Notes for CIS 3200: Introduction to Algorithms
Graph Algorithms

Erik Waingarten

Contents

1 Breadth First Search (BFS)

We are going to describe breadth first search for *undirected* graphs. The algorithmic problem we will consider is that of traversing a graph with a breadth-first search.

- **Input**: We receive as input a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, and we also receive an input source vertex $s \in V$.
- **Output**: We want to output all of the vertices in the *connected component* of s. This is all vertices for which there exists a path along the graph which connects s to that vertex.

Oftentimes, we will want to perform this search so that we can determine the *distance* from s to every other vertex in the connected component. This new notion of distance in a graph, requires a new definition.

Definition 1.1. *Let $G = (V, E)$ denote an unweighted graph and let $s, t \in V$ be any two vertices.*

- *Then, s and t are* connected *if there exists a sequence of edges $(p_0, p_1), (p_1, p_2), \ldots, (p_{k-1}, p_k)$, where $s = p_0$ and $t = p_k$.*
- *Thus, for any two connected $s, t \in V$, the* distance *from s to t, which we denote $\delta(s, u)$, is the number of edges on the shortest path from s to t, i.e.,*

$$\delta(s, u) = \min_{P \text{ path from } s \text{ to } t} |P|.$$

We note that the distance that we defined above satisfies natural properties of distance:

- Distances are non-negative and 0 only for $\delta(s, s)$.
- Distance are symmetric, $\delta(s, t) = \delta(t, s)$.
- Distances satisfy the "triangle inequality," for any three vertices s, u, t,

$$\delta(s, t) \leq \delta(s, u) + \delta(u, t).$$

As we will see, the algorithm for breadth first search will allow us to output every vertex in the connected component of s, and in addition, compute $\delta(s, t)$ for every t in the connected component.

In addition, it can also compute a *shortest path tree*, which is a tree rooted at s such that every vertex in the connected component is in the tree, and the connections will specify shortest paths in the tree. In this way, the depth of a node t in the tree exactly will specify $\delta(s, t)$.

Algorithm. Roughly speaking, we will start the search from s, and maintain a "frontier" of the vertices we can explore. In each step, we will expand our frontier by one edge, thereby exploring more. Concretely, the textbook describes the algorithm by considering a coloring of the vertices, where at any points,

- We have unvisited vertices (which are colored white)
- We have visiting vertices which have been seen but not explored (these are gray),
- We have the explored vertices, which are the ones which we've seen and explored (these are black).

Each vertex will maintain a variable $\ell(v)$ which will prove that $\delta(s, v) = \ell(v)$, it will keep the parent of the vertex $p(v)$ which will become the parent in the shortest path (BFS) tree, and it will maintain the color of each node $c(v)$.

- **Initialization**: We initialize all vertices to have color $c(v)$ be white (as these are unvisited), their levels $\ell(v)$ are infinite (as they are not connected), and have parents $p(v)$ set to null (since they are isolated). For the source vertex s, we let s be the root (so its parent $p(s)$ will be set to null), $\ell(s) = 0$ since it will be the root of the tree, and we make the color $c(s)$ be gray. Furthermore, we will initialize a queue which is initialized to contain s—in general, the queue will maintain the gray vertices.
- **Loop-Condition**: Here, we will want to define a loop that iteratively will "grow" or "improve" the current solution. In this case, our condition for not being done (and the condition that we loop for) is that the current queue is non-empty. In particular, whenever the queue is non-empty, we are still "exploring" a vertex and we are not yet done "exploring" those vertices.
- **Loop**: Now, in each iteration of the loop, we should explore the vertices in the queue. We do this by:
  - Pick a vertex u from the queue.
  - For that vertex u, we iterate through all of the edges which are connected to u and we check the neighbor v.
  - If the vertex v has color $c(v)$ being gray or black, we have either explored it or it is currently in the queue and we do nothing.
  - However, if $c(v)$ is white, then we set $\ell(v) = \ell(u) + 1$, and $p(v) = u$. We update its color $c(v)$ to gray and put v into the queue.
  - Once we are done iterating through neighbors of u, we remove u from the queue and we set its color $c(v)$ to black.

The algorithm above has elements of loops as well as solutions to the optimization problem. Therefore, we will set up a loop invariant which allows us to argue about it.

Fact 1.2. *Before we pull something from the queue, all gray vertices and in the queue and all vertices in the queue are gray. All vertices which are connected to s will eventually enter the queue and exit the queue.*

So the only lemma to prove is that the values $\ell(v)$ are the distances. In particular, we want to prove that at the end of the BFS claim, every vertex $v \in V$ has $\ell(v) = \delta(s, v)$.

Analysis. Consider the following definitions, which will allow us to analyze the algorithm. For any integer $d \geq 0$, we let V_d denote the set of vertices which are at distance d from s—the definition is completely independent of the breadth first search algorithm and only depends on the structure of the graph. The first claim we will show is that if we consider the order of vertices defined by when they are exiting the queue, we have the following claim, showing that vertices which exit the queue follow an order which respects d.

Claim 1.3. *For all $d \geq 0$, if u is any vertex in V_d and v is a vertex in V_{d+1}, then u is processed in the queue before v.*

Proof. We will prove the claim by induction on d.

- **Base Case**: The base case consists of $d = 0$ and V_0 consists of only s. It is placed in the queue at the beginning of the algorithm, so vertices in V_1 will clearly come after.
- **Inductive Hypothesis**: We assume that there is an integer d such that the vertices in V_{d-1} leave the queue before vertices in V_d leave the queue.

We now finish the proof by showing the inductive step. Suppose a vertex $w \in V_{d+1}$ is removed from the queue, but there is still some vertex $v \in V_d$ which is left in the queue. Then, we note that w was placed into the queue before v was placed into the queue. Furthermore, w was placed into the queue when we removed its parent $p(w)$ from the queue (which is in V_d); similarly, v was placed into the queue when we removed its parent $p(v)$ from the queue (which is in V_{d-1}). Since $p(w) \in V_d$ was removed from the queue before $p(v) \in V_{d-1}$ was removed from the queue, we obtain a contradiction to the inductive hypothesis. □

Lemma 1.4. *At the end of the algorithm, $\ell(v) = \delta(s, v)$ for all v.*

Proof. We will also prove this by induction by phrasing the lemma as showing that for all $v \in V_d$, we have $\ell(v) = d$. We prove this by induction on d.

- **Base Case**: When $d = 0$, this is correct by the initialization.
- **Inductive Hypothesis**: We assume that $\ell(v) = d$ for all $v \in V_d$.

Consider $v \in V_{d+1}$. Then, we set $\ell(v)$ when we place it into the queue. This value was placed into the queue when some element w which was connected to v was removed from the queue, and in addition, this was the first element which was placed into the queue among those which are connected to v. There are two cases:

1. Suppose $w \in V_d$. Then, $p(v)$ will become w, and $\ell(v) = \ell(w) + 1$. By induction $\ell(w) = d$.
2. Suppose $w \in V_{d+1} \cup V_{d+2}$. Then, we consider the shortest path from s to v and we let $u \in V_d$ denote the vertex on this path to s. Then, by the claim, u was removed from the queue before w and therefore, v would have been already in the queue. This means that we cannot have the case $w \in V_{d+1} \cup V_{d+2}$.

□

2 Depth First Search (DFS)

The depth first search algorithm proceeds by continuing to explore the graph by following a path. Similarly to breadth-first search, our problem is the following:

- **Input**: Our input is a graph $G = (V, E)$.
- **Output**: The goal is to explore every vertex and assign a start time and an end time to each vertex.

Importantly, depth-first search will not produce a tree, nor the distances for depth first search. We can think of the algorithm as a recursive algorithm which maintains, for each node v a color $c(v)$ which can also be white, gray, or black, and keeps track of a parent. The algorithm can be described as an algorithm which repeatedly calls a sub-routine DFS-Visit.

- **Initialization**: We first initialize every vertex to have color $c(v)$ set to white and the parent $p(v)$ to null, the current start time and end time is null.
- **Loop-Condition**: We will continue while there is a vertex v whose color $c(v)$ is set to white.
- **Loop**: We take the vertex which is colored white and we will color it gray. The moment we color a vertex gray, we set the start time. Then, we execute a sub-routine DFS-Visit(G, v).

The sub-routine DFS-Visit, when executed on a vertex v on a graph G, will first set the color $c(v)$ to be gray and set the start time. Then, it will check all of its vertices which are connected to v in G, and while there exists a neighbor u of v, whose color $c(u)$ is set to white, we set $p(u) = v$ and recursively execute DFS-Visit(G, u). Once all of these are done, we make the color $c(v)$ to black and set the finish time. Effectively, the start time and the finish time records (1) the time when it became gray, and (2) the time when it became black.

Claim 2.1. *The algorithm* DFS-Visit(G, v) *is only called once; because it is initially has color $c(v)$ set to white, and then set to gray, and then set to black. After executing* DFS-Visit(G, v)*, it will change from white to black, so it is only executed once.*

Suppose that we consider the interval of time and consider the a vertex v and look at the start and the end time of v. Then, we note that v has $c(v)$ being white before the start time, it becomes gray between the start and end time, and then becomes black after the end time. If we consider the time between the start and end time, and we call this the "active" interval of v. Then, we have

Claim 2.2. *For any two vertices u, v, either the active interval of one is contained in the other, or they are disjoint. In particular, they cannot be partially intersecting.*

The proof is simple. If a vertex u has its start time first, it becomes gray at that time. If v is set from white to gray before u is set to black, then we recursively execute DFS-Visit(G, v) within the call to DFS-Visit(G, u). This means that DFS-Visit(G, v) returns before DFS-Visit(G, u) returns.

3 Minimum Spanning Tree

We receive as input an (undirected) weighted graph $G = (V, E)$ with the weights $w\colon E \to \mathbb{R}_{\geq 0}$. Then, we have the task of:

- **Goal**: First, a spanning tree $T \subset E$ is a *sub-graph* of the original graph G (i.e., it contains a subset of the edges) which contains the vertices V and which is a tree (there are no cycles). Furthermore, each tree T may be assigned a total weight, given by the sum of the edge weights, and the goal is to find the minimum spanning tree T which minimizes the sum of edge weights.

The applications are building a broadcasting network, but also come up as a sub-routine of other algorithms. In addition, we will simplify the discussion significantly by *assuming that all edges are distinct.* The lecture will proceed with the following plan:

1. We will seek to find the minimum spanning tree of a graph, but first, we will try to understand structural properties of the true minimum spanning tree (even though we may now know how to find it).
2. Having structural properties about an optimal solution is useful algorithmically, because it lets you limit the type of algorithms that you may design in order to find the optimal solution.

3.1 Cut Structure of Graphs and Minimum Spanning Trees

Definition 3.1. *A cut S in a graph $G = (V, E)$ is a partition of the vertices V into two sets S and $\overline{S}$. The set of edges in the cut $E(S, \overline{S})$ consists of the edges (u, v) where $u \in S$ and $v \in \overline{S}$.*

Lemma 3.2. *Let G be a weighted and connected graph, and $(S, \overline{S})$ be a cut of G, and let e denote the lightest edge in the cut (with respect to the weight function), then, e lies in any minimum spanning tree.*

Proof. Suppose T is a minimum spanning tree without containing the lightest edge in the cut $(S, \overline{S})$. Then, we add the edge e to T and we note that this creates a cycle which crosses the cut $(S, \overline{S})$ at least *twice* (once for e, and at least one other edge). If we remove this other edge, the graph would still be connected because we still had a cycle, but the total cost has now decreased because we have swapped an edge for the lightest edge in the cut. □

The above lemma gives something which we may be able to use algorithmically, because the true minimum spanning tree will always contain the minimum edge in a cut. In our algorithm, if we ever encounter a cut and see its minimum edge, we can always include it into our potential minimum spanning tree T.

3.2 Prim's Algorithm

3.3 Kruskal's Algorithm

3.4 The Union-Find Data Structure

Last time we were talking about Kruskal's algorithm, and we left the implementation details unspecified. In particular, the key aspect of Kruskal's algorithm was to maintain a disjoint collection of connected components in a way that we can answer:

> Is vertex u connected to vertex v?

We will do this by keeping a data structure which maintains a disjoint collection of connected components, and can perform the following operations:

- **Find**: Given a vertex u, output a distinct pointer to the connected component (so that if we query u and v, we can determine if they are in the same connected component).
- **Union**: Given the pointers to two different connected components, we update to take the union of the connected component.

The data structure will maintain a rooted tree for each component, and the root of the tree will serve as the unique pointer which specifies that component. Thus, if we ever query "find" on two vertices which are in the same connected component, they will output the same tree. This means that the "find" operation is quite easy to implement:

- We keep walking up the tree to its parent.
- When we get to the root, we output the root.

Then, when we want to take a union, we first find the root of one component, then we find the root of another component, and we assign the parent of a root be the other root.

Note that the above data structure may run into a problem. If the tree becomes unbalanced and there is a long path, then it will take a while to find the root of the component.

Optimization 1: Union by Size We will be a bit more careful about how we take the union operation. In particular, we could keep track in each tree, the number of nodes, or the height of the tree. Then, when we take the union, we put the smaller tree below the larger tree.

Lemma 3.3. *Suppose that, whenever we link the root of x and the root of y, we make x a child of y if and only if the size of y is larger than the size of x. Then, the height of every component is always at most* $\log n$.

The above lemma follows from a simple proof by induction, but this means that the worst-case running time of taking a union and doing a find operation is at most $O(\log n)$. For the proof, inductively assume that we have two trees (rooted at x and y), and that the height h of any is always at least 2^h. Then, the height of the tree y becomes $h+1$ only when the heights of x and y are equal to h. If the heights are equal to at least h, then their size of x and y must both be at least 2^h, so the combined tree has at least 2^{h+1} nodes.

Then, we obtain the running time of $O(m \log n)$ of Kruskal's algorithm, and this is the time it takes to sort the edges by weight in the first place. However, we can even try to improve the running time of union-find. This means that if we assume that we read the edges in weighted order, then we can speed up the execution of Kruskal's algorithm.

Optimization 2: Find with Path Compression. The idea here is to compress the length of paths for the future. In particular, if we run a find of x, we will walk up the tree towards the root. When we do so, we may take advantage of this work in order to make all vertices in the path point directly to the root. Asymptotically, this doesn't add any running time to the "find" operation. The goal, however, is that if we ever run find on a path which we just compressed, we will find the root much quickly.

It turns out that the *amortized* running time becomes much better. There is a very tight analysis which shows that the amortized time become $\alpha(n)$, where α is the inverse Ackerman function (which grows extremely slowly—much much slower than $\log n$). For us, it will suffice to have a weaker bound.

Lemma 3.4. *Suppose one maintains a union-find data structure with both union by size, as well as path compression. Then, the amortized running time per operation is $O(\log^* n)$, where*

$$\log^* n = \textit{ number of times we must take } \log_2 n \textit{ to get down to } 1 \textit{ or lower.}$$

We note that $\log^* n$ is extremely small:

$$\begin{aligned} \log^* 2 &= 1 \\ \log^* 4 = \log^*(2^2) &= 2 \\ \log^* 16 = \log^*(2^{2^2}) &= 3 \\ \log^*(2^{2^{2^2}}) &= 4 \end{aligned}$$

We note that $\log^* n$ is still a growing function of n—so its not constant! In particular, there is a theorem that the union-find data structure *must* have an amortized running time whose time complexity must grow with n. The inverse Ackerman function is even slower!

4 Shortest Paths

The input to a shortest path problem is a weighted, *directed* graph $G = (V, E, w)$, where now the edges (u, v) have an order, so u points to v. The problem shows up all over the place: the graph may represent a road network (where the direction is important as some streets may be one-way), or the hyperlinks in the "internet graph." We now that in unweighted, undirected graphs, we can solve the problem with breadth-first search; however, we will now obtain the more general result of shortest paths on directed, weighted graphs. There are three variants:

- Given a source vertex s and target vertex t, find the shortest path between s and t.
- Given a source vertex s, find the shortest path between s and every vertex in V.
- Find the shortest paths from all vertices to all other vertices.

The above are a single-source, single-target shortest path, the single-source shortest path (where we are given one source), and then the all-pairs shortest path. It turns out that we do not know how, in the worst-case complexity, solve the single-source, single-target problem any faster than the single-source shortest path. Thus, we will focus on the single-source shortest path.

Single-Source Shortest Paths. We will first handle a technicality—should the shortest path always be a simple path (i.e., one without loops)? Well, if the edge weights are all positive, then taking a loop is always spending extra weight, so it is not the shortest. If there are negative edge weights, then either (1) there is a negative-weight cycle, and thus there is no shortest path (as going around the cycle always makes the path shorter), or (2) if there is a zero-weight cycle, in which case we can avoid taking the cycle. We summarize this by a claim:

Claim 4.1. *For any two vertices s and t, if there is a shortest path, then there is simple shortest path and one which has at most $n-1$ edges.*

It will thus be useful to distinguish between the length of a path, and the number of "hops" in a path. The length refers to the sum of the weights, and the hops refers to the number of edges in the path. The above claim tells us that it suffices to consider at most $n-1$ hops.

Bellman-Ford. The first algorithm which we will see is a dynamic programming algorithm called the "Bellman-Ford" algorithm, where Bellman is associated with the inventor of dynamic programming. Thus, we need to come up with the G-function for our dynamic programming formulation of shortest path, and thus an appropriate high level guide.

> **Top-Level Guide**: The shortest path between s and any vertex t is at most the length of the shortest path with at most $n-1$ hops, and the shortest path from s to t using at most h hops is at most the shortest, of all neighbors of t of at most $h-1$ hops.

We can now summarize the Bellman-Ford algorithm as iteratively finding the shortest paths with a bound on the number of hops that we may take. In particular, we may define

$$G(t,k) = \text{ length of shortest path from } s \text{ to } t \text{ with at most } k \text{ hops.}$$

We note that

$$G(t,k) = \min_{v \in N(t)} \left\{ G(v,k-1) + w(v,t), G(t,k-1) \right\},$$

where $N(t)$ denotes the neighborhood of t, i.e., the set of vertices v where (v,t) is an edge. The above would be a top-down approach, but the Bellman-Ford algorithm is generally described as a bottom-up algorithm.

1. We maintain the shortest path from s to every other vertex $t \in V$ by keeping a function $d(t)$ which specifies the length of the shortest path from s to t. Initially, we set $d(s) = 0$ and $d(t) = \infty$ for $t \neq s$.
2. We repeat $n - 1$ rounds, where in each round, we iterate through every edge (v, t) and we update
$$d(t) \leftarrow \min \{d(v) + w(v, t), d(t)\}.$$

Formally, we can prove that after iteration k, the values $d(t)$ are smaller than $G(t, k)$ (where we define $G(t, k)$ as infinity if there are no paths from s to t of at most k hops).

Lemma 4.2. *Suppose t is a vertex which contains a shortest path from s of k hops. Then, at the end of the k-th iteration, the $d(t)$ contains the length of the shortest path, and for iterations after, it will not take.*

Proof. The proof proceeds by induction on k, where the base case is when $k = 0$ and corresponds only to s. So, if we assume that it works up to the end of the k-th iteration, we consider a vertex t where there is a shortest path of k hops. This path goes by first connecting s to v with a path of k hops, and the final edge goes from v to t. By induction, by the end of the k-th round, $d(v)$ contains the length of the path from s to v and therefore, the $k + 1$-th round will eventually use edge (v, t), and then $d(t)$ will become the length of the shortest path from s to t. In addition, every time we consider an update, it corresponds to a path from s of that length, so by definition of the shortest path, the length will not decrease anymore. $\square$

Negative Weight Edges and Cycles Suppose that the graph has some negative weight edges, but it does not have negative weight cycles; in this case, Bellman-Ford works just fine—the only thing we used in the proof is that a shortest path exists. If there are negative weight edges but not negative weight cycles, shortest paths do exist. Suppose, however, that there are negative weight cycles which are reachable from s. Then, some shortest path is not defined. However, we can do a bit more work in order to flag whenever this is the case: we run an n-th iteration, if any of the values change, we know that the shortest path is not defined (by the lemma!).

Suppose there is a negative weight cycle which is reachable from s. Then, if the n-th iteration does update any $d(t)$, we have that the negative weight cycle containing edges (v, t) satisfies

$$d(t) \leq d(v) + w(v, t),$$

so if we sum across the cycle C, we obtain

$$\sum_{t \in C} d(t) \leq \sum_{v \in C} d(t) + \sum_{(v,t) \in C} w(v, t) \qquad \Longrightarrow 0 \leq \sum_{(v,t) \in C} w(v, t),$$

so the cycle is not negative weight.

4.1 Djikstra's Algorithm

Last time we spoke about the single-source shortest path problem, which receives a directed graph $G = (V, E)$ with a weight function $w \colon E \to \mathbb{R}$ and a source vertex s. Recall that we are trying to find:

$$\delta(s, v) = \text{ length of the shortest path from } s \text{ to } v.$$

We first make a very simple observation, that if we have an edge (u, v), then we are guaranteed that

$$\delta(s, v) \leq \delta(s, u) + w(u, v),$$

since the path from s to v which goes through u gives us an upper bound on $\delta(s, v)$. We may thus summarize the Bellman-Ford algorithm as updating:

$$\textbf{Bellman-Ford:} \qquad D(v) = \min\{D(v), D(u) + w(u, v)\},$$

and we performed these updates for each edge (u, v) for $n - 1$ iterations. Furthermore, we always have that $D \colon V \to \mathbb{R}$ is a potential distance function and corresponds to the distance of a particular path from s to v. We always have that $D(v)$ is the length of some path that we've found in the algorithm, so we have that, throughout the execution,

$$D(v) \geq \delta(s, v) \qquad : \qquad \forall v \in V.$$

When we update $D(v)$ in the Bellman-Ford update, we update the distance to $D(u) + w(u, v)$ by re-routing the previous path so that it goes through the vertex u to get to v.

A More Efficient Algorithm for Non-negative Edges. We will now study Djikstra's algorithm to find the single-source shortest path, where we will use a greedy approach to find the shortest paths (which you can interpret as a combination of BFS and Prim's algorithm). The algorithm only works when the weights $w \colon E \to \mathbb{R}_{\geq 0}$, and proceeds in the following way:

- We maintain a source s, and we will initialize to maintain $D(v)$ for all $v \in V$, as well as a set $S \subset V$ which is initially $S = \emptyset$. Intuitively, the set S will correspond to the "fully explored" vertices, and once a vertex v enters the set S, it has its value $D(v)$ fixed. Initially, only $D(s) = 0$ and the rest $D(v) = \infty$ for all $v \in V \setminus \{s\}$.
- At each iteration, we will find the vertex $v \in V \setminus S$ which has the current smallest $D(v)$ (among this set $V \setminus S$). This vertex $v \in V \setminus S$ is the vertex which has the best path from s which is currently unexplored. Once we find that vertex, we update:
  - First, we let $S \leftarrow S \cup \{v\}$.
  - Then, for every $u \in V \setminus S$ where $(v, u) \in E$, we let

    $$D(u) \leftarrow \min\{D(u), D(v) + w(u, v)\}.$$

We will prove the correctness of the algorithm by considering the following invariant:

Loop Invariant: For every $v \in S$, $D(v)$ contains the value of the shortest path from s to v.

Then, we show that the algorithm is correct by showing that the invariant holds at initialization (this is trivial), and that we may maintain it. Then, at termination, the algorithm has $S = V$ and we are done. So it suffices to consider the maintenance step.

Claim 4.3. *Suppose that the claim is true for a set S, and let v denote a vertex in $V \setminus S$ with smallest $D(v)$, then $D(v)$ is the length of the shortest path from s.*

Proof. Suppose that there is a shorter path to v. Then, this path leaves S, and consider the first time it leaves S. Then, if the first time it leaves S is to v, then we are done. Suppose otherwise, then it goes out from $u \in S$ to $u' \in V \setminus S$. Then, we have

$$\delta(s, v) = D(u) + w(u, u') + \delta(u', v) < D(v),$$

and note that, since $u \in S$, we must have $D(u') \leq D(u) + w(u, u')$ since the algorithm performed that operation when u was brought into S and $D(u)$ remained unchanged. Therefore, we have

$$D(u') \leq D(u) + w(u, u') \leq D(u) + w(u, u') + \delta(u', v) < D(v).$$

But this is a contradiction since v was chosen instead of u'. □

For the implementation details, note that we need to maintain $D(v)$ for all $v \in V \setminus S$, and be able to extract the minimum value, and then update the keys of all of these neighbors (while updating each edge at most once). Thus, the running time would become $O(m \log n)$. We note that, using Fibonacci heaps, we can get running time $O(m + n \log n)$.

5 All Pairs Shortest Path

As the name implies, we receive as input a directed graph $G = (V, E)$, and we are trying to find all of the shortest paths between all pairs of vertices. **(1)** If the edge weights are non-negative, we can apply Djikstra's algorithm for single-source shortest path, where we run the algorithm n times, once for each source—the running time becomes $O(nm \log n)$. **(2)** If some of the edge weights are negative, then we run Bellman-Ford for all n vertices and obtain $O(mn^2)$, and this could be as high as $O(n^4)$. We will now give a faster algorithm that can handle negative edge weights.

Floyd-Warshall Algorithm. The algorithm is a dynamic programming algorithm, where the idea is to consider the vertices in some arbitrary order. Then, at a particular iteration k, we consider a pair of vertices i, j, and we consider paths from i to j which are all fully contained in the vertices $1, \ldots, k$. In other words, we may maintain the G-function

$$G(i, j, k) = \text{ shortest path from } i \text{ to } j \text{ which only considers vertices from } \{1, \ldots, k\}.$$

Then, we have the initialization:

$$G(i,j,0) = \begin{cases} 0 & i = j \\ w(i,j) & (i,j) \in E \\ \infty & \text{o.w.} \end{cases}.$$

Then, we have the recurrence:

$$G(i,j,k+1) = \min\left\{G(i,j,k), G(i,k+1,k) + G(k+1,j,k)\right\},$$

and this immediately implies an algorithm which runs in time $O(n^3)$.

6 Network Flows

We start with a directed graph $G = (V, E)$, and we will specify a source vertex s and a sink vertex t. In your mind, you should think about algorithmic problems where we want to understand how to *route* or transport objects across a network. Then, the direction of the edge is important, as it specifies that we can send our object across that edge. So we will think about each of these edges as being a capacity $c \colon E \to \mathbb{R}_{\geq 0}$ which assigns to each edge.

Perhaps a useful example to keep in mind is the method for routing water in a sequence of pipes across a network. We have the source of water be at s, and it must make its way across a city to a particular house t. Then, the capacity has a natural interpretation in the amount of flow that can flow across a pipe (modeled by an edge), which is perhaps related to the diameter of the pipe. Formally, we have

- There is a directed graph $G = (V, E)$.
- There is a set of capacities $c \colon E \to \mathbb{R}_{\geq 0}$, and for this lecture, we will restrict ourselves to capacities always being *non-negative integers* $\mathbb{Z}_{\geq 0}$.
- Finally, there is a source s, and a sink t.

Algorithmic Question: How should we route as much *flow* as possible from s to t along the graph G, while satisfying the capacity constraints?

We note that the above problem also has the flavor of an optimization problem. There is a (yet unspecified) notion of a "flow" which sends from s to t, and we are trying to send as much "flow" as we can. Thus, up to specifying the solution space $\mathcal{S}$ of "flows" and the constraint set $\mathcal{K}$, we seek to maximize the amount of flow we send from s to t.

The Solution Space. So, what is a *flow*? Well, intuitively, we want to be able to route stuff from s to t, so we need a way to specify our notion of routing. For us, we will specify a flow by a function $f \colon E \to \mathbb{R}_{\geq 0}$ which maps edges to non-negative real values, where we will let

$$f(e) = \text{"flow" which goes across edge } e \in E.$$

Now, an arbitrary function of the flow doesn't really make sense in terms of routing stuff in G only makes sense if we satisfy certain "stuff conservation" constraints. In particular, we should have for any vertex v, if that vertex is not the source s or the sink t, whatever comes into v should also exit v. In particular,

$$\textbf{Flow Conservation Constraint:} \qquad \forall v \in V \setminus \{s,t\}, \qquad \sum_{(u,v)\in E} f(u,v) = \sum_{(v,u)\in E} f(v,u).$$

Then, the constraints imposed by $\mathcal{K}$ will say that the flow which we send never is above our capacity constraints. In particular, for a flow $f \colon E \to \mathbb{R}_{\geq 0}$, we say that the edge $e \in E$ does not violate the capacity constraint if

$$\textbf{Capacity Constraint:} \qquad f(e) \leq c(e).$$

Therefore, we can now define our solution space $\mathcal{S}$ of flows and the constraint set $\mathcal{K}$ to be

$$\mathcal{K} = \{f \colon E \to \mathbb{R}_{\geq 0} : \text{ satisfies flow conservation and capacity constraints } \}.$$

Therefore, we will try to solve optimization problems whose feasible set of solutions are flows f which satisfy our constraints. We are now left with the final step of defining our optimization problem, which involves specifying the objective function $g \colon \mathcal{K} \to \mathbb{R}$ which measures the quality of a flow (or that which we aim to optimize). The max-flow problem asks to maximize the flow that we can send from the source s to the vertex t:

$$\textbf{Max-Flow Problem:} \qquad \max_{f\in\mathcal{K}} \sum_{(s,u)\in E} f(s,u).$$

Pushing flows along paths. Suppose that we start with a directed graph $G = (V, E)$ with capacities $c \colon E \to \mathbb{Z}_{\geq 0}$ and source vertex s and sink vertex t. Then, let's try to send *something* from s to t: if we find a path from s to t (for example, by using BFS or DFS), then we can send flow along the path, where the amount of flow that we can send is at most the capacity of the lowest capacity edge on that path. In particular, we can "push" flow along the path while satisfying the conservation and capacity constraints, where we are limited by the smallest capacity value on that edge.

Definition 6.1. *We say that an edge e is* saturated *by the flow f if $f(e) = c(e)$.*

Note that this increases the flow we sent from s to t, and therefore, our max flow is certainly above the value of the flow we are sending. This gives rise to a natural idea for a greedy algorithm:

> **Greedy Algorithm:** At every step, find any path with edges which have capacity, and send the flow needed to saturate some edge along the found path.

Does this algorithm work? The answer is no, and the reason is that we did not specify which path we are going to take, and therefore could choose a path such that we saturate a sequence of edges across a cut S, where $s \in S$ and $t \notin S$.

Residual Graphs and Possibly Changing Your Mind. Suppose that we have send flow some flow from s to t, and we have currently formalized it by $f\colon E \to \mathbb{R}_{\geq 0}$, we could consider the graph which models sending flow *back* from where it came from if I had send some flow along an edge. In particular, suppose we consider the graph G_f such that:

- We will let the vertex set V_f be the same as the original vertex set V, so we keep the set V.
- The edges E_f change slightly and there are two types of edges:
  - We will have an edge $(u, v) \in E_f$ if there is an edge $(u, v) \in E$. We will then set the capacity $c_f(u, v) = c(u, v) - f(u, v)$. This represents the "residual capacity" on the edge (u, v).
  - On the other hand, we can also consider the *back edge* (v, u) which encodes us "changing our mind," and sending flow back. Here, we set $(v, u) \in E_f$ be the edge if $(u, v) \in E$ and $f(u, v) \geq 0$. We then let $c_f(v, u) = f(u, v)$.

A First Algorithm via Augmenting Paths. We can now specify the first algorithm for finding a maximum flow on the directed graph $G = (V, E)$, which will be a minor modification to our first greedy approach.

> **Ford-Fulkerson:** Start with f being all-0's, and at every step, find any path *on the residual graph* G_f, and send the flow needed to saturate some edge in G_f along the found path. Update the flow by adding the flow along the found path.

We can now prove the main lemma.

Lemma 6.2. *Suppose $G = (V, E)$ is a directed graph with capacities $c\colon E \to \mathbb{Z}_{\geq 1}$ which are integers (and note that if $c(e) = 0$, we can remove e from E). Then, if $f\colon E \to \mathbb{R}_{\geq 0}$ is a flow found by the Ford-Fulkerson algorithm, there are two cases:*

- *If there is an s to t path in G_f, the minimum capacity edge along that path is at least 1. Thus, when we add the path, the current value of the flow increases by at least 1.*
- *If there is no path from s to t in G_f, then the current f is the maximum flow.*

Proof. The first item of the proof is true because we are always adding and subtracting integers, which means that non-zero values have their minimum being at least 1. The second is the more subtle part. Suppose that f^* represents the true maximum flow on G. Consider the flow $f^* - f$ (which, whenever $f^* - f$ is negative, we interpret as some flow on the back-edge).

- First, we claim that $f^* - f$ is feasible in G_f. If $f(e) \geq 0$, then $f^*(e) - f(e) \leq c(e) - f(e) = c_f(e)$. If $f(e) = 0$, then $f^*(e) \leq c(e) = c_f(e)$.
- Second, we have $f^* - f$ sends flow from s to t.

Therefore, there exists a path from s to t in G_f. $\square$

The above lemma implies the following theorem.

Theorem 1 (Time Complexity of Ford-Fulkerson)**.** *Suppose* $G = (V, E)$*,* $c\colon E \to \mathbb{Z}_{\geq 0}$ *, and* $s, t \in V$ *denote an input to the max-flow problem. Then, if* F *is the value of the maximum flow, then the Ford-Fulkerson algorithm takes time* $O((n+m)F)$*, since we must run at most* F *many BFS/DFS executions.*