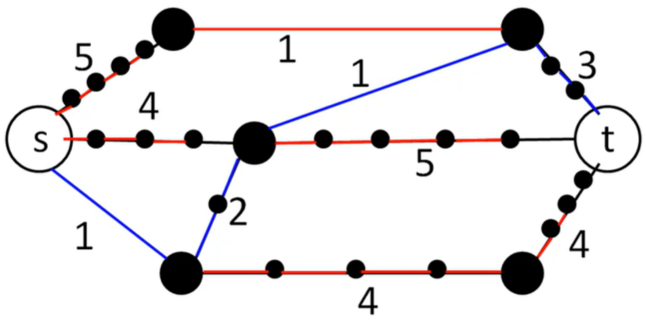
1 Dijkstra's Algorithm & Priority Queues

Given a graph where each edge has some length ℓ, how do we find the shortest path between two vertices?

1.1 Trivial Way

The idea is the break edges into unit length edges. So, an edge of length 5 can actually be seen as 5 edges of length 1. With this conversion, we can run BFS.



Here, we see that the blue path has distance 7 (the lowest) while the red paths are the other paths that the BFS algorithm took.

Now, this *works*. However, there are some issues.

- BFS is linear time for the graph that we run it on. However, we're breaking this graph up so every edge can be represented as a unit edge (so an edge of weight n becomes n edges of weight 1). The runtime is $\mathcal{O}$(Sum of Edge Lengths).
- What if we have an edge of length one million?
- What if our edges have decimal weights (e.g. π, 15.12314352452, so on)?

1.2 Another Way

If we have very long edge lengths, most steps will just consist of advancing slightly (one unit) along a bunch of edges. The issue is, there may be a time period where all we're doing is advancing along one unit for each path that BFS is taking.

So, how do we "fast forward" through these boring steps? Well, occasionally, we have an interesting step where, for instance, the wavefront hits a new vertex. In other words, every time you hit a new vertex, you might need to update some things. But, ignoring that, all we're really doing is slowly advancing through each edge.

1.3 Algorithm

This algorithm is associated with the "ooze" that was mentioned in lecture.

```
Distances(G, s, l)
    dist(v) = 0
    While not all distances found
        Find minimum over (v, w) in E
            with v discovered w not
            of dist(v) + l(v, w)
        dist(w) = dist(v) + l(v, w)
        prev(w) = v
```

1.3.1 Why Does This Work?

Proposition. *Whenever the algorithm assigns a distance to a vertex v, that is the length of the shortest path from s to v.*

Proof. We use induction.

Base Case: We know that `dist(s) = 0`. The empty path has length 0.

Inductive Step: When assigning distances to w, suppose that all previously assigned distances are correct. We can imagine a bubble containing the vertices s and v, with edge denoted by dist(v). By the inductive hypothesis, we assume that all edge lengths inside this bubble is correct. Take a vertex w which is outside of the bubble so that the path length from v (inside the bubble) to w has length $\ell(v, w)$. This is the shortest path from s to any vertex outside the bubble.

$\square$

1.3.2 Runtime of Initial Algorithm

This runs in $\mathcal{O}(|V| \cdot |E|)$ time. There are $\mathcal{O}(|V|)$ iterations and $\mathcal{O}(|E|)$ edges.

- This is too slow because every iteration we have to check every edge.
- The idea is that most of the comparison doesn't change much iteration to iteration. So, we can use this to save time.
- Specifically, record for each w the best value of dist$(v) + \ell(v, w)$.

1.4 Better Algorithm

Our better algorithm is based on the observations that we had above.

```
Distances(G, s, l)
    For v in V
        dist(v) = infinity
        done(v) = false
    dist(s) = 0
    done(s) = false

    while not all vertices done
        Find v not done with minimum dist(v)
        done(v) = true
        For (v, w) in E
            if dist(v) + l(v, w) < dist(w)
                dist(w) = dist(v) + l(v, w)
                prev(w) = v
```

The initialization is $O(|V|)$, and the while loop is $\mathcal{O}(|V|)$. The for loop is $\mathcal{O}(|E|)$ time. Thus, the runtime is:

$$\mathcal{O}(|V|^2 + |E|)$$

- This repeatedly asks for the smallest vertex. Even though not much is changing from round to round, the algorithm is computing the minimum from scratch every time.
- We can use a data structure to help answer a bunch of similar questions faster than answering each question individually (like the one above).

1.5 Priority Queue

A **priority queue** is a data structure that stores elements sorted by a **key** value. Its operations are:

- `insert`: Adds a new element to the priority queue.
- `decreaseKey`: Changes the key of an element of the priority queue to a specified smaller value.
- `deleteMin`: Deletes the element with the lowest key from the priority queue.

1.6 Even Better Priority Queue

```
Dijkstra(G, s, l)
    Initialize Priority Queue Q
    For v in V
        dist(v) = infinity
        Q.insert(v) // using dist(v) as key
    dist(s) = 0
    while Q not empty
        v = Q.deleteMin()
        For (v, w) in E
            if dist(v) + l(v, w) < dist(w)
                dist(w) = dist(v) + l(v, w)
                prev(w) = v
                // w has a faster path and needs to updated in
                // the priority queue
                Q.decreaseKey(w)
```

The runtime is as follows:

- We need to iterate $\mathcal{O}(|V|)$ times for the initial loop.
- In the `while` loop, we run $\mathcal{O}(|V|)$ times.
- We need to run through the edges $\mathcal{O}(|E|)$ times.

So, $\mathcal{O}(|V|)$ `inserts` + $\mathcal{O}(|V|)$ `deleteMins` + $\mathcal{O}(|E|)$ `decreaseKey`.