CS 3510 Algorithms 2/14/2023

Lecture 7: Shortest Path

Lecturer: Abrahim Ladha *Scribe(s): Joseph Gulian*

Finding the shortest path is a challenging thing in graphs, with many useful applications. There are also many variants: shortest path from one node to another, one node to all, and all nodes to all other nodes. Each of these has its particular use case, but today we'll mainly be focusing on the first one.

1 Breadth-First Search

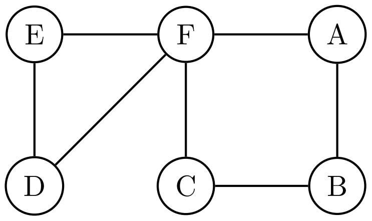

Figure 1: A sample graph.

Say you have an unweighted graph like the one above and you want to find the distance from A to D. You could try running explore on the graph, but that would not work. In this graph for instance, explore would go A to B to C to F to D even though there is a path to D with A to F to D. For sake of understanding the problem, let's look at the shortest path to all nodes, with the edges between them.

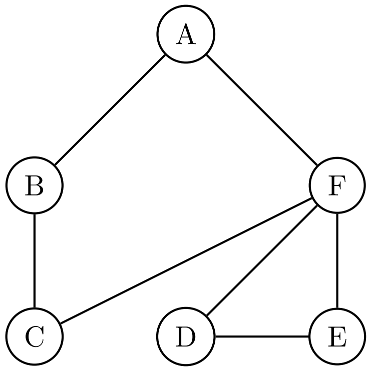

Figure 2: The shortest paths in that graph.

In this graph, we see that there are multiple levels to the graph, each corresponding to the distance from the starting node. The levels correspond to having an edge to a level above. We can simulate this behavior with a queue.

```
def bfs(G, s):
    for all u in V
        dist[u] = inf

    dist[s] = 0

    Q = [s]
    while Q is not empty
        u = eject(Q)
        for all edges (u, v)
            if dist(v) = inf
                inject(Q, v)
                dist[v] = dist[u] + 1
```

Figure 3: Breadth-first search traversal.

This algorithm will place elements in a queue as they are seen. At each step we'll take one element off the queue and compute new distances given that edge. This algorithm works in linear time. Notice that the queue will only have no more than two levels in it at a time, and there will be times it has exactly one level. Consider why this is the case. Lastly notice that this algorithm, like explore will only work on connected graphs. Thats fine, since the problem we care about is shortest paths. We have no path to a node in a disconnected component.

1.1 Search

Search problem are useful all over computing. One example is Chess which can be thought of as a search problem where each state is stored in a node, and the edges represent the moves made by the players. Then you're searching for a win given the nearest state. If you think about this graph, it is somewhat wide, but very deep. On the first turn you have about 18 options, and my games go on about 50 turns. Very roughly this is about 18^{50} possible configurations. To search through all of these using either explore or bfs is going to take a long time, and there are better algorithms. [1] , but for our purposes, use BFS for wide exploration trees and explore for shallow exploration trees.

[1] As you may have learned or be painfully learning in CS 3600.

2 Weighted Shortest Paths

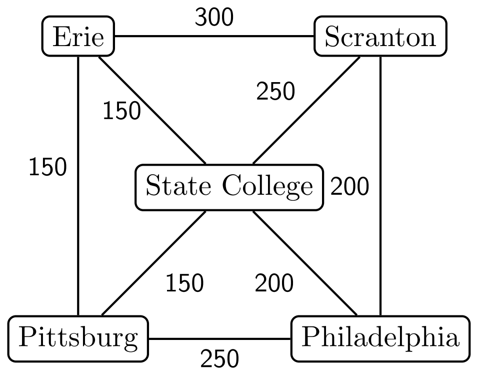

Figure 4: A weighted graph representing cities.

Unweighted graphs aren't as applicable as weighted graphs which offer more for real world applications. Consider the graph representing cities in the if you run breadth-first search from Philadelphia. Here, BFS would say all over two edges to Erie are the same distance even though they aren't.

As we often try to do, let's try to take this problem, turn it into another problem, and use an algorithm for that problem. As we've just presented BFS, let's try to do that. The first idea might be to expand edges into a series of nodes of that length. The problem with this is that your exploration is over the weights rather than the nodes or edges, making the run time now a function of the weights, which could perhaps be huge. Then, let's try to create a new algorithm.

3 The Priority Queue

The priority queue is the primary data structure we'll be talking about, and although there are multiple implementations we'll be discussing, it generally has the following operations.

- Insert - Insert an element into the set.
- Decrease Key - Update the value for a particular element.
- Delete Min - Return the element with the smallest key, and remove it.
- Make Queue - Build the priority queue.

To briefly summarize the implementations, an array is simply searching for and removing the smallest element for deletemin. There are three other implementations: the binary heap [2] , the d-ary heap [3] , and the fibonacci heap [4] . Again, we don't care about the

[2]This data structure you likely learned about in a prior class.
[3]A generalization of the binary heap with d children.
[4]There's a chapter in CLRS if you're curious.

implementation details of these data structures, but we do care about the runtime. We now have enough information to present Djikstra's algorithm.

4 Djikstra's Algorithm

```
def djikstras-algorithm(G, l, s):
    for all u in V
        dist(u) = infinity
        prev(u) = null

    dist(s) = 0
    H = make-queue(V)

    while H is not empty
        u = delete-min(H)
        for all edges (u, v) in E:
            if dist(v) > dist(u) + l(u, v):
                dist(v) = dist(u) + l(u, v)
                prev(v) = u
                decrease-key(H, v)
```

Figure 5: Djikstra's algorithm for shortest paths.

This algorithm is a lot like BFS. It makes a priority queue, iterates over the vertices in the queue, marking them as the shortest path. Note that it also stores `prev`, which can help the get the actual path to the node. For each node, it contains a back node to its parent, along the path of minimum distance from `s`.

Considering the graph of cities, starting at Philadelphia, the algorithm will add Pittsburg, State College and Scranton. Now the priority queue will return either State College or Scranton. If we're given Scranton, it will add Erie with a weight of 500. The new distance to State College is longer, so that won't change. Now another item will be popped off the queue, that being state college. The path to Pittsburg is longer, so it will not be added; the path to Erie is longer though so that will be updated. Now Pittsburg will be explored; the distance to Erie will not be updated. Lastly Erie will be explored.

At a high level you might think of Djikstra's as having a known region and an unknown region. At each step in Djikstra's it chooses to explore one more vertex outside the known region. Specifically it explores the node nearest to the source node. It builds outwards into the unknown region with the nearest vertices in the known region. In this way it will have the shortest path.

This view also illuminates a problem with Djikstra's algorithm. Imagine for a moment, that there is a shortest path from the source, through a vertex outside the known region, to a vertex in the known region. This could cause other vertices outside the known region dependent on this node to cause problem. This can happen if the weight from the node outside the region has a negative weight to the node in the region. [5] Then for Djikstra's we can't have any edges for negative weights. [6]

5 Runtime

Over the course of our discussion we've been ignoring the priority queue. Although this is important for the discussion of runtime. To find runtime based on implementation, we'll count the number of calls to each method. we know we call make-queue once, delete-min is bounded by the number of vertices, and decrease-key/insert is linearly boudned. This makes our runtime equation $\mathcal{O}((\text{make-queue}) + (\text{delete-min})|V| + (\text{decrease-key and insert})(|V| + |E|)$. Of course this is no help to us. We want our runtime in terms of the sets V and E.

implementation	deletemin	insert/decreasekey	runtime
array	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|^2)$
binary heap	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$\mathcal{O}(\frac{d \log \|V\|}{\log d})$	$\mathcal{O}(\frac{\log \|V\|}{\log d})$	$\mathcal{O}((\|V\| \cdot d + \|E\|) \frac{\log \|V\|}{\log d})$
fib heap	$\mathcal{O}(\log \|V\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$

Looking at this graph, you can see that implementation is dependent on which data structure you use, and the fastest turns out to be dependent on your graph. If your graph is dense, you might as well go with an array. Otherwise if you can, choose $d \approx |E|/|V|$.

[5] Think about why this would be.

[6] You will soon learn an algorithm to fix this problem, but it introduces another problem itself.