Last update: January 18, 2023 Translated From: e-maxx.ru

Breadth-first search

Breadth first search is one of the basic and essential searching algorithms on graphs.

As a result of how the algorithm works, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs.

The algorithm works in $O(n + m)$ time, where n is number of vertices and m is the number of edges.

Description of the algorithm

The algorithm takes as input an unweighted graph and the id of the source vertex s. The input graph can be directed or undirected, it does not matter to the algorithm.

The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source s is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the "ring of fire" is expanded in width by one unit (hence the name of the algorithm).

More precisely, the algorithm can be stated as follows: Create a queue q which will contain the vertices to be processed and a Boolean array $used[]$ which indicates for each vertex, if it has been lit (or visited) or not.

Initially, push the source s to the queue and set $used[s] = true$, and for all other vertices v set $used[v] = false$. Then, loop until the queue is empty and in each iteration, pop a vertex from the front of the queue. Iterate through all the edges going out of this vertex and if some of these edges go to vertices that are not already lit, set them on fire and place them in the queue.

As a result, when the queue is empty, the "ring of fire" contains all vertices reachable from the source s, with each vertex reached in the shortest possible way. You can also calculate the lengths of the shortest paths (which just requires maintaining an array of path lengths $d[]$) as well as save information to restore all of these shortest paths (for this, it is necessary to maintain an array of "parents" $p[]$, which stores for each vertex the vertex from which we reached it).

Implementation

We write code for the described algorithm in C++ and Java.

C++

```cpp
vector<vector<int>> adj;  // adjacency list representation
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

Java

```java
ArrayList<ArrayList<Integer>> adj = new ArrayList<>(); // adjacency list
representation

int n; // number of nodes
int s; // source vertex


LinkedList<Integer> q = new LinkedList<Integer>();
boolean used[] = new boolean[n];
int d[] = new int[n];
int p[] = new int[n];

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.isEmpty()) {
    int v = q.pop();
    for (int u : adj.get(v)) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

If we have to restore and display the shortest path from the source to some vertex u, it can be done in the following manner:

C++

```
if (!used[u]) {
    cout << "No path!";
} else {
    vector<int> path;
    for (int v = u; v != -1; v = p[v])
        path.push_back(v);
    reverse(path.begin(), path.end());
    cout << "Path: ";
    for (int v : path)
        cout << v << " ";
}
```

Java

```
if (!used[u]) {
    System.out.println("No path!");
} else {
    ArrayList<Integer> path = new ArrayList<Integer>();
    for (int v = u; v != -1; v = p[v])
        path.add(v);
    Collections.reverse(path);
    for(int v : path)
        System.out.println(v);
}
```

Applications of BFS

- Find the shortest path from a source to other vertices in an unweighted graph.
- Find all connected components in an undirected graph in $O(n+m)$ time: To do this, we just run BFS starting from each vertex, except for vertices which have already been visited from previous runs. Thus, we perform normal BFS from each of the vertices, but do not reset the array $used[]$ each and every time we get a new connected component, and the total running time will still be $O(n+m)$ (performing multiple BFS on the graph without zeroing the array $used[]$ is called a series of breadth first searches).
- Finding a solution to a problem or a game with the least number of moves, if each state of the game can be represented by a vertex of the graph, and the transitions from one state to the other are the edges of the graph.
- Finding the shortest path in a graph with weights 0 or 1: This requires just a little modification to normal breadth-first search: Instead of maintaining array $used[]$, we will now check 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 else we add it to the back of the queue.This modification is explained in more detail in the article 0-1 BFS.

- Finding the shortest cycle in a directed unweighted graph: Start a breadth-first search from each vertex. As soon as we try to go from the current vertex back to the source vertex, we have found the shortest cycle containing the source vertex. At this point we can stop the BFS, and start a new BFS from the next vertex. From all such cycles (at most one from each BFS) choose the shortest.
- Find all the edges that lie on any shortest path between a given pair of vertices (a, b). To do this, run two breadth first searches: one from a and one from b. Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS from b. Now for every edge (u, v) it is easy to check whether that edge lies on any shortest path between a and b: the criterion is the condition $d_a[u] + 1 + d_b[v] = d_a[b]$.
- Find all the vertices on any shortest path between a given pair of vertices (a, b). To accomplish that, run two breadth first searches: one from a and one from b. Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS (from b). Now for each vertex it is easy to check whether it lies on any shortest path between a and b: the criterion is the condition $d_a[v] + d_b[v] = d_a[b]$.
- Find the shortest path of even length from a source vertex s to a target vertex t in an unweighted graph: For this, we must construct an auxiliary graph, whose vertices are the state (v, c), where v - the current node, $c = 0$ or $c = 1$ - the current parity. Any edge (u, v) of the original graph in this new column will turn into two edges $((u, 0), (v, 1))$ and $((u, 1), (v, 0))$. After that we run a BFS to find the shortest path from the starting vertex $(s, 0)$ to the end vertex $(t, 0)$.

Practice Problems

- SPOJ: AKBAR
- SPOJ: NAKANJ
- SPOJ: WATER
- SPOJ: MICE AND MAZE
- Timus: Caravans
- DevSkill - Holloween Party (archived)
- DevSkill - Ohani And The Link Cut Tree (archived)
- SPOJ - Spiky Mazes
- SPOJ - Four Chips (hard)
- SPOJ - Inversion Sort
- Codeforces - Shortest Path
- SPOJ - Yet Another Multiple Problem

- UVA 11392 - Binary 3xType Multiple
- UVA 10968 - KuPellaKeS
- Codeforces - Police Stations
- Codeforces - Okabe and City
- SPOJ - Find the Treasure
- Codeforces - Bear and Forgotten Tree 2
- Codeforces - Cycle in Maze
- UVA - 11312 - Flipping Frustration
- SPOJ - Ada and Cycle
- CSES - Labyrinth
- CSES - Message Route
- CSES - Monsters

Contributors:

back2sqr1 (45.41%) jakobkogler (20.0%) paramsingh (20.0%) Morass (7.57%) adamant-pwn (3.24%) Aryamn (1.08%) PAVBAN95 (1.08%) BemwaMalak (0.54%) wttc-nitr (0.54%) tcNickolas (0.54%)