CS 3510 Algorithms Last updated: September 24, 2023

Lecture 8: Breadth First Search and Dijkstra's Algorithm

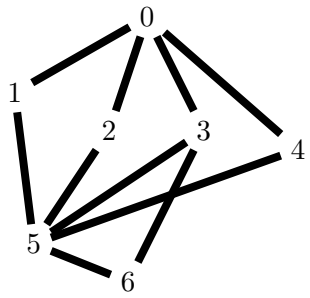
Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

1 Breadth First Search

Breadth First Sarch is comparable to Depth First Search, but it can be used to find the shortest path to all nodes. It works by visiting nodes based on when you first see an edge to them. Another way to think of this is that you visit all nodes of length 1 away, then all nodes of length 2 away all the way to at most nodes of length $|V|$ away. Breadth First Search takes advantage of a queue.

Algorithm 1 Breadth First Search

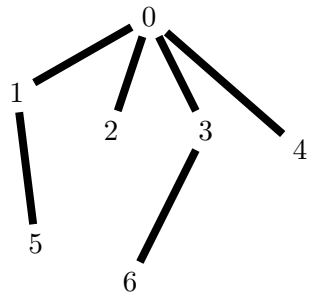
```
1: procedure BreadthFirstSearch(G, r)
2:     Q ← {r}
3:     visited[s] = false ∀s ∈ V
4:     while Q ≠ do
5:         s ← Q.out
6:         visited[s] = true
7:         for v of S do
8:             if visited[v] = false then
9:                 Q.insert(v)
10:            end if
11:        end for
12:    end while
13: end procedure
```



As an example, consider the graph above. Assume that the edges are ordered anti-clockwise; that is, in our implementation of BFS, we visit edges in an anti-clockwise order. We start at the node at the top (here, labeled 0).
In BFS, we first visit the node 0, adding all of the unvisited nodes from 0 into a queue Q. At this moment, we have $Q = \{1, 2, 3, 4\}$. Then, we pop off the first element of the queue

(1), and add all of the unvisited nodes from 1 into Q. When we visit 1, we add 5 into the queue to observe $Q = \{2, 3, 4, 5\}$. From 2, we add no elements, and from 3, we add vertex 6. The algorithm continues, but no new vertices are added, and concludes that all vertices are visited.

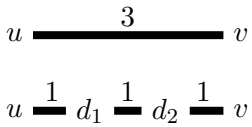
Consider the idea of the BFS-Tree, which is constructed by taking the edges from where each vertex was visited from. We have, as the BFS-Tree of the graph above:



We claim that on a general BFS-Tree, the path from the root to any vertex $v \in V(G)$ is the shortest path in the number of edges. However, instead of using the number of edges, what if we are interested in finding shortest paths on weighted graphs (where the weight of each edges is the "length")? [1] The runtime for BFS is linear $\mathcal{O}(|V| + |E|)$.

To formalize the notion of a "weight", you are given a graph $G = (V, E)$, and now a function $w : E \to \mathcal{R}_{\geq 0}$ which is a parameter of G (that is, each edge is assigned a non-negative weight). [2] We'd like to for any given root node, find all the shortest path (and distance) to all other nodes.

What if we would like to try to use BFS to solve this problem? One approach we could take is to first assume all weights are integers, and for each weight, we add the number of nodes as equal to the weight along each edge:



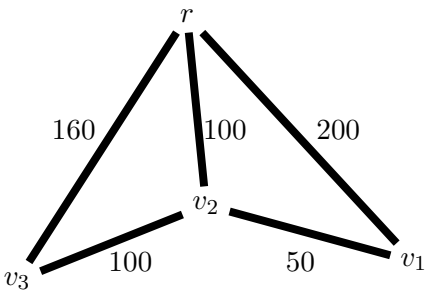
Now you can run BFS, and find the shortest path. Consider the runtime of this though; obviously BFS is linear, but you're adding W nodes and W new edges for each edge, where W is the maximum weight. Then the number of vertices you have will be on the order of $|V| + W|E|$, and the number of edges will be on the order of $W|E|$. Then the total runtime is $|\mathcal{V}| + \mathcal{W}|\mathcal{E}|$, which scales with the factor of the weight. If you recall that numbers of unbounded size are stored in bits, the runtime of this algorithm is exponential.

[1] This is a fairly natural definition, as we may want to represent distances on maps with edges as some notion of distance or time. Consider, for instance, road maps, which may use graphs to denote driving distances between cities.

[2] Having no negative weights is an assumption we will make for this section of the lecture, as it allows us to work graphs that are easier to work with. One problem we could encounter is if there exists a cycle with all negative weights, which would allow us to continuously traverse the cycle to lose distance.

2 Dijkstra's Algorithm

The idea here is that you want to do calculations while ignoring the dummy graph. Ideally we wouldn't need to worry about weights at all in the runtime.



We can trace this graph by hand to gain an intuition for the order in which we should be visiting nodes. Let's start from root r, then try to visit every node in this above graph. Here, the first node we'd hit is v_2, as it has the shortest distance from r. Looking at the distances on the graph, we will hit v_1 through node v_2, and finally we'd hit v_3 directly from the root. The main idea is that we'd like to maintain a list of the vertices we've already hit and how far these vertices are from the root. From this list, select the "cheapest" vertex (i.e. the vertex with the least distance from the root). To do this efficiently, however, we will need to implement a data structure called the priority queue.

The priority queue is a variant of a queue that stores key-value pairs, but is not necessarily first-in-first-out. Instead, it allows the following operations:

- Insert/Update
- Output the (key,value) pair with the minimum value
- Remove key

We'll copy the algorithm from above and make some slight changes. First, we initialize a priority queue:

$$Q = \{(r, 0), (v_1, \infty), (v_2, \infty), \dots\}$$

To demonstrate, we can trace how the algorithm updates Q:

$$Q = \{(r, 0), (v_1, \infty), (v_2, \infty), (v_3, \infty)\}$$

r is popped off, as 0 is the minimum value element. Update the neighbors of r;

$$Q = \{(v_1, 200), (v_2, 100), (v_3, 160)\}$$

v_2 is popped off, and we update $v_1 = 150$, but do not update v_3 as $100 + 100 > 160$;

$$Q = \{(v_1, 150), (v_3, 160)\}$$

v_1 is popped off, and nothing is updated since there are no neighbors of v_1 still in Q;

$$Q = \{(v_3, 160)\}$$

Algorithm 2 Dijkstra's Algorithm

```
1: procedure DIJKSTRA'S(G, r)
2:     Q ← {(r, 0), (v, ∞)}∀v ∈ V − {r}
3:     while Q ≠ do
4:         u ← Q.remove
5:         for (u, v) in E do
6:             if d[v] > d[u] + w(u, v) then
7:                 Q.update(v, (d[u] + w(u, v)))
8:             end if
9:         end for
10:    end while
11: end procedure
```

v_3 is popped off;

$$Q = \emptyset$$

and the algorithm terminates.

What is the runtime of this algorithm? Roughly we perform $|V|$ deletions and $|E|$ updates. Now there are multiple implementations of Priority Queues [3] , and the naive implementation does all of this in $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}|) \log \backslash$. A more advanced implementation like with Fibonacci heap, we can get the runtime down to $\mathcal{O}(|E| + |V| \log |V|)$

3 Minimum Spanning Tree

We'll use the remaining time to start the next section: Minimum Spanning Tree. Essentially in this problem you have a graph $G = (V, E), w : E \to \mathcal{R}$. These graphs are undirected and connected. The goal is to find a subset $T \subset E$ s.t. Tremains connected & $\sum_{e \in T} w(e)$ is minimized. The tree is spanning because the graph remains connected and it is minimum because it has the smallest total weight.

Essentially most of our arguments for Minimum Spanning Trees relies on the fact that if you have some cut between S and $V - S$, the minimum edge in that will appear in the spanning tree.

[3] There are four in the book, you likely know 1-3, but it's worth reading about the rest if you're interested.