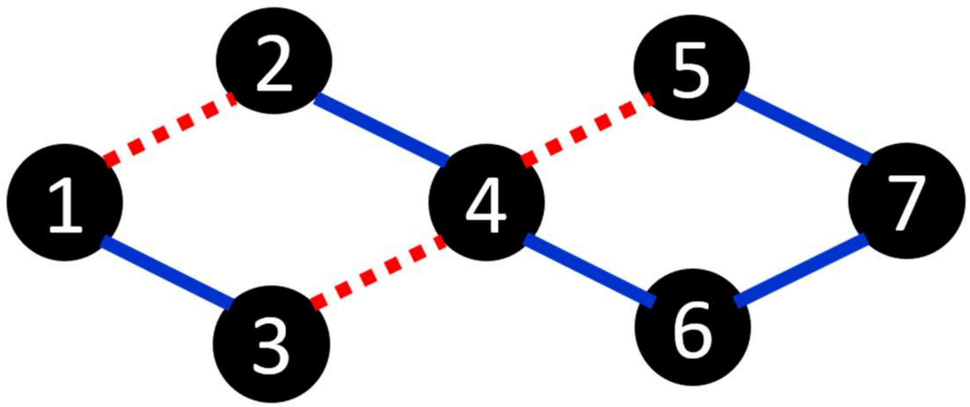
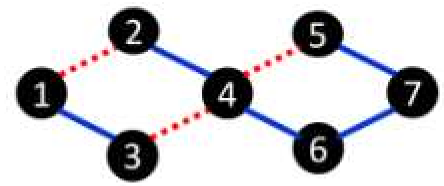
Assumption

Queues, BFS learnt

On Average They're Purple



Alice and Bob are playing a game on a simple connected graph with N nodes and M edges.



Alice colors each edge in the graph red or blue.

A path is a sequence of edges where each pair of consecutive edges have a node in common. If the first edge in the pair is of a different color than the second edge, then that is a "color change."

After Alice colors the graph, Bob chooses a path that begins at node 1 and ends at node N. He can choose any path on the graph, but he wants to minimize the number of color changes in the path. Alice wants to choose an edge coloring to maximize the number of color changes Bob must make. What is the maximum number of color changes she can force Bob to make, regardless of which path he chooses?

Input

The first line contains two integer values N and M with $2 \leq N \leq 100\,000$ and $1 \leq M \leq 100\,000$. The next M lines contain two integers a_i and b_i indicating an undirected edge between nodes a_i and b_i ($1 \leq a_i, b_i \leq N$, $a_i \neq b_i$).

All edges in the graph are unique.

Output

Output the maximum number of color changes Alice can force Bob to make on his route from node 1 to node N.

Goal: Get from 1 to N

- Given
  - Nodes
  - Edges
- Keywords: “regardless of which **path** he chooses”

Sub-Goal: max(colour change)

- Given
  - Single Path (no alternate route)
- Can change colour at every node
- max = path length - 1!

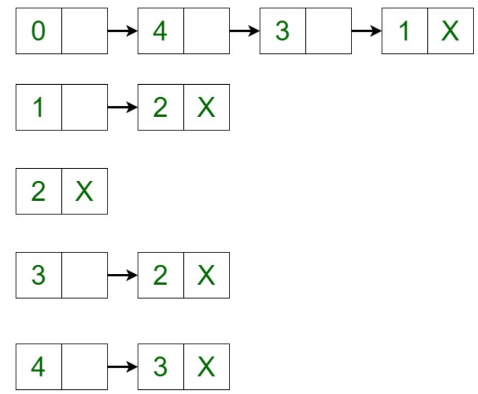
Goal: Get from 1 to N

- Given
  - Nodes
  - Edges
- Keywords: “regardless of which **path** he chooses”
- “maximum colour changes she can force Bob to make”
- $\min\limits_{all\ paths}(path\ length\ - 1)$
- Can Alice guarantee Bob would colour change any more than for the shortest path?
  - No, Bob can just take the shortest path
- Alice colours for shortest path!

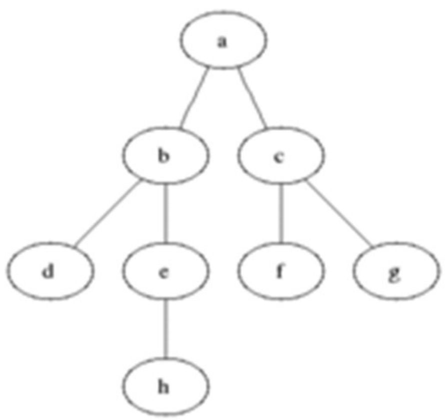
Goal: Shortest Path from 1 to N

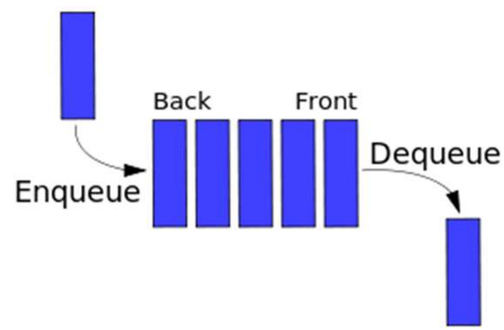
- Given
  - Node
  - Edges
  - Cost? **Equal/constant**
- Which algorithm? Graph SP.
  - BFS? Bellman-Ford? Dijkstra?
- No cost → BFS

Data Structure



- AM/AL/EL?
- BFS iterates through all neighbours → AL
- BFS uses queue





Images from Wikipedia,
GeekforGeeks

BFS Algorithm

function BREADTH-FIRST-SEARCH(graph, goal node) returns a solution, or failure

node ←source node, PATH-COST = 0

if *node* is *goal node* **then return** SOLUTION(*node*)

frontier ← a FIFO queue with node as the only element

explored ← an empty set

loop do

  if EMPTY?(*frontier*) **then return** failure

  node ← POP(*frontier*) /* chooses the shallowest node in frontier */

  for each *neighbour* in *node*.neighbours do

    if *neighbour* is not in *explored* **then**

      if *neighbor* is *goal node* **then return** SOLUTION(*neighbour*)

      add *node* to *explored*

      frontier ← INSERT(*neighbour*, *frontier*)

Adapted from AIMA

C++

```
int n, m, a, b;

cin >> n >> m;
vector<vector<int>> g(n);   // Adjacency List
while (m--) {
        cin >> a >> b;          // Input Edge
        --a; --b;               // 0-based indexing
        g[a].push_back(b);   // Edge from a to b
        g[b].push_back(a);   // Edge from b to a
}

cout << bfs(g, n-1);
```

C++ BFS

```
typedef pair<int, int> ii;

int bfs(vector<vector<int>> &g, int n) {
        queue<ii> q;                                    // Queue (node, distance) pairs
        vector<bool> vis(n, false);                     // Visited Nodes
        int v, d;
        q.push(make_pair(0,0));

        while (!q.empty()) {                            // Use Queue for BFS
                v = q.front().first;
                d = q.front().second;
                q.pop();

                for (int v2 : g[v]) {                   // For each neighbour node
                        if (v2 == n) return d;          // Goal reached
                        if (vis[v2]) continue;          // Skip if visited
                        vis[v2] = true;                 // Make new node visited
                        q.push(make_pair(v2, d+1));     // Add to Queue, +1 distance

        }
        return 0;
}
```

End

Appendix – Bellman-Ford

```
function BellmanFord(list vertices, list edges, vertex source) is
    ::distance[], predecessor[]

    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v] := inf              // Initialize the distance to all vertices to infinity
        predecessor[v] := null          // And having a null predecessor

    distance[source] := 0               // The distance from the source to itself is, of course, zero

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)−1 do //just |V|−1 repetitions; i is never referenced
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"

    return distance[], predecessor[]
```

Source: Wikipedia

Appendix - Dijkstra

function ~~BREADTH-FIRST-SEARCH~~ DIJKSTRA(graph, goal node) returns a solution, or failure

    node ←source node, PATH-COST = 0

    if GOAL-TEST(*node*) **then return** SOLUTION(*node*)

    frontier ← a ~~FIFO~~ priority queue ordered by PATH-COST with node as the only element

    explored ← an empty set

    loop do

        if EMPTY?(*frontier*) **then return** failure

        node ← POP(*frontier*) /* chooses the ~~shallowest~~ lowest-cost node in frontier */

        if GOAL-TEST(*neighbour*) **then return** SOLUTION(*neighbour*)

        add *node* to *explored*

        for each *neighbour* in *node*.neighbours do

            if *neighbour* is not in *explored* **then**

                ~~**if** GOAL-TEST(*neighbour*) **then return** SOLUTION(*neighbour*)~~

                frontier ← INSERT(*neighbour*, *frontier*)

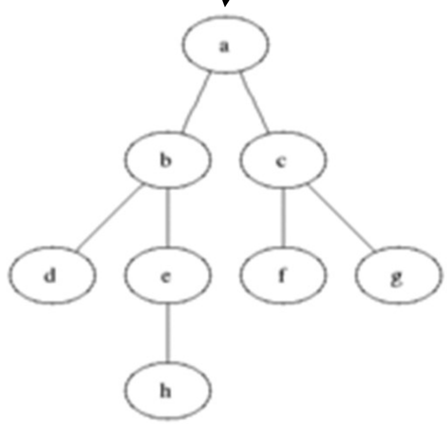
            else if *neighbour* is in *frontier* with higher PATH-COST **then**

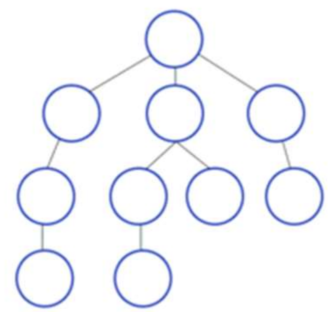
                replace that *frontier* node with *neighbour*

Appendix – BFS vs Dijkstra

	BFS	**Dijkstra**
Main Concept	Visit nodes level by level based on the closest to the source	In each step, visit the node with the lowest cost
Optimality	Gives an optimal solution for unweighted graphs or weighted ones with equal weights	Gives an optimal solution for both weighted and unweighted graphs
Queue Type	Simple queue	Priority queue
Time Complexity	$O(V + E)$	$O(V + E(logV))$

Appendix – BFS vs DFS





Source: Wikipedia