each undirected edge by two directed edges

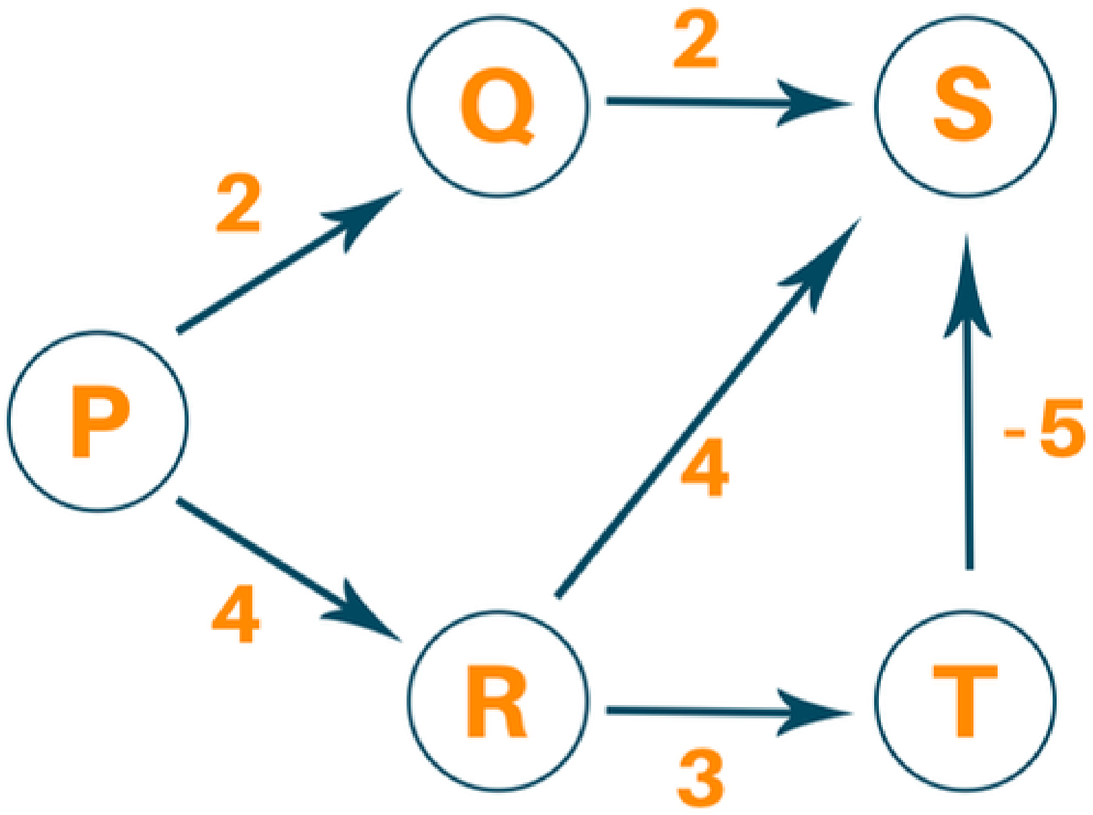
How Bellman Ford's algorithm works

- Go on relaxing all the edges n-1 times
  where n is the number of node
- we have to check if there exists any negative weighted cycle

Relaxing means:

1. If (d(u) + c(u , v) < d(v))
   d(v) = d(u) + c(u , v)

How Bellman Ford's algorithm works

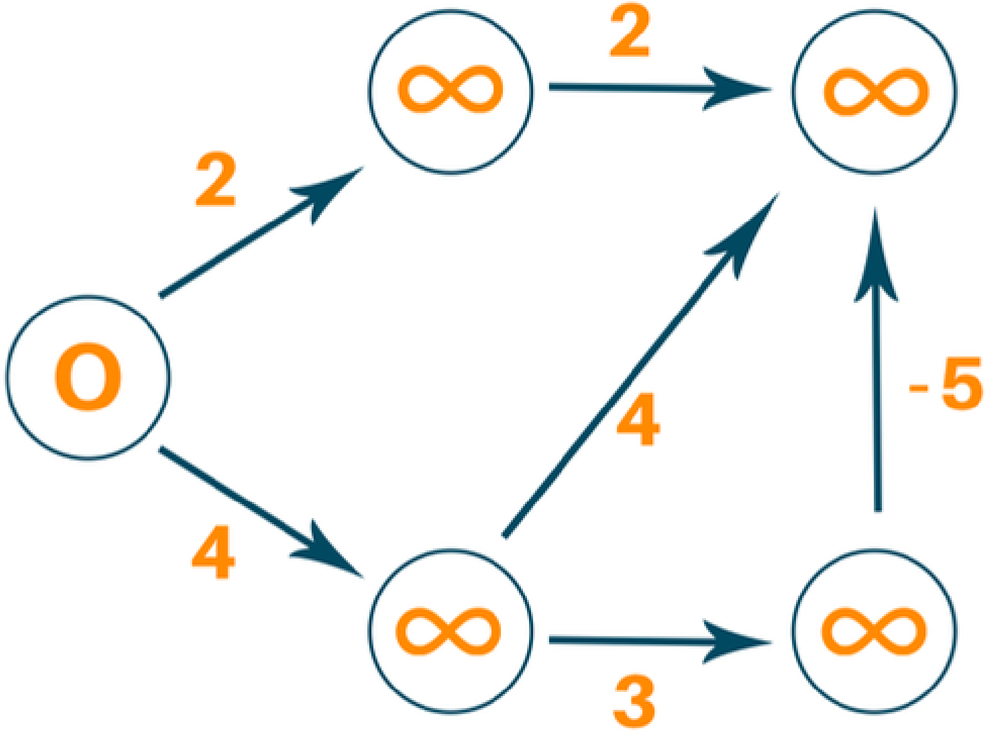


total node v = 5

total iteration will be v - 1 = 4

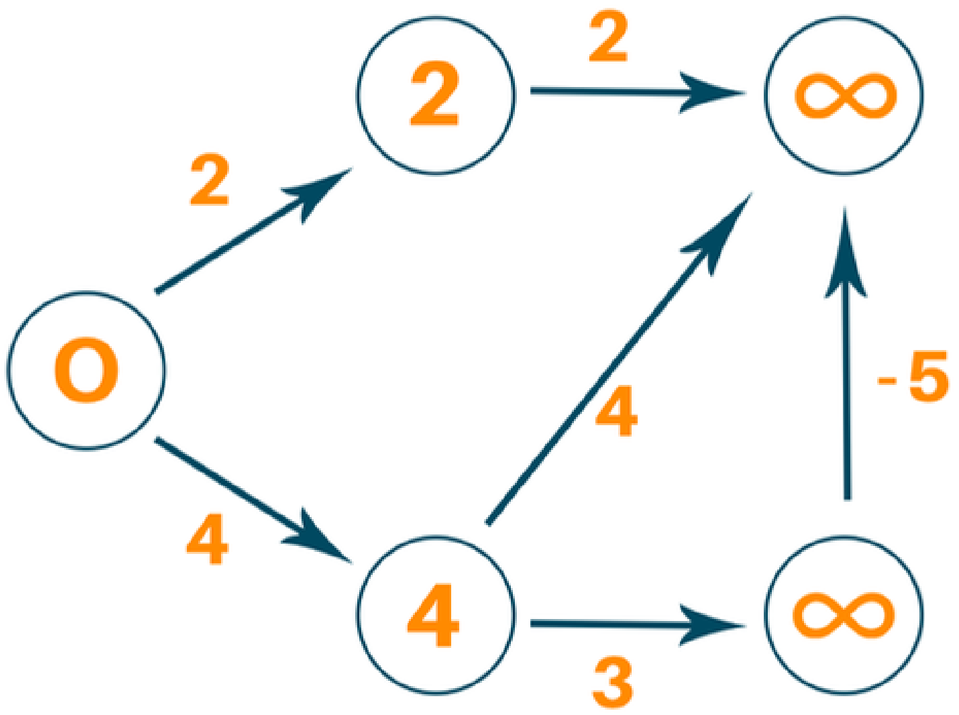
How Bellman Ford's algorithm works

0th Iteration



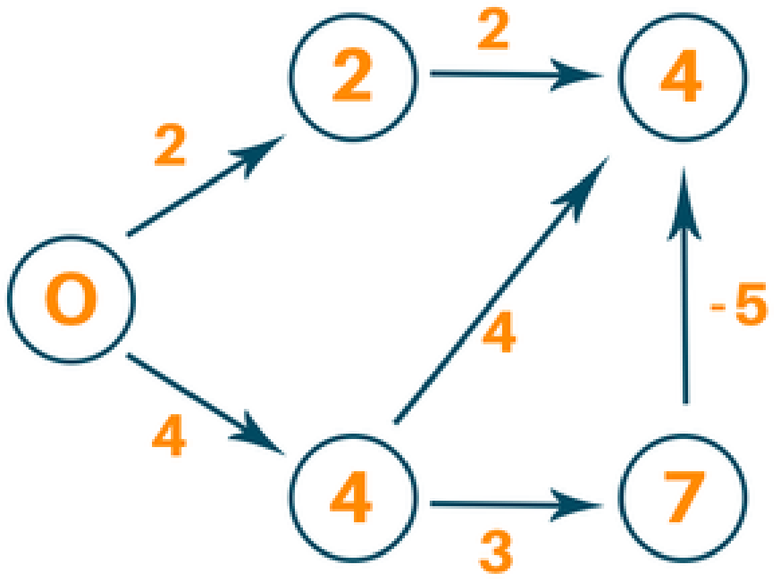
How Bellman Ford's algorithm works

1st Iteration

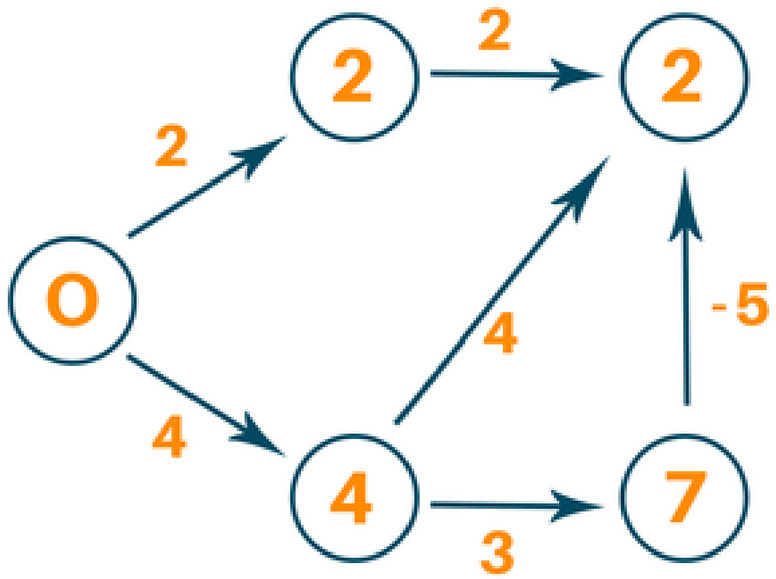


How Bellman Ford's algorithm works

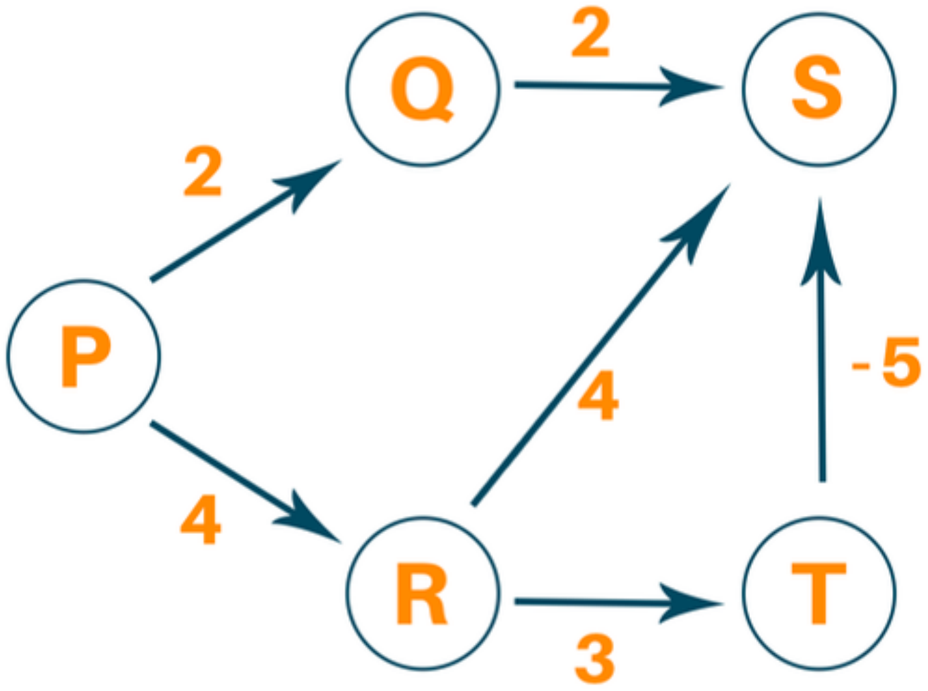
2nd Iteration



3rd Iteration

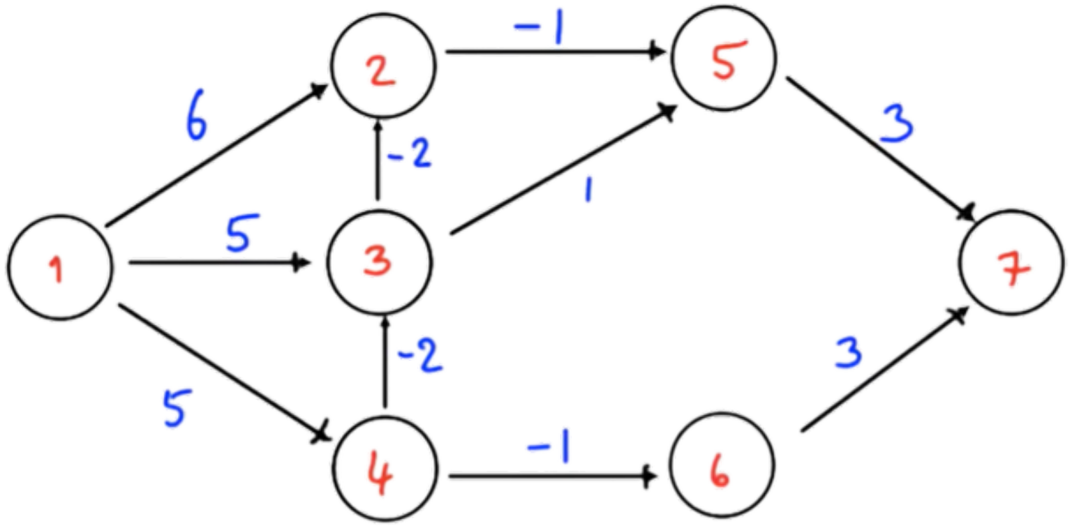


How Bellman Ford's algorithm works



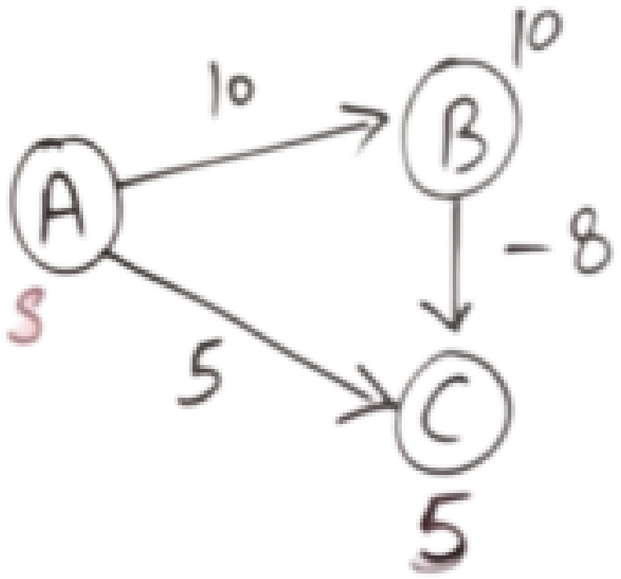
	Q	R	S	T
0	∞	∞	∞	∞
0	2	4	∞	∞
0	2	4	4	7
0	2	4	2	7
0	2	4	2	7

How Bellman Ford's algorithm works



	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

negative weights in Dijstra Algorithm



A	B	C
0	∞	∞
	10	5
AC	10	5
ACB		

negative weights in Dijstra Algorithm

```
BellmanFord ( G, V, E, S)
Step 1
 for each vertex v ∈ G do
     dist [v] = ∞
     dist [Source] = 0
Step 2
   for i = 1 to |V| - 1
      for each edge (u,v) ∈ G
          Relax( u,v,w)
Step 3
    for each edge (u,v) ∈ G
       if (dist[u] + w(u,v) < dist[v])
         return " Graph Contain Negative weight cycle
return distance
```

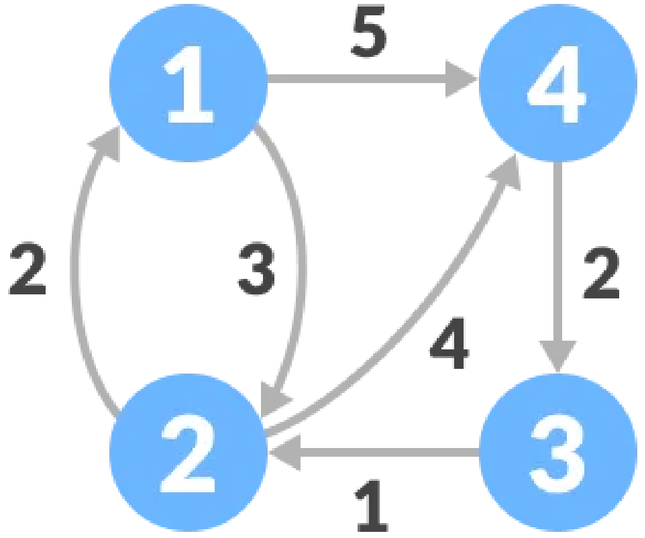
O(V)

O(V*E)

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

How Floyd-Warshall Algorithm works



How Floyd-Warshall Algorithm works

Create a matrix A0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

How Floyd-Warshall Algorithm works

2. Now, create a matrix `A¹` using matrix `A⁰`. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

   Let `k` be the intermediate vertex in the shortest path from source to destination. In this step, `k` is the first vertex. `A[i][j]` is filled with `(A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j])`.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

How Floyd-Warshall Algorithm works

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

How Floyd-Warshall Algorithm works

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

How Floyd-Warshall Algorithm works

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Floyd-Warshall Algorithm complexity

Floyd Warshall Algorithm Complexity

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Space Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

2D Grid Traversal

Grid Traversal

Step 1 :

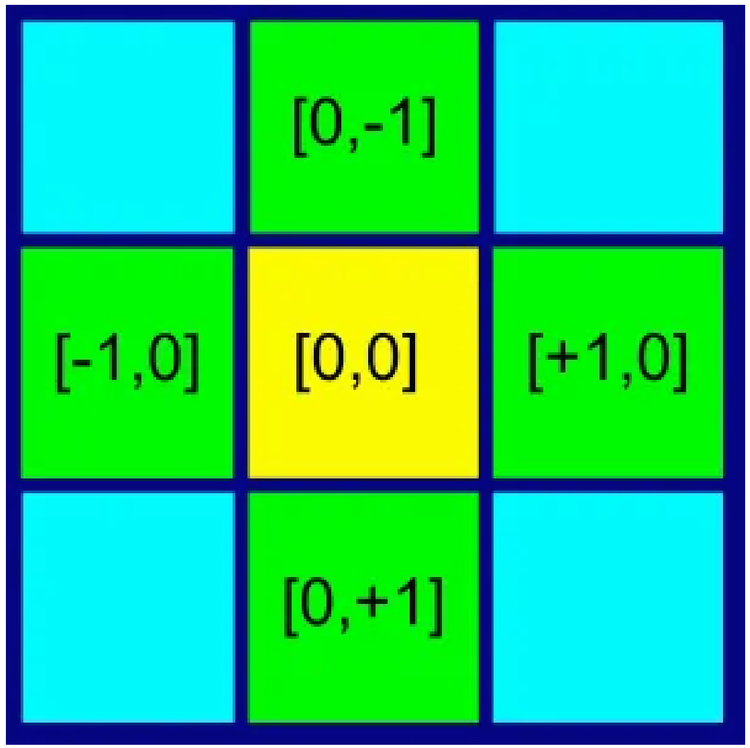
Variable Legend

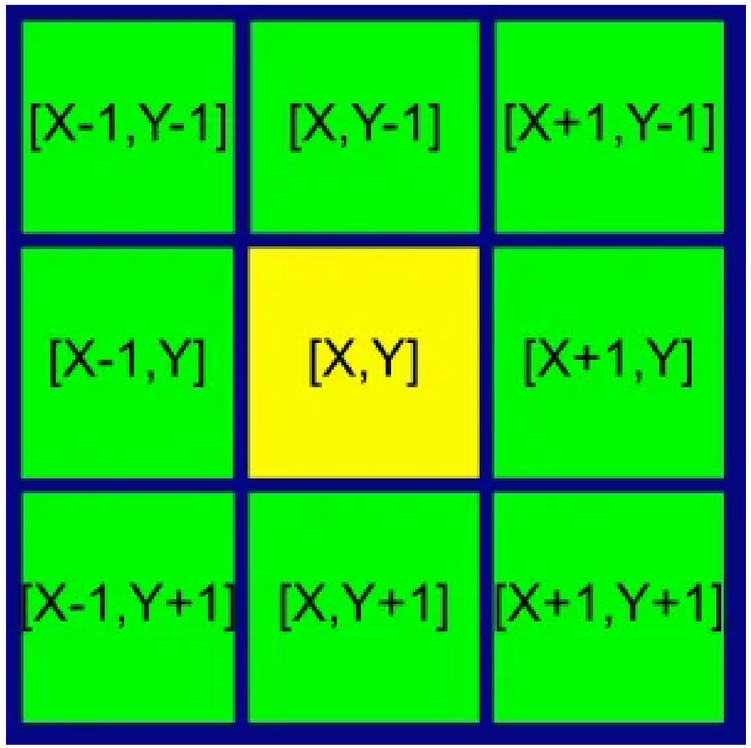
```
int n, m;

vector<vector<pair<int, int>>> path;
vector<vector<bool>> vis;
int sx, sy, ex, ey;
vector<pair<int, int>> moves = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

Grid Traversal

Step 1 :

Variable Legend, moves





Grid Traversal

Step 2 :

Room Boundary Detection, Wall Detection, Visit Checks

```
bool isValid(int x, int y)
{
    if (x < 0 or x >= n or y < 0 or y >= m)
        return false;
    if (vis[x][y])
        return false;
    return true;
}
```

Grid Traversal

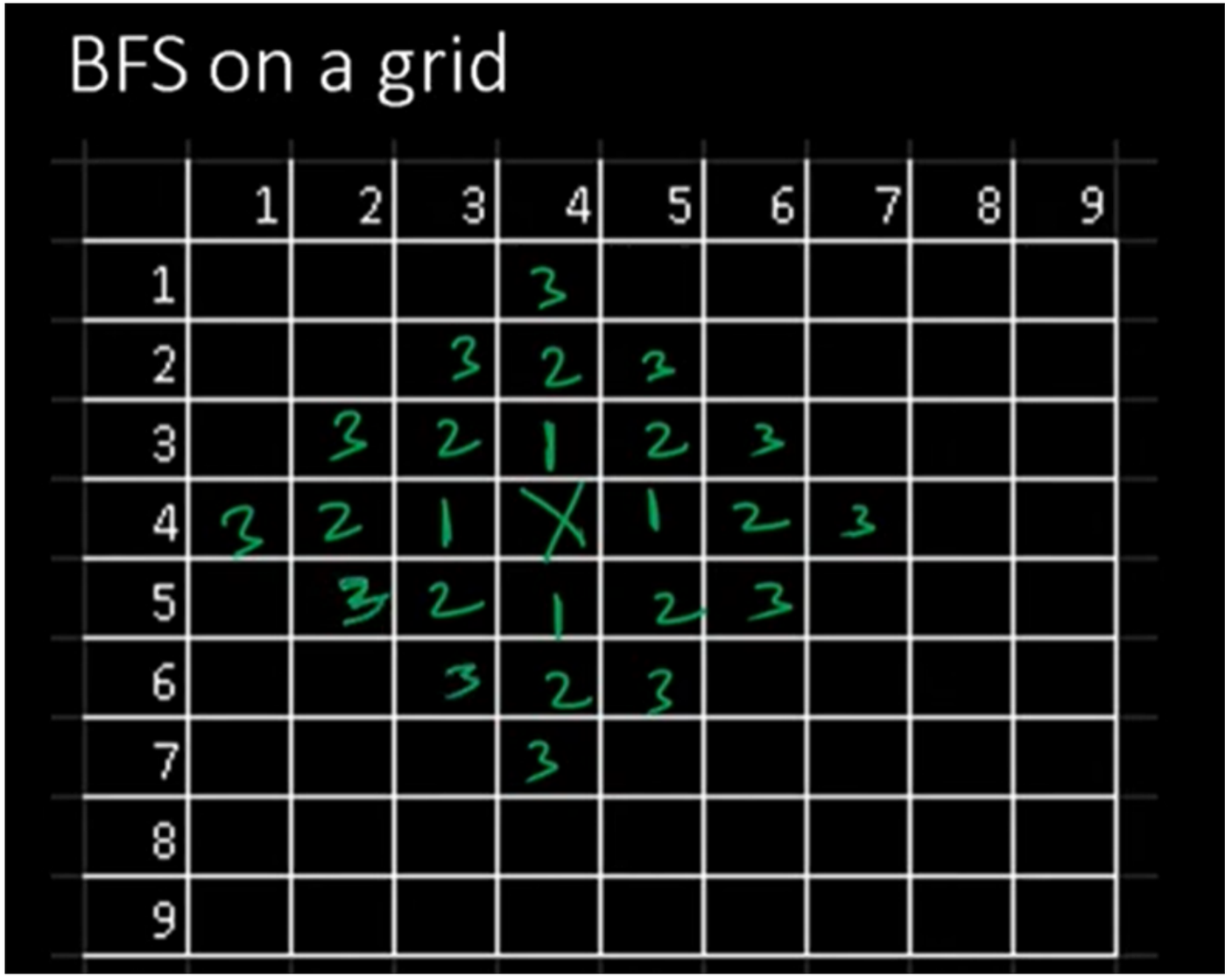
Step 3 :

DFS on GRID

```
void dfs(int i, int j)
{
    vis[i][j] = true;
    for (auto p : moves)
    {
        if (isValid(i + p.first, j + p.second))
        {
            dfs(i + p.first, j + p.second);
        }
    }
}
```

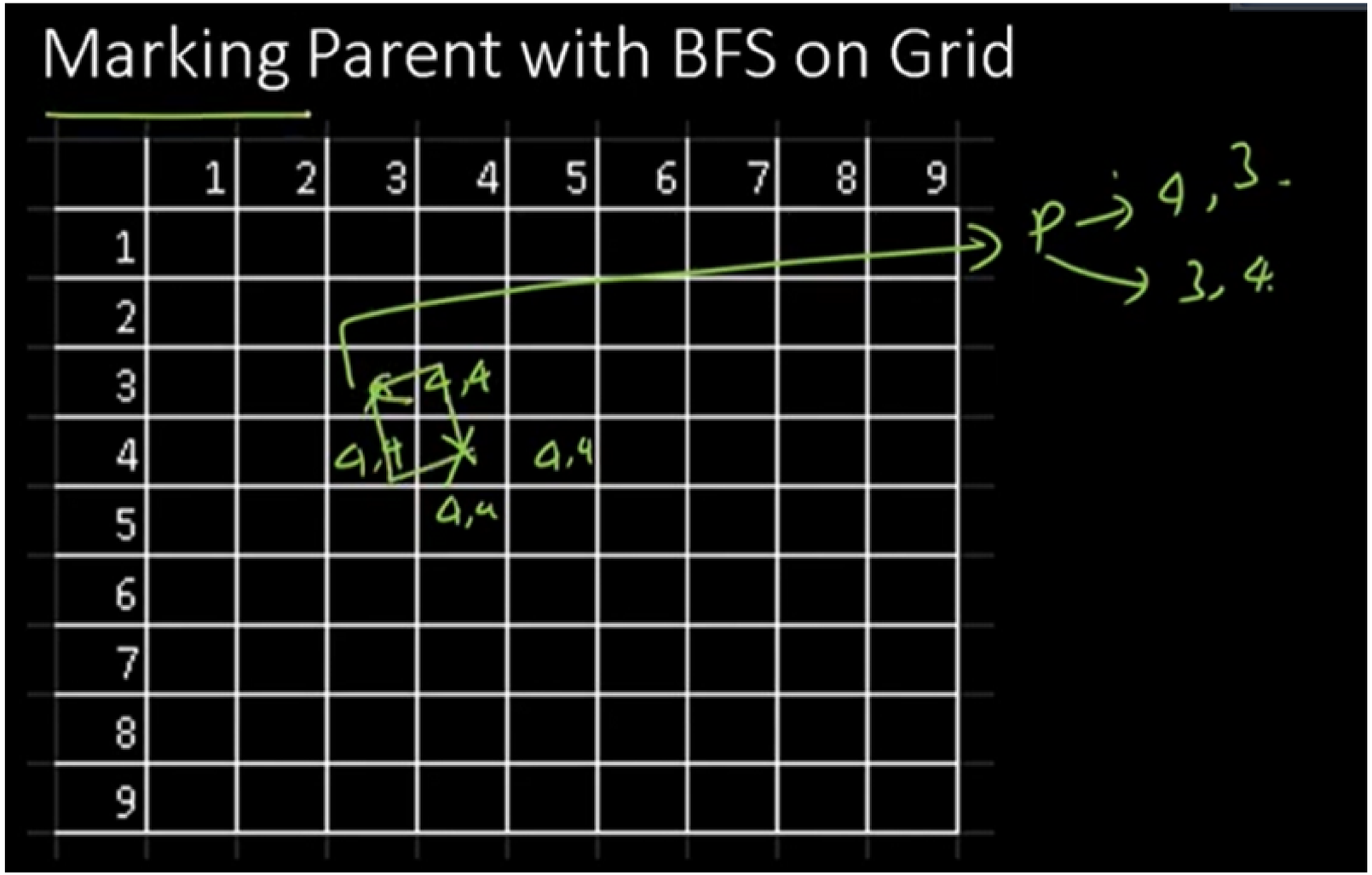
Grid Traversal

Step 3 :
BFS on GRID



Grid Traversal

Step 3 :
BFS on GRID



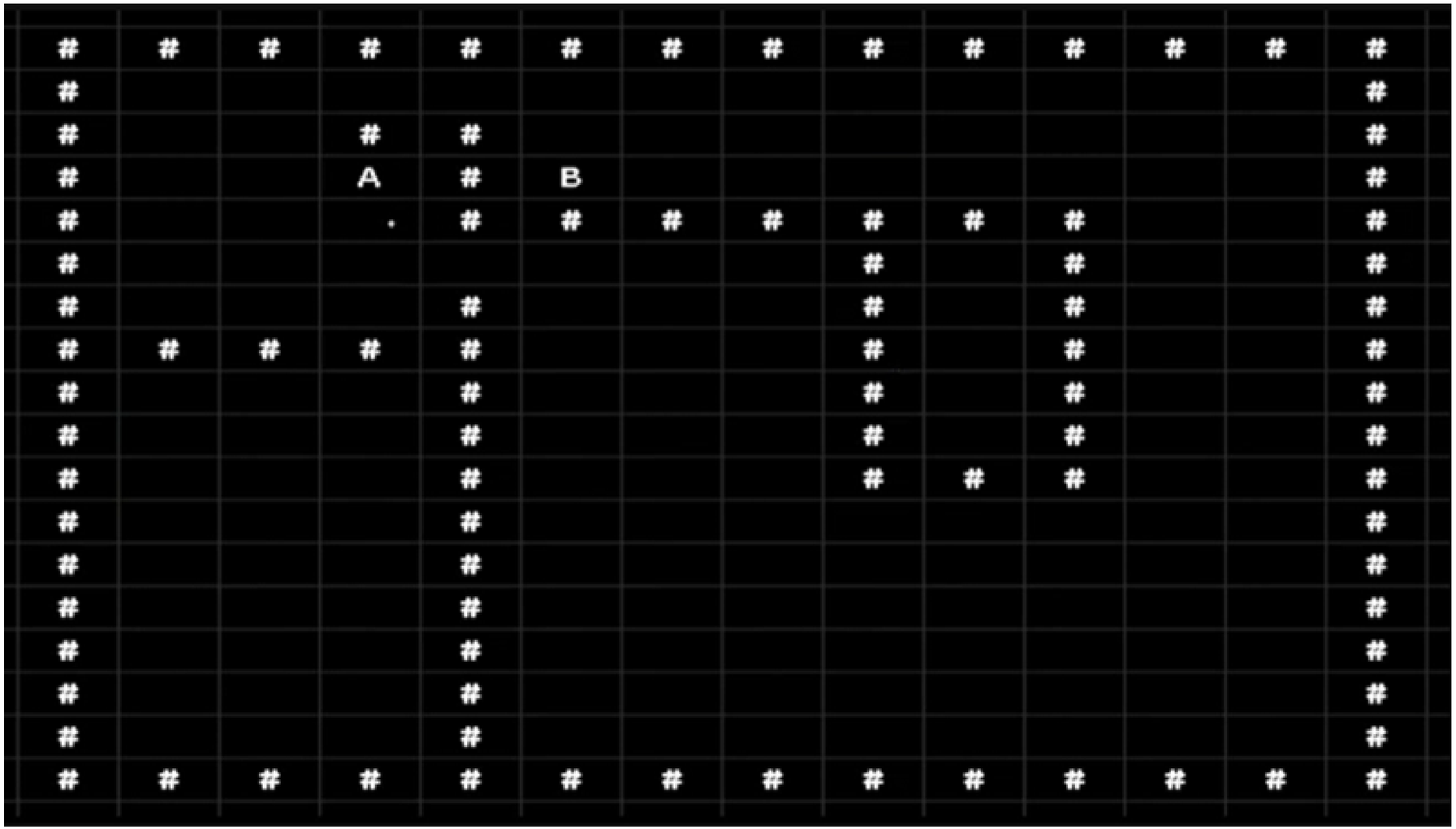
Grid Traversal

Step 3 :
BFS on GRID

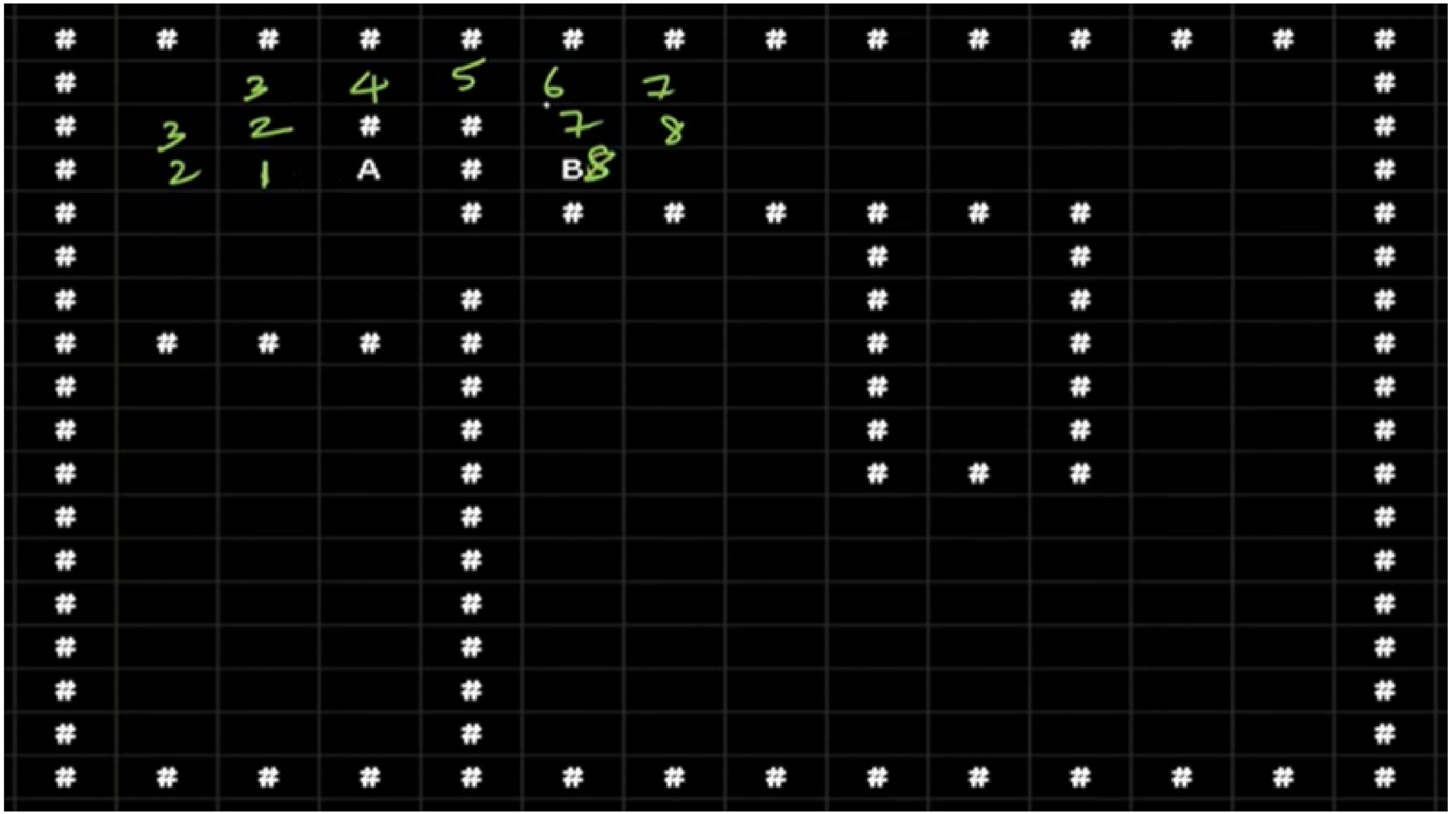
```
void bfs()
{
    queue<pair<int, int>> q;
    q.push({sx, sy});
    while (!q.empty())
    {
        int cx = q.front().first;
        int cy = q.front().second;
        q.pop();
        for (auto mv : moves)
        {
            int mvx = mv.first;
            int mvy = mv.second;
            if (isValid(cx + mvx, cy + mvy))
            {
                q.push({cx + mvx, cy + mvy});
                vis[cx + mvx][cy + mvy] = true;
                path[cx + mvx][cy + mvy] = {mvx, mvy};
            }
        }
    }
}
```

Grid Traversal

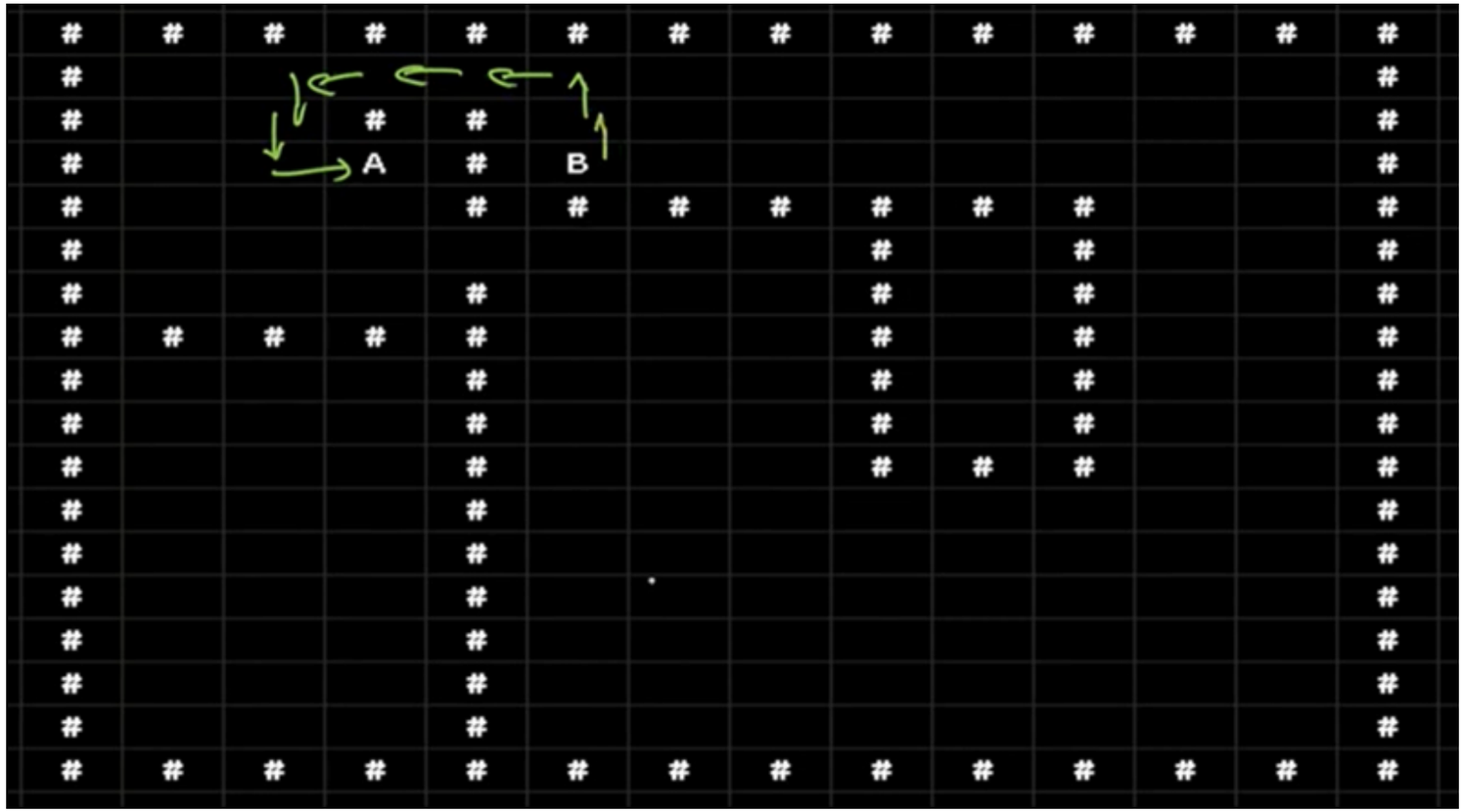
Labyrinth



Labyrinth



Labyrinth



P[B] -> P[P[B]] ---
--->P[A]
then reverse the path

Labyrinth Code

Input Legend

```
int n, m;
vector<vector<pair<int, int>>> path;

vector<vector<bool>> vis;

int sx, sy, ex, ey;

vector<pair<int, int>> moves = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

Labyrinth Code

Checking Cell Validity

```
bool isValid(int x, int y)
{
    if (x < 0 or x >= n or y < 0 or y >= m)
        return false;
    if (vis[x][y])
        return false;
    return true;
}
```

Labyrinth Code

BFS on Grid

```
void bfs()
{
    queue<pair<int, int>> q;
    q.push({sx, sy});
    while (!q.empty())
    {
        int cx = q.front().first;
        int cy = q.front().second;
        q.pop();
        for (auto mv : moves)
        {
            int mvx = mv.first;
            int mvy = mv.second;
            if (isValid(cx + mvx, cy + mvy))
            {
                q.push({cx + mvx, cy + mvy});
                vis[cx + mvx][cy + mvy] = true;
                path[cx + mvx][cy + mvy] = {mvx, mvy};
            }
        }
    }
}
```

Thank You

Code Link :
https://ideone.com/gLxc1q