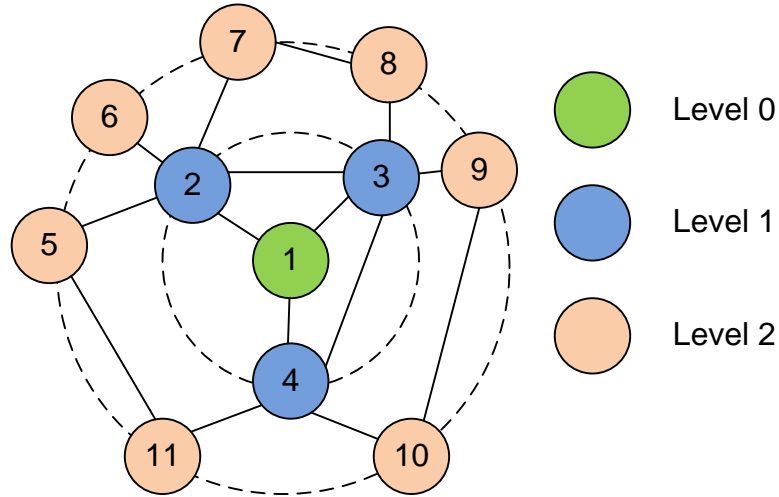


# Breadth-First Search

**Breadth-first search** (bfs) is a traversing algorithm where you start from a given node (source) and traverse the graph layerwise. We start from the source located on the layer 0. Then we visit all vertices on the layer 1, then on layer 2 and so on.



- We start from the vertex 1 (*source*) that belongs to the level 0.
- Then we visit all vertices on the distance 1 from the *source*: 2, 3, 4. These vertices are located on the level 1.
- Then we visit the other vertices located on the level 2.

The level of the vertex  $v$  corresponds to the minimum distance from the *source* to the vertex  $v$ . We shall keep this information in  $\text{dist}[v]$ . So

- $\text{dist}[1] = 0$ ;
- $\text{dist}[2] = \text{dist}[3] = \text{dist}[4] = 1$ ;
- $\text{dist}[5] = \text{dist}[6] = \dots = \text{dist}[11] = 2$ ;

Breadth-first search algorithm finds the **shortest path** from one vertex of the unweighted graph to all others. If we start  $\text{bfs}(\text{source})$ , then  $\text{dist}[v]$  ( $1 \leq v \leq n$ ) contains the length of the shortest path from *source* to  $v$ . Here **length of the path** equals to the number of edges in the path.

**Complexity** of the algorithm  $O(n + m)$ , where  $n$  is the number of vertices,  $m$  is the number of edges.

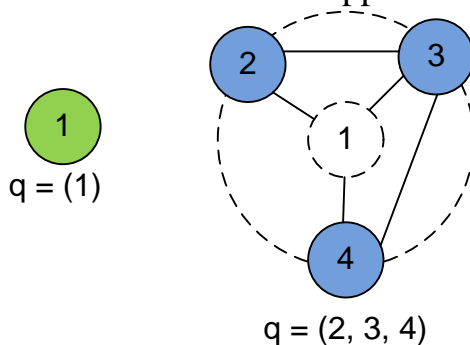
Algorithm can be understood as a process of "**lighting**" the graph: at *zero* step lights only the vertex *source*. At each next step fire spreads from the burning vertex to all its neighbors; that is, in one iteration of the algorithm there is an expansion of "**Ring of Fire**" in breadth per unit (hence the name of the algorithm).

How can we organize a "*Ring of Fire*"? Let's take a queue and push *source* vertex 1 into it:

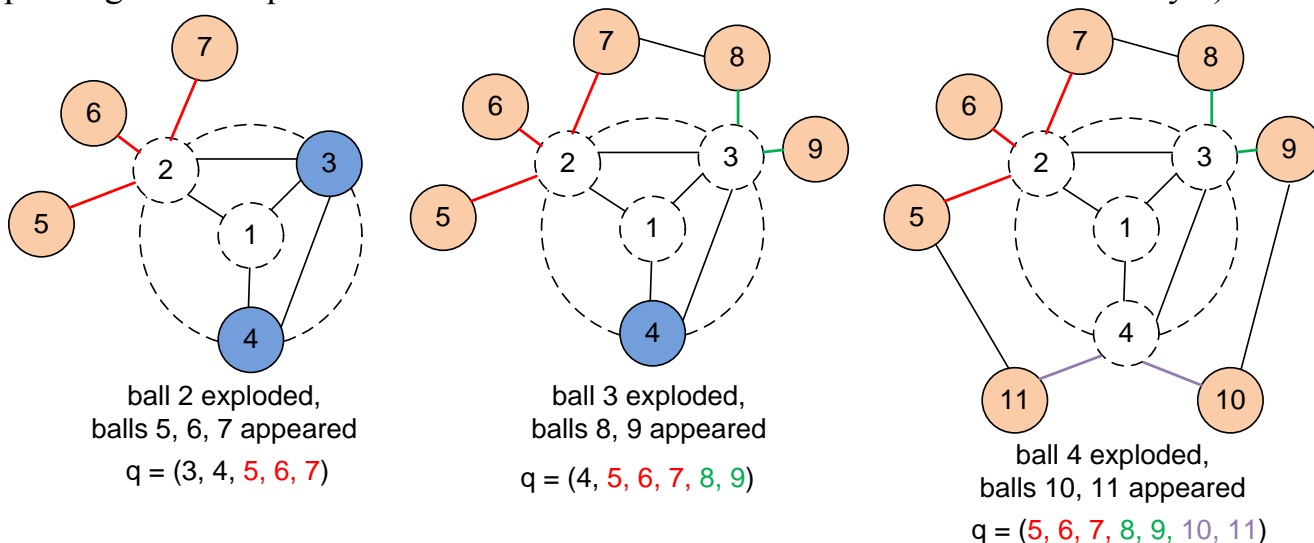
```
queue<int> q;
q.push(1);
```

Now  $q = (1)$ , queue contains only one vertex. Let's *pop* a vertex ( $v = 1$ ) and *push* into queue all vertices connected to it. Vertex 1 is connected with 2, 3 and 4. So we'll push these vertices. Now  $q = (2, 3, 4)$ . What is in the queue? All vertices at the level 1!

Imagine like vertex 1 is exploded and vertices connected to 1 appeared. Vertices of level 0 are exploded and vertices of level 1 are appeared.



Let's continue the process of *explosion* (popping vertex from the queue and pushing into the queue all the vertices connected to it – which are not visited yet).



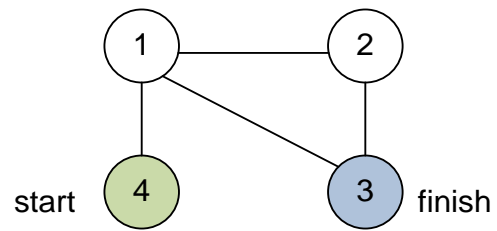
Now vertex 5 will be popped from the queue. No other unvisited vertex is connected to 5. So nothing will be pushed and  $q = (6, 7, 8, 9, 10, 11)$ . Now the vertices will start to be popped from the queue until queue becomes empty:  $q = ()$ . When  $q$  becomes empty, BFS algorithm terminates.

**E-OLYMP 2401. Breadth first search** Undirected graph is given. Find the shortest path from vertex  $s$  to vertex  $f$ .

**Input.** First line contains number of vertices  $n$  ( $n \leq 100$ ) and vertices  $s$  and  $f$  of a graph. Next  $n$  lines describe the adjacency matrix of the graph.

**Output.** Print the minimum distance from  $s$  to  $f$ . If path does not exist, print 0.

Sample input	Sample output
4 4 3 0 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0	2



► Let  $g$  be the adjacency matrix of the graph ( $g[i][j] = 1$  if there exists an edge between vertices  $i$  and  $j$ , and  $g[i][j] = 0$  otherwise),  $dist$  is an array where  $dist[v]$  contains the shortest length from *source* to the vertex  $v$ .  $dist[v] = -1$  means that vertex  $v$  is not used (not visited). The numeration of the vertices in the graph starts from 1 (zero's row and column are not used).

```
#include <cstdio>
#include <vector>
#include <queue>
#include <cstring>
#define MAX 101
using namespace std;

int i, j, n, s, f;
int g[MAX][MAX], dist[MAX];

// breadth first search starts from the vertex `start`
void bfs(int start)
{
    // initialise array dist.
    // dist[i] = -1 means that vertex i is not visited
    memset(dist, -1, sizeof(dist));
    dist[start] = 0;

    // declare and initialize queue
    queue<int> q;
    q.push(start);

    // continue algorithm until queue is not empty
    while (!q.empty())
    {
        // take vertex v from the head of the queue
        // remove vertex v from the queue
        int v = q.front(); q.pop();

        // where can we go from v? Try an edge v -> to
        for (int to = 1; to <= n; to++)
            // if there exists an edge from v to to (g[v][to] == 1)
            // and vertex to is not visited yet (dist[to] = -1)
            if (g[v][to] && dist[to] == -1)
            {
                // push vertex to to queue, calculate dist[to]
                q.push(to);
                dist[to] = dist[v] + 1;
            }
    }
}

int main(void)
```

```

{
    // freopen("bfs.in", "r", stdin);
    // read number of vertices n, starting s and final f vertex
    scanf("%d %d %d", &n, &s, &f);
    // read adjacency matrix
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &g[i][j]);

    // call bfs from the vertex s
    bfs(s);

    // if dist[f] = -1, path is not found, set dist[f] = 0
    if (dist[f] < 0) dist[f] = 0;

    // print the answer
    printf("%d\n", dist[f]);
    return 0;
}

```

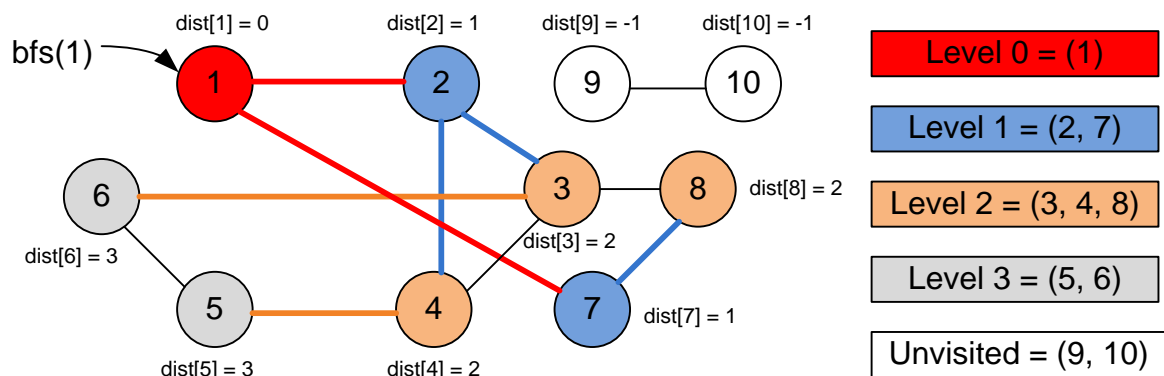
**E-OLYMP 4852. The shortest distance** Directed graph is given. Find the shortest path from the vertex  $x$  to other vertices of the graph.

► Run  $\text{bfs}(x)$ . Use  $\text{dist}$  array to store the shortest distances from  $x$  to other vertices. If there is no path from  $x$  to  $v$ , then  $\text{dist}[v] = -1$ .

**E-OLYMP 5338. Complete Graph - 2** Undirected graph is given with adjacency matrix. Find the shortest path from  $x$  to  $y$ . If path not found, print -1.

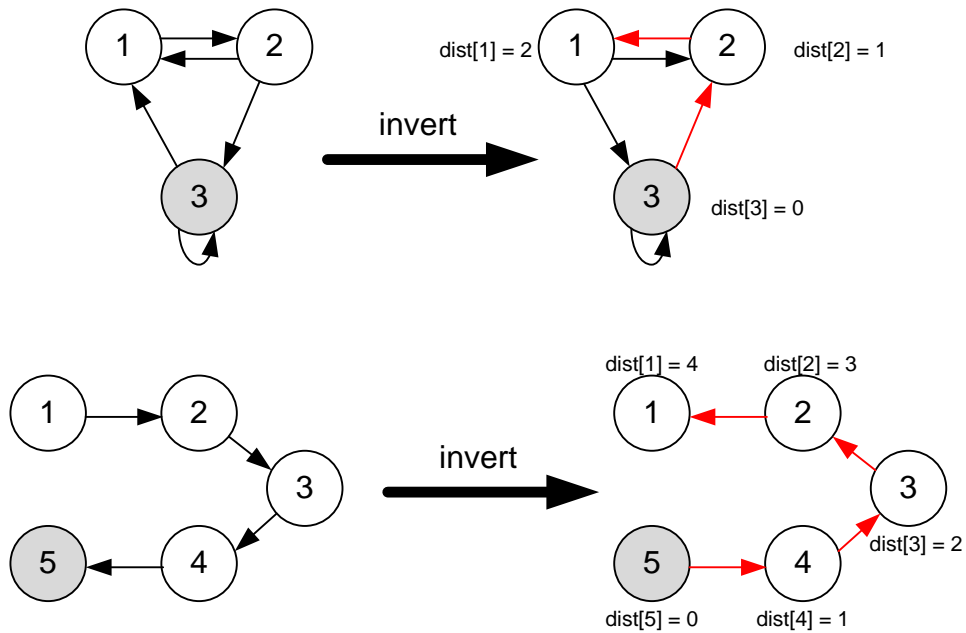
► Use *breadth first search* to find the shortest path.

Let's start breadth first search from the vertex 1. If we go from vertex  $v$  to vertex  $to$ , then  $\text{dist}[to] = \text{dist}[v] + 1$ .



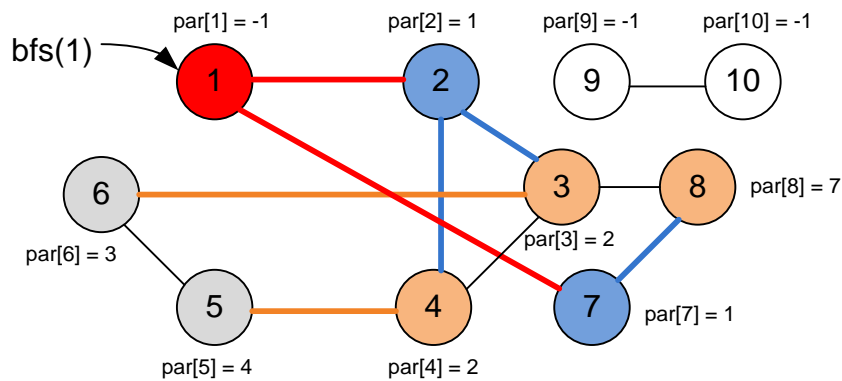
**E-OLYMP 4819. Maximum by minimum** Directed graph is given. Find in it a vertex, the shortest distance from which to the given one  $s$  is maximum, and print this distance.

► Number of vertices is about 5000, you can use adjacency matrix to store the graph. Reverse all edges. Find the shortest distance from  $s$  to all other vertices using bfs. Print the maximum distance.



If we want to restore the shortest path, for each vertex we need to know from which vertex we arrived there. Let  $\text{parent}[v]$  contains this information. If we go from vertex  $v$  to vertex  $to$  along the edge  $v \rightarrow to$ , then  $\text{parent}[to] = v$ . For source  $s$  we have  $\text{parent}[s] = -1$ . If vertex  $v$  is not reachable from the source,  $\text{parent}[v] = -1$ .

v	1	2	3	4	5	6	7	8	9	10
parent[v]	-1	1	2	2	4	3	1	7	-1	-1



How to find the shortest path from *source* to  $v$ ? Let's move backwards from  $v$  until we reach source:

$$v, \text{parent}[v], \text{parent}[\text{parent}[v]], \dots, \text{source}$$

Of course, the path should be printed in the reverse order.

For example, let's find the shortest path from *source* = 1 to  $v = 5$ :

$$5, \text{parent}[5] = 4, \text{parent}[4] = 2, \text{parent}[2] = 1$$

So the shortest path from 1 to 5 is 1, 2, 4, 5

**E-OLYMP 4853. The shortest path** Undirected graph is given. Find the shortest path from  $a$  to  $b$ . Print the length of the shortest path and the path itself.

► Number of vertices is about  $5 * 10^4$ , use adjacency list to store the graph. Run  $\text{bfs}(a)$ . If  $\text{dist}[b] \neq -1$ , the path is found. Use array  $\text{parent}$  to restore the path.

```
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

int i, j, n, m, a, b, u, v;
vector<int> dist, parent;
vector<vector<int>> > g;

void bfs(int start)
{
    // declare arrays
    parent = vector<int>(n + 1, -1);
    dist = vector<int>(n + 1, -1);
    dist[start] = 0;

    // initialize a queue
    queue<int> q;
    q.push(start);

    while (!q.empty())
    {
        // remove vertex v from the queue
        int v = q.front(); q.pop();
        for (int i = 0; i < g[v].size(); i++)
        {
            // there is an edge v -> to
            int to = g[v][i];
            // if vertex v is not visited
            if (dist[to] == -1)
            {
                q.push(to); // push to into the queue
                dist[to] = dist[v] + 1; // recalculate the shortest distance
                parent[to] = v; // if v -> to, parent for to is v
            }
        }
    }
}

int main(void)
{
    scanf("%d %d", &n, &m);
    scanf("%d %d", &a, &b);

    // construct adjacency list
    g.resize(n + 1);
    while (scanf("%d %d", &u, &v) == 2)
    {
        g[u].push_back(v);
        g[v].push_back(u);
    }

    bfs(a); // run bfs from vertex a
}
```

```

if (parent[b] == -1) // if vertex b is NOT reachable, print -1
    printf("-1\n");
else
{
    printf("%d\n", dist[b]); // print shortest distance from a to b
    vector<int> path(1, b); // construct a resulting path

    // b, parent[b], parent[parent[b]], ..., source, -1
    while (parent[b] != -1)
    {
        b = parent[b];
        // insert vertices on the path into vector path
        path.push_back(b);
    }

    // print the shortest path in the order from a to b
    for (i = path.size() - 1; i >= 0; i--)
        printf("%d ", path[i]);
    printf("\n");
}

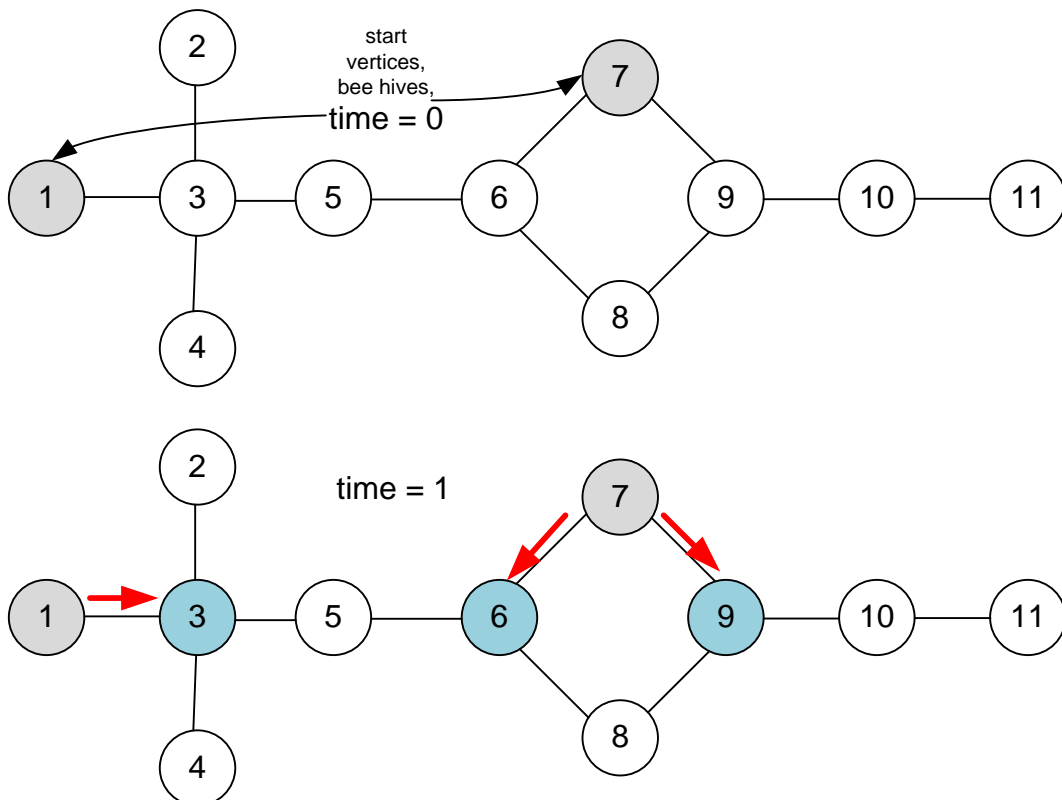
return 0;
}

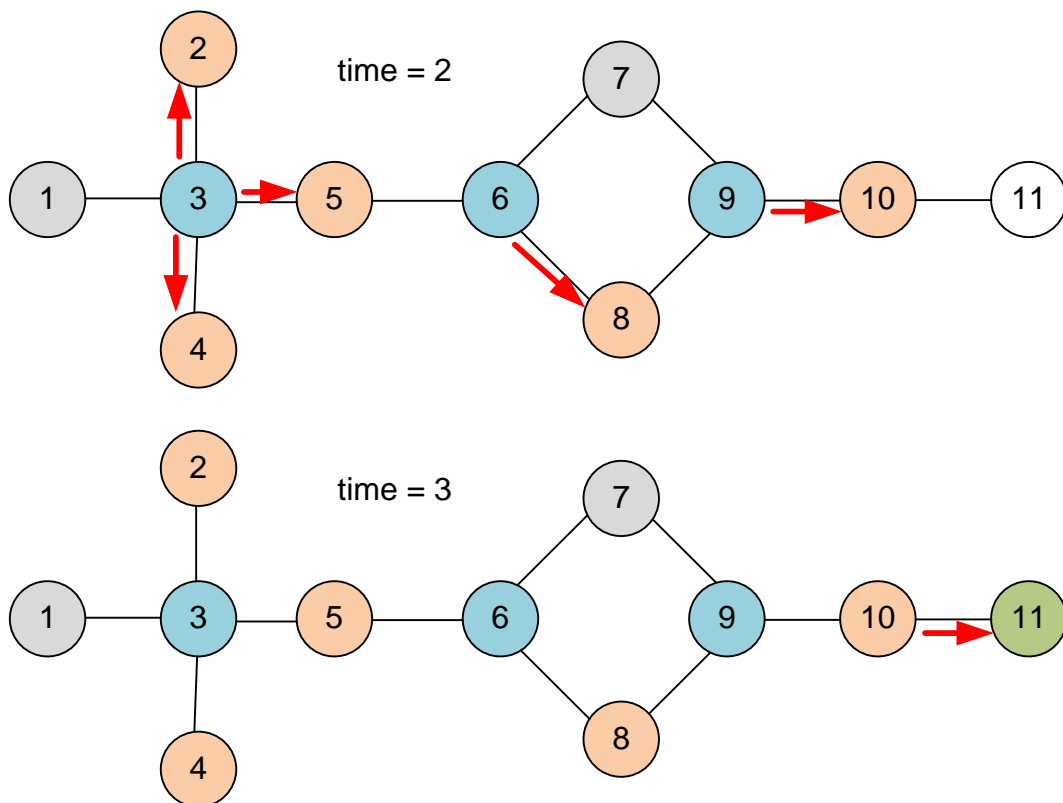
```

How to start *bfs* from **multiple vertices** simultaneously? Imagine we have a graph and some of its vertices are **bee hives**. At the moment *time* = 0 bees start to spread through the graph. Not fly, but spread. It means that at the *time* = 1 bees will be located in the bee hives and in all vertices at distance 1 from bee hives.

Solution is very simple:

***push all started vertices (bee hives) into the queue and start bfs***





**E-OLYMP 4369. Arson** Undirected connected graph is given. Some vertices were fired. Find how many seconds will pass until the last vertex lights up and find this vertex.

► Number of vertices is about  $10^5$ , use adjacency list to store the graph. Insert all vertices that were fired initially into the queue. Run bfs. Find minimum vertex  $v$  for which  $\text{dist}[v]$  is maximum. Print  $\text{dist}[v]$  and  $v$ .

**E-OLYMP 10049. Bitmap** Rectangular bitmap of size  $n * m$  is given. Each pixel of the bitmap is either white or black, but at least one is white. The pixel in  $i$ -th line and  $j$ -th column is called the pixel  $(i, j)$ . The distance between two pixels  $p_1 = (i_1, j_1)$  and  $p_2 = (i_2, j_2)$  is defined as:

$$d(p_1, p_2) = |i_1 - i_2| + |j_1 - j_2|$$

For each pixel find the distance to the nearest white pixel.

► Put the coordinates of all one's in the bitmap into the queue. Start a breadth first search from multiple sources.

Declare the constants.

```
#define INF 0x3F3F3F3F
#define MAX 1002
```

Store the bitmap in an array of strings  $g$ . The shortest distance from the point  $(i, j)$  to the nearest one (array of shortest distances) is stored in  $\text{dist}[i][j]$ .

```
string g[MAX];
int dist[MAX][MAX];
```



Declare a queue that will contain the coordinates of the points.

```
deque<pair<int, int> > q; // (x, y)
```

Adding point  $(x, y)$  to the queue. The shortest distance from it to the nearest point with one is  $d$ .

```
void Add(int x, int y, int d)
{
```

If you go beyond the rectangular area, then ignore the point.

```
    if ((x < 1) || (x > n) || (y < 1) || (y > m)) return;
```

If the value  $\text{dist}[x][y]$  has already been computed, then ignore the point.

```
    if (dist[x][y] != INF) return;
```

Assign the value  $\text{dist}[x][y] = d$ . Push the point  $(x, y)$  into the queue.

```
    dist[x][y] = d;
    q.push_back(make_pair(x, y));
}
```

Function *bfs* implements the breadth first search.

```
void bfs(void)
{
    int x, y;
```

While the queue is not empty, pop the point *temp* and push the coordinates of its four neighbors into the queue.

```
    while (!q.empty())
    {
        pair<int, int> temp = q.front();
        q.pop_front();
        x = temp.first; y = temp.second;
        Add(x + 1, y, dist[x][y] + 1); Add(x - 1, y, dist[x][y] + 1);
        Add(x, y + 1, dist[x][y] + 1); Add(x, y - 1, dist[x][y] + 1);
    }
}
```

The main part of the program. Read the input data.

```
cin >> tests;
while (tests--)
{
    cin >> n >> m;
    for (i = 1; i <= n; i++)
    {
        cin >> g[i];
        g[i] = " " + g[i];
    }
}
```

Initialize the array of shortest distances with infinity.

```
memset(dist, 0x3F, sizeof(dist));
```

Push to the queue  $q$  the coordinates of all points with ones.

```
for (i = 1; i <= n; i++)
for (j = 1; j <= m; j++)
    if (g[i][j] == '1')
    {
        q.push_back(make_pair(i, j));
        dist[i][j] = 0;
    }
```

Run the breadth first serch.

```
bfs();
```

Print the answer – the required distances.

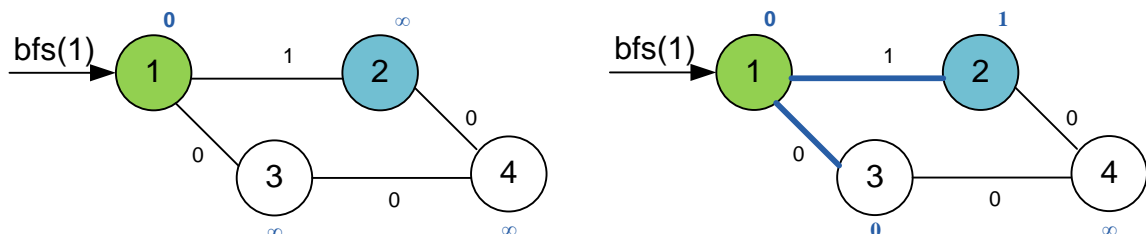
```
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= m; j++)
        cout << dist[i][j] << " ";
    cout << endl;
}
```

**E-OLYMP 10056. Breadth first search 0 - 1** Undirected graph with edges of weight 0 and 1 is given. Find the shortest distance between  $s$  and  $d$ .

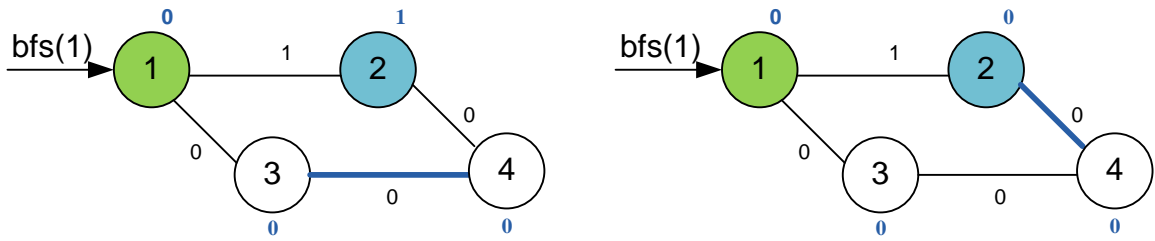
► 0 – 1 graph is given. It is sufficient to slightly modify the breadth-first search. If the distance to vertex is shorter than current found distance, then if the current edge is of zero weight, we add it to the front of the queue, otherwise we add it to the back of the queue.

Initialization:  $\text{dist}[i] = \infty$  ( $2 \leq i \leq 4$ ),  $\text{dist}[1] = 0$ ,  $\text{queue} = (1)$ .

Consider the edges outgoing from vertex 1: 1 – 2 and 1 – 3. Set  $\text{dist}[2] = 1$ ,  $\text{dist}[3] = 0$ ,  $\text{queue} = (3, 2)$  because vertex 3 will be added to the start of the queue, and vertex 2 will be added to the end of the queue.



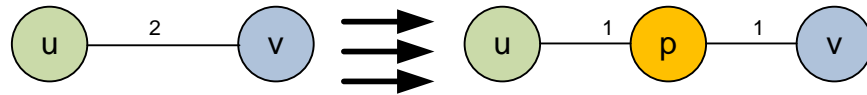
But the value  $\text{dist}[2] = 1$  is not final. Pop the vertex 3 from the queue and relaxate the edge 3 – 4, we get  $\text{dist}[4] = 0$ ,  $\text{queue} = (4, 2)$  because vertex 4 will be added to the start of the queue. Now we relaxate the edges outgoing from vertex 4 and after considering the edge 4 – 2 the value  $\text{dist}[2]$  becomes equal to 0.



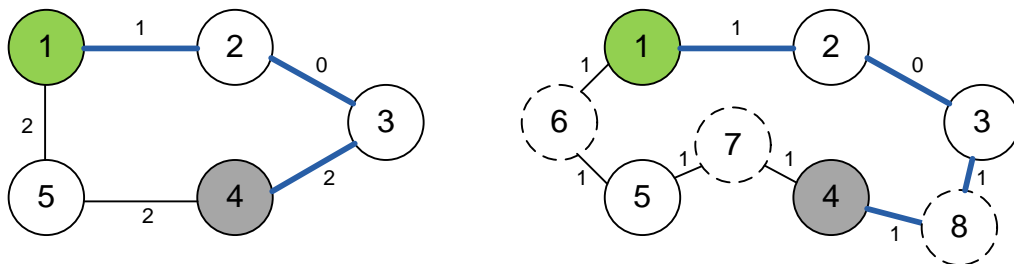
So the value  $\text{dist}[2]$  was assigned two different values: 1 and 0.

**E-OLYMP 10058. Breadth first search 0 – 1 - 2** Undirected graph with edges of weight 0, 1 and 2 is given. Find the shortest distance between  $s$  and  $d$ .

► For each edge  $(u, v)$  of weight 2, we introduce a fictitious vertex  $p$  and replace it with two edges of weight 1:  $(u, p)$  and  $(p, v)$ . Initially, the vertices of the graph are numbered  $1, 2, \dots, n$ . Vertices  $n + 1, n + 2, \dots$  will be declared fictitious. Since there are at most  $m$  edges in the graph, there will be at most  $m$  fictitious vertices.



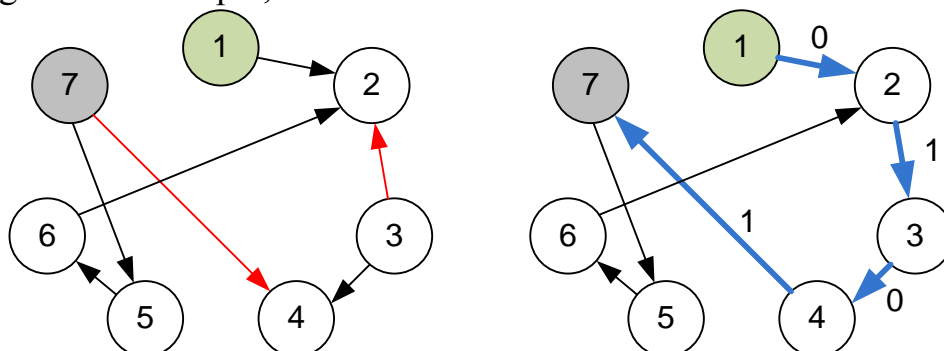
Problem is reduced to breadth-first search on 0 - 1 graph.



**E-OLYMP 10048. Reverse the graph** You are given a directed graph with  $n$  vertices and  $m$  edges. The vertices in the graph are numbered from 1 to  $n$ . What is the minimum number of edges you need to reverse in order to have at least one path from vertex 1 to vertex  $n$ .

► Construct the 0-1 graph. Assign weight 0 to existing edges, and weight 1 to reversed edges. Run the breadth first search. The value of the shortest path from vertex 1 to vertex  $n$  equals to the least number of edges to reverse.

Graph given in a sample, has the form:



**E-OLYMP 10050. Longest path in a tree**

Undirected weighted tree is given. Find the length of the longest path. Find two vertices the distance between which is maximum.

► Select any vertex, for example vertex 1 and run bfs. Find the farthest vertex from vertex 1, let it be  $v$ . Run bfs from vertex  $v$  and find the farthest vertex from it (let it be  $u$ ). Path from  $v$  to  $u$  is the longest.

**Example.** Run bfs(1), the farthest vertex is 7. Run bfs(7), the farthest vertex is 6. Path from 7 to 6 is the longest.

