BFS

Benjamin Schmid

2018-11-10

1 Introduction

1. Given a graph, want to visit all nodes by walking from node to node
2. Could be done in any order
3. Systematic way often preferable
4. E.g. walking in a supermarket
5. Allows us to find properties of graph
6. E.g. components (connected subgraph)
7. Connected: can reach each node from all other nodes
8. E.g. bipartite (two colors for graph, Prelim tables)
9. Want to find shortest path in graph with same lengths (e.g. Water flowing), not possible with DFS

2 Shortest Path

1. Analogy: Network of pipes, fill in water at source, time to sink
2. Want to find shortest path between two nodes
3. Start at origin
4. Go to all direct neighbours
5. Obviously we took shortest path to neighbours
6. Next, go to all neighbours of neighbours we did not yet visit
7. Parent is also neighbour, but visited

8. If a neighbour of a neighbour was already visited it has to be a direct neighbour
9. All other newly visited nodes have a distance of two to the origin
10. Continue this until we visited all nodes
11. Each time we visit a node we know the shortest path to it
12. If it were not the shortest path we would have already found it earlier

3 BFS

1. BFS is doing exactly this
2. Instead of going down as long as possible (DFS) we first explore the breadth
3. Find nodes in order of distance to origin

4 Implementation

1. Implementation of basic BFS: want to find component of start node (i.e. connected nodes)
2. Show example with explicit list by generation
3. Is equivalent to queue
4. Graph given as adjacency list
5. Keep visited flag for each node
6. Use queue to remember next nodes
7. Start at one node: add node to queue
8. Mark start node as visited
9. Take next node from queue
10. Check each neighbour of current node: if not visited, add to queue and mark visited
11. While queue not empty goto 9

```
vector<vector<int>> graph(n);
vector<int> vis(n, 0);
queue<int> q;
q.push(start);
vis[start] = 1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int w : graph[v]) {
        if (vis[w] == 0) {
            vis[w] = 1;
            q.push(w);
        }
    }
}
```

1. Frequent errors: forget visited, visited at wrong position, start not initialized

5 Implementation Shortest Path

1. Change previous implementation to find length of shortest path
2. Add additional list of distances
3. When adding node to queue, set distance

```
vector<vector<int>> graph(n);
vector<int> dist(n, -1);
queue<int> q;
q.push(start);
dist[start] = 0;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    int d = dist[v];
    for (int w : graph[v]) {
        if (dist[w] == -1) {
            dist[w] = d + 1;
            q.push(w);
        }
    }
}
```

1. Possible optimization: break if target reached

6 Implementation Shortest Path with path

1. Change previous implementation to return shortest path
2. Add additional list of parents
3. When adding node to queue, set parent
4. At end, reconstruct path

```
vector<vector<int>> graph(n);
vector<int> par(n, -1);
queue<int> q;
q.push(start);
par[start] = start;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int w : graph[v]) {
        if (par[w] == -1) {
            par[w] = v;
            q.push(w);
        }
    }
}

vector<int> path;
if (par[target] == -1) {
    return path;
}
int current = target;
while (current != par[current]) {
    path.push_back(current);
    current = par[current];
}
path.push_back(start);
return vector<int>(path.rbegin(), path.rend());
```

7 Runtime

1. Each node is visited exactly once
2. Each edge is visited exactly twice
3. Thus runtime of $\mathcal{O}(n + m)$

8 Summary

1. Similar to DFS
2. Find components, check bipartite, find shortest path
3. Shortest path only works if unweighted
4. Useful for many tasks
5. Advanced: also works on directed graphs, find distance to all other nodes, implicit state