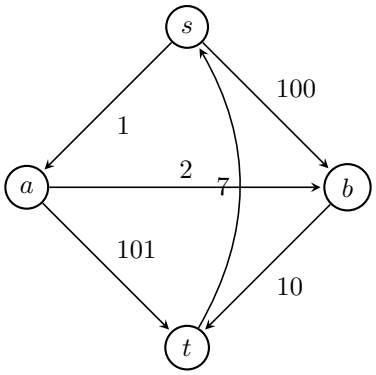
1 The Shortest Path Problem

In this lecture, we'll discuss the shortest path problem. Assume we're given a directed graph $G = (V, E)$ with arbitrary nonnegative weights on edges. The *shortest path* in G from source node s to destination node t is the directed path that minimizes its sum of edge weights. Let $d(s, t)$ denote this sum, also called the *distance* between s and t.



In the above graph $d(s, t) = 1 + 2 + 10 = 13$ whereas $d(t, s) = 7$.

1.1 Variants of Shortest Path Problem

There are three common variants of the shortest path problem.

- *s-t shortest path*: compute the shortest path and the distance between two nodes s and t-
  input : graph G, nodes $s, t \in V$
  output : $d(s, t)$ and possibly a shortest path as well (it has at most $n - 1$ edges).

- *Single source shortest paths (SSSP)*, compute the distances from source node to all nodes -
  input : G, node $s \in V$
  output : $\forall t \in V$, return $d(s, t), \pi(t)$
  where $\pi(t)$ is the node on a shortest $s - t$ path that occurs right before t.

  It is more efficient to store and output $\pi(t)$ than the actual path. Every path from s to t can have length $\Omega(n)$, so enumerating all paths could take $\Omega(n^2)$ output. This output results not only in large space usage but also in at least quadratic time to run the algorithm. Instead, we just output the predecessor nodes $\pi(t)$. It's not hard to see that in order to reconstruct a shortest path from s to u, it is sufficient to know only the node $\pi(u)$ right before each node on the shortest path. (Start from u, look up $\pi(u)$, then look up $\pi(\pi(u))$, continuing until you've found s.) This compressed output only has size $O(n)$, which will allow our SSSP algorithms to run faster than $\emptyset(n^2)$.

  One detail to keep in mind is that in order for our path reconstruction to work, our algorithms must be *consistent*; that is, for every node on a shortest path, if p is the shortest path from s to t, then $\forall x$ on this shortest path, the shortest path from s to x is completely on path p. Intuitively, this is not a very restrictive assumption, it just means we need to break ties between equivalent shortest paths consistently. This tie-breaking scheme should ensure that the shortest paths we pick are simple, so

that there is a well-defined predecessor node for each node. This requirement is more crucial than the consistency described above.

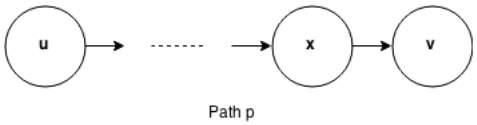
- *All pairs shortest paths (APSP)* -
  input : G
  output : $\forall s, t \in V$, return $d(s,t), \pi(s,t)$ Again, when computing the shortest paths between all pairs of nodes, we compress the output, and for each pair $\pi(s,t)$ we only return the node preceding t on the shortest $s-t$ path.

2 SSSP in unweighted graphs: Breadth First Search (BFS)

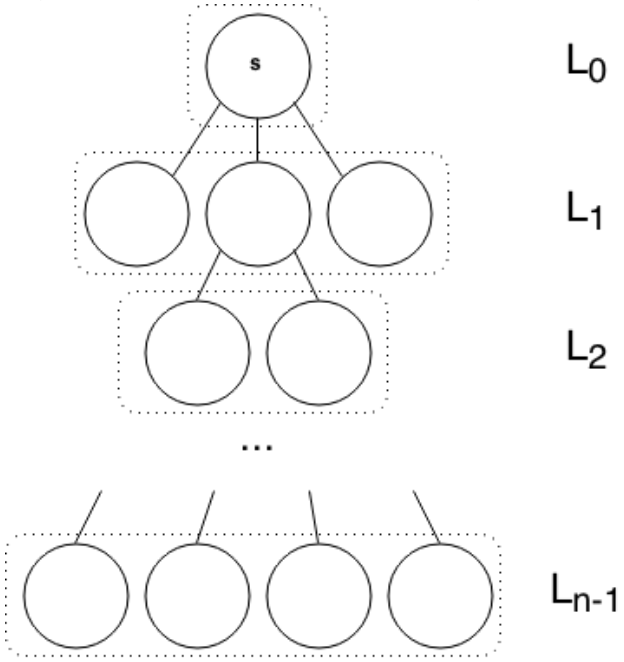
BFS will be similar in spirit to DFS: starting from a node we will traverse edges discovering more and more of the graph. In DFS, we search "deeper" in the graph whenever possible, exploring edges out of the most recently discovered node that still has unexplored edges leaving it. Breadth first search (BFS) instead expands the frontier between discovered and undiscovered nodes uniformly across the breadth of the frontier, discovering all nodes at a distance k from the source node before nodes at distance $k+1$.

BFS(s) computes for every node $v \in G$ the distance from s to v in G. As G is unweighted, $d(u,v)$ is the number of edges on the shortest path from u to v.

A simple property of unweighted graphs is as follows: let P be a shortest $u \to v$ path and let x be the node before v on P. Then $d(u,v) = d(u,x) + 1$.



BFS(s) computes sets L_i, the set of nodes at distance i from s, as seen in the diagram below.



Algorithm 1: BFS(s)

```
Set vis[v] ← false for all v;
Set all L_i for i = 1 to n − 1 to ∅;
L_0 = s;
vis[s] ← true;
for i = 0 to n − 1 do
    if L_i = ∅ then
        exit
    while L_i ≠ ∅ do
        u ← L_i.pop();
        foreach x ∈ N(u) do
            if vis[x] is false then
                vis[x] ← true;
                L_{i+1}.insert(x);
                p(x)← u;
```

2.1 Runtime Analysis

We will now look at the runtime for our BFS algorithm (Algorithm 1) for a graph with n nodes and m edges. All of the initialization above the first for loop runs in $O(n)$ time. Visiting each node within the while loop takes $O(1)$ time per node visited. Everything inside the inner foreach loop takes $O(1)$ time per edge scanned, which we can simplify to a runtime of $O(m)$ time overall for the entire inner for loop.

Overall, we see that our runtime is $O(\#\text{ nodes visited} + \#\text{ edges scanned}) = O(m+n)$.

2.2 Correctness

We will now show that BFS correctly computes the shortest path between the source node and all other nodes in the graph. Recall that L_i is the set of nodes that BFS calculates to be distance i from the source node.

Claim 1. *For all i, $L_i = \{x | d(s,x) = i\}$.*

Proof of Claim 1. We will prove this by (strong) induction on i. Base case ($i = 0$): $L_0 = s$.

Suppose that $L_j = \{x | d(s,x) = j\} \forall j \leq i$ (induction hypothesis for i).

We will show two things: (1) if y was added to L_{i+1}, then $d(s,y) = i+1$, and (2) if $d(s,y) = i+1$, then y is added to L_{i+1}. After proving (1) and (2) we can conclude that $L_{i+1} = \{y | d(s,y) = i+1\}$ and complete the induction.

Let's prove (1). First, if y is added to L_{i+1}, it was added by traversing an edge (x,y) where $x \in L_i$, so that there is a path from s to y taking the shortest path from s to x followed by the edge (x,y), and so $d(s,y) \leq d(s,x)+1$. Since $x \in L_i$, by the induction hypothesis, $d(s,x) = i$, so that $d(s,y) \leq i+1$. However, since $y \notin L_j$ for any $j \leq i$, by the induction hypothesis, $d(s,y) > i$, and so $d(s,y) = i+1$.

Let's prove (2). If $d(s,y) = i+1$, then by the inductive hypothesis $y \notin L_j$ for $j \leq i$. Let x be the node before y on the $s \to y$ shortest path P. As $d(s,y) = i+1$ and the portion of P from s to x is a shortest path and has length exactly i. Thus, by the induction hypothesis, $x \in L_i$. Thus, when x was scanned, edge (x,y) was scanned as well. If y had not been visited when (x,y) was scanned, then y will be added to L_{i+1}. Hence assume that y was visited before (x,y) was scanned. However, since $y \notin L_j$ for any $j \leq i$, y must have been visited by scanning another edge out of a node from L_i, and hence again y is added to L_{i+1}.

□

2.3 BFS versus DFS

If you simplify BFS and DFS to the basics, ignoring all timestamps and levels that we would usually create, BFS and DFS have a very similar structure. Breadth first search explores the nodes closest and then moves outwards, so we can use a queue to put new nodes at the end of the list and pull the oldest/nearest nodes from the top of the list. Depth first search goes as far down a path as it can before coming back to explore other options, so we can use a stack which pushes new nodes on the top and also pulls the newest nodes from the top. See the pseudocode below for more detail.

Algorithm 2: DFS(s) \\ s is the source node

```
T ← stack
push s onto T
while T is not empty do
    u ← pop from top of T
    push all unvisited neighbors of u on top of stack T
```

Algorithm 3: BFS(s) \\ s is the source node

```
T ← queue
push s onto T
while T is not empty do
    u ← pop from front of T
    push all unvisited neighbors of u on back of queue T
```

3 SSSP in graphs with nonnegative weights: Dijkstra's Algorithm

Now we will solve the single source shortest paths problem in graphs with nonnengative weights using Dijkstra's Algorithm. The key idea, that Dijkstra will maintain as an invariant, is that $\forall t \in V$, the algorithm computes an estimate of the distance of t from source $d[t]$ such that -

1. At any point in time, $d[t] \geq d(s,t)$, and
2. when t is finished, $d[t] = d(s,t)$

Algorithm 4: Dijkstra$(G = (V, E), s)$

```
∀t ∈ V, d[t] ← ∞ // set initial distance estimates
d[s] ← 0
F ← {v | ∀v ∈ V} // F is set of nodes that are yet to achieve final distance estimates
D ← { } // D will be set of nodes that have achieved final distance estimates
while F ≠ { } do
    x ← element in F with minimum distance estimate
    for (x, y) ∈ E do
        d[y] ← min{d[y], d[x] + w(x, y)} // "relax" the estimate of y
        // to maintain paths: if d[y] changes, then π(y) ← x
    F ← F \ {x}
    D ← D ∪ {x}
```

We will prove that Dijkstra correctly computes the distances from s to all $t \in V$.

Claim 2. *For every u, at any point of time $d[u] \geq d(s,u)$.*

A formal proof of this claim proceeds by induction. In particular, one shows that at any point in time, if $d[u] < \infty$, then $d[u]$ is the weight of some path from s to t. Thus at any point $d[u]$ is at least the weight of the *shortest* path, and hence $d[u] \geq d(s,u)$.

As a base case, we know that $d[s] = 0 = d(s,s)$ and all other distance estimates are $+\infty$, so we know that the claim holds initially. Now, when $d[u]$ is changed to $d[x]+w(x,u)$ then (by the induction hypothesis) there is a path from s to x of weight $d[x]$ and an edge (x,u) of weight $w(x,u)$. This means there is a path from s to u of weight $d[u] = d[x] + w(x,u)$. This implies that $d[u]$ is at least the weight of the shortest path $= d(s,u)$, and the induction argument is complete.

Claim 3. *When node x is placed in D, $d[x] = d(s,x)$.*

Notice that proving the above claim is sufficient to prove the correctness of the algorithm since $d[x]$ is never changed again after is added to D: the only way it could be changed is if for some node $y \in F$, $d[y]+w(y,x) < d[x]$ but this can't happen since $d[x] \leq d[y]$ and $w(y,x) \geq 0$ (all edge weights are nonnegative). The assertion $d[x] \leq d[y]$ for all $y \in F$ stays true at all points after x is inserted into D: assume for contradiction that at some point for some $y \in F$ we get $d[y] < d[x]$ and let y be the first such y. Before $d[y]$ was updated $d[y'] \geq d[x]$ for all $y' \in F$. But then when $d[y]$ was changed, it was due to some neighbor y' of y in F, but $d[y'] \geq d[x]$ and all weights are nonnegative, so we get a contradiction

We prove this claim by induction on the order of placement of nodes into D. For the base case, s is placed into D where $d[s] = d(s,s) = 0$, so initially, the claim holds.

For the inductive step, we assume that for all nodes y currently in D, $d[y] = d(s,y)$. Let x be the node that currently has the minimum distance estimate in F (this is the node about to be moved from F to D). We will show that $d[x] = d(s,x)$ and this will complete the induction.

Let p be a shortest path from s to x. Suppose z is the node on p closest to x for which $d[z] = d(s,z)$. We know z exists since there is at least one such node, namely s, where $d[s] = d(s,s)$. By the choice of z, for every node y on p between z (not inclusive) to x (inclusive), $d[y] > d(s,y)$. Consider the following options for z.

1. If $z = x$, then $d[x] = d(s,x)$ and we are done.

2. Suppose $z \neq x$. Then there is a node z' after z on p. (Here it is possible that $z' = x$.) We know that $d[z] = d(s,z) \leq d(s,x) \leq d(x)$. The first $\leq$ inequality holds because subpaths of shortest paths are shortest paths as well, so that the prefix of p from s to z has weight $d(s,z)$. In addition, the weights on edges are non-negative, so that the portion of p from z to x has a nonnegative weight, and so $d(s,z) \leq d(s,x)$. The subsequent $\leq$ holds by Claim 1. We know that if $d[z] = d[x]$ all of the previous inequalities are equalities and $d[x] = d(s,x)$ and the claim holds.

   Finally, towards a contradiction, suppose $d[z] < d[x]$. By the choice of $x \in F$ we know $d[x]$ is the minimum distance estimate that was in F. Thus, since $d[z] < d[x]$, we know $z \notin F$ and must be in D, the finished set. This means the edges out of z, and in particular (z,z'), were already relaxed by our algorithm. But this means that $d[z'] \leq d(s,z) + w(z,z') = d(s,z')$, and because z is on the shortest path from s to z', the distance estimate of z' must be correct. However, this contradicts z being the closest node on p to x meeting the criteria $d[z] = d(s,z)$. Thus, our initial assumption that $d[z] < d[x]$ must be false and $d[x]$ must equal $d(s,x)$.

Now that we have proved the correctness of Dijkstra, we'll look at how long it will take to run. We'll see that the runtime is highly-dependent on the data structure used to store F.

The initialization step takes $O(n)$ operations to set n distance estimate values to infinity and one to 0. Then, within the while loop, we make $O(n)$ calls to finding node with $\min(d[x])$ from F (this is called `FindMin`) as nodes are never re-inserted into F. Additionally, there will be $O(n)$ calls to deleting $min(d[x])$ node from F (`DeleteMin`). Finally, each edge will be relaxed at most once, each in constant time, for a total

of $O(m)$ work. During these relaxations, there will be $O(m)$ potential decreases of distance estimates $d[u]$ (`DecreaseKey`).

Thus, the total runtime of Dijkstra's algorithm depends on how quickly our data structure can support each call to `FindMin`, `DeleteMin`, and `DecreaseKey`. That is, the runtime is on the order of

$$n \cdot (FindMin(n) + DeleteMin(n)) + m \cdot DecreaseKey(n).$$

Consider implementing Dijkstra using the following data structures.

- Store F as an array where each slot corresponds to a node and where $F(j) = d[j]$ if $j \in F$, else NIL.
  DecreaseKey runs in $O(1)$ as key is indexed.
  FindMin and DeleteMin run in $O(n)$ as the array is not sorted.
  The total runtime is $O(m + n^2) = O(n^2)$

- Store F as a Red-Black tree.
  DecreaseKey and FindMin run in $O(\log n)$ time. You implement DecreaseKey by deleting and reinserting with the new key.
  Total runtime is $O((m+n)\log n)$, which improves over the array implementation, except in the special case when $m = \Theta(n^2)$.

- Store F as a Fibonacci Heap. Fibonacci Heaps are a complex data structure, which is able to support the operations Insert in $O(1)$, FindMin in $O(1)$, DecreaseKey in $O(1)$ and DeleteMin in $O(\log n)$ "amortized" time, over a sequence of calls to each operation. The meaning of amortized time in this case is as follows: starting from an empty Fibonacci Heaps data structure, any sequence of operations that includes a Inserts, b FindMins, c DecreaseKeys and d DeleteMins take $O(a+b+c+d\log n)$ time. Thus, the outcome is that Dijkstra's algorithm can be implemented to run in $O(m + n\log n)$ time.