University of Washington

Math 336 Term Paper

Graph Theory: Network Flow

Author:
Elliott Brossard

Adviser:
Dr. James Morrow

3 June 2010

Contents

1 Introduction

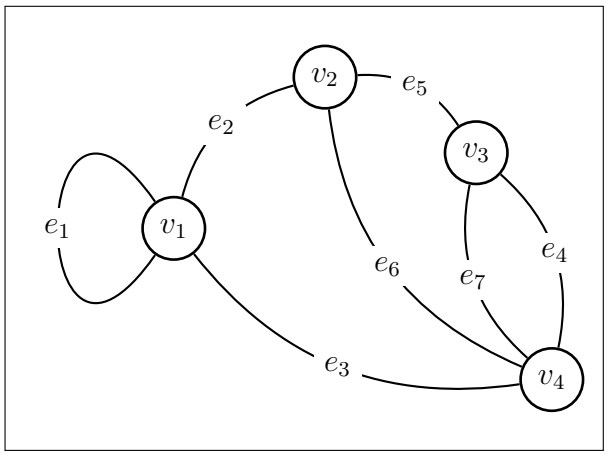
An important study in the field of computer science is the analysis of networks. Internet service providers (ISPs), cell-phone companies, search engines, e-commerce sites, and a variety of other businesses receive, process, store, and transmit gigabytes, terabytes, or even petabytes of data each day. When a user initiates a connection to one of these services, he sends data across a wired or wireless network to a router, modem, server, cell tower, or perhaps some other device that in turn forwards the information to another router, modem, etc. and so forth until it reaches its destination. For any given source, there are often many possible paths along which the data could travel before reaching its intended recipient. When I initiate a connection to Google from my laptop here at the University of Washington, for example, packets of data first travel from my wireless card to my router, then from my router to a hub in the underworkings of McCarty Hall, then to the central servers of the University of Washington, which transmit them along some path unknown to me until they finally reach one of Google's many servers 40 milliseconds later.

From any one source to an intended destination, however, there are often many different routes that data can take. Under light traffic, the University of Washington might distribute the task of sending its Internet users' packets through just two or three servers, whereas under heavy traffic it might use twice or three times as many. By spreading out the traffic across multiple servers, the University of Washington ensures that no one server will bog down with overuse and slow down the connection speeds of everyone using it.

The idea that there are many possible paths between a source and a destination in a network gives rise to some interesting questions. Specifically, if the connection speed between every two interlinked components is known, is it possible to determine the fastest possible route between the source and destination? Supposing that all of the traffic between two components was to follow a single path, however, how would this affect the performance of the individual components and connections along the route? As hinted above, the optimum solution for the fastest possible transmission of data often involves spreading out traffic to other components, even though one component or connection might be much faster than all the others. In this paper, we seek to build a framework in which we can precisely state these questions, and one that will allow us to provide satisfactory answers.

2 Terminology

The study of networks is often abstracted to the study of graph theory, which provides many useful ways of describing and analyzing interconnected components. To start our discussion of graph theory—and through it, networks—we will first begin with some terminology. First of all, we define a *graph* $G = (V, E)$ to be a set of *vertices* $V = \{v_1, v_2, \ldots, v_m\}$ and a set of *edges* $E = \{e_1, e_2, \ldots, e_n\}$. An edge is a connection between one or two vertices in a graph; we express an edge e_i as an unordered pair (v_j, v_k), where $v_j, v_k \in V$. If $j = k$, then e_i is called a *loop*. To illustrate these concepts, consider the following graph:



If we let G denote this graph, then $G = (V, E)$, where $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$. We can express the edges as $e_1 = (v_1, v_1)$, $e_2 = (v_1, v_2)$, $e_3 = (v_1, v_4)$, $e_4 = (v_2, v_4)$, $e_5 = (v_2, v_3)$, and $e_6 = (v_1, v_4)$. Note that e_1 is a loop, since it connects v_1 to itself.

This type of graph is also known as an *undirected graph*, since its edges do not have a direction. A *directed graph*, however, is one in which edges do have direction, and we express an edge e as an ordered pair (v_1, v_2). Remark that in an undirected graph, we have $(v_1, v_2) = (v_2, v_1)$, since edges are unordered pairs.

Sometimes it is convenient to think of the edges of a graph as having *weights*, or a certain cost associated with moving from one vertex to another along an edge. If the cost of an edge $e = (v_1, v_2)$ is c, then we write $\mathrm{w}(e) = \mathrm{w}(v_1, v_2) = c$. Graphs whose edges have weights are also known as *weighted graphs*. We define a *path* between two vertices v_1 and v_n to be an ordered tuple of vertices $(v_1, v_2, \ldots, v_n)$, where (v_j, v_{j+1}) is an edge of the graph for each $1 \leq j \leq n-1$. Note that in a directed graph, if the path $(v_1, v_2, \ldots, v_n)$ connects the two vertices v_1 and v_n, it is not necessarily the case that $(v_n, v_{n-1}, \ldots, v_1)$ is a path connecting them as well, since $(v_i, v_{i+1}) \neq (v_{i+1}, v_i)$. Two vertices v_1, v_2 are said to be *path-connected* if there is a path from v_1 to v_2. The total cost of a path is the sum of the costs of the edges, so $C = \sum_{j=1}^{n-1} \mathrm{w}(v_j, v_{j+1})$ is the cost of the path from v_1 to v_n.

3 Shortest Path Problem

A commonly occurring problem involving weighted graphs, both directed and undirected, is to find the minimum cost necessary to move from one vertex to another; that is, to find the shortest path between them. The most well-known algorithm for determining this path is **Dijkstra's Algorithm**, presented by Edsger Dijkstra in 1959, which actually solves the more general problem of finding the shortest paths from a given vertex to all other vertices under the restriction that edges have non-negative weights. In contrast, the A* Search Algorithm finds the shortest path between *any two* given vertices in a weighted graph with non-negative edge weights, and Ford's Algorithm (sometimes credited as the Bellman-Ford Algorithm) finds the shortest path from a given vertex to all other vertices in a weighted graph without restriction on the sign of the edge weights. For the purpose of network flow, we are concerned only with graphs that have non-negative edge weights, and prefer a solution that solves the more general problem of shortest paths from one vertex to all other vertices.

For further reading, however, the reader may refer to [2] for an exploration of the A* Search Algorithm or [4, p. 15] or [1, sec. 24.1] for an exploration of the Bellman-Ford Algorithm.

Before we present the steps and proof of Dijkstra's Algorithm, we will first formulate a precise statement of the problem we are attempting to solve. Let $G = (V, E)$ be a weighted graph with the set of vertices V and set of edges E, with each $e \in E$ satisfying $\mathrm{w}(e) \geq 0$, and let $v_1, v_2 \in V$ such that v_1 and v_2 are path-connected. We will assume that G is directed; if it is undirected, then consider the edge (u, v) to be the same as the edge (v, u). Which path $P = (w_1, w_2, \ldots, w_n)$ for $w_1 = v_1$, $w_n = v_2$ minimizes the sum $C = \sum_{i=1}^{n-1} \mathrm{w}(w_i, w_{i+1})$?

3.1 Dijkstra's Algorithm

Let $G = (V, E)$ be a directed weighted graph with the set of vertices V and the set of edges E with non-negative weights as above, and let $v_1, v_2 \in V$ such that v_1 and v_2 are path-connected. We claim that the path P produced by the below algorithm has the lowest total cost C of any possible path from v_1 to v_2.

1. Assign a distance value to every vertex. Where d denotes distance, set $\mathrm{d}(v_1) = 0$ and $\mathrm{d}(v) = \infty$ for all other $v \in V$.
2. Mark each vertex as "unknown", and set the "current vertex" v_c to be v_1.
3. Mark the current vertex v_c as known.
4. Let $N_c = \{v \in V \mid (v_c, v) \in E \text{ and } v \text{ is unknown}\}$ be the set of unknown neighbors of v_c. For each $v \in N_c$, let $\mathrm{d}(v) = \min\{\mathrm{d}(v), \mathrm{d}(v_c) + \mathrm{w}(v_c, v)\}$. If the value of $\mathrm{d}(v)$ changes, then set $\mathrm{p}(v) = v_c$. We will use $\mathrm{p}(v)$ to denote the vertex through which this one was accessed.
5. Set $v_c = v_d$, where $v_d \in N_c$ has the property that $\mathrm{d}(v_d) = \min_{v \in V}\{\mathrm{d}(v)\}$. If $v_d = v_2$, then proceed to the next step; otherwise, return to step 3.
6. Let P be an ordered tuple of vertices currently consisting of the single vertex v_2, and let $v_c = v_2$.
7. Let $N_c = \{v \in V \mid (v, v_c) \in E\}$, and let $v_d \in N_c$ such that for all $v \in N_c$, $\mathrm{d}(v_d) \leq \mathrm{d}(v)$. Insert v_d at the front of P.
8. If $v_d \neq v_1$, then set $v_c = v_d$ and return to step 7. If $v_d = v_1$ then we are done; P is the lowest-cost path from v_1 to v_2.

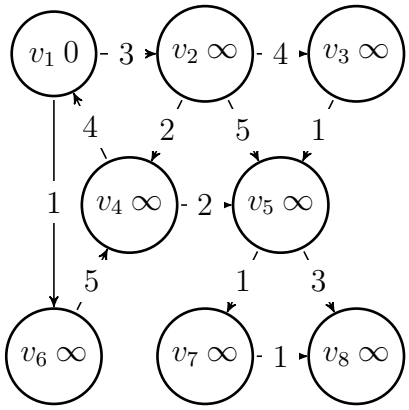
Note that at step 5 of the algorithm, replacing the check for $v_c = v_2$ with the condition that all vertices are known would allow the algorithm to continue until the shortest distance from v_1 to *any* any vertex is "known". For this to be guaranteed to work, however, we would need to add the restriction that there is at least one path from v_1 to all other vertices, since otherwise it would fail when there is no path from a known node to any of the unknown nodes. As for whether terminating the algorithm when v_2 is known is significantly faster on average than allowing it to run until all vertices are known, Thomas Cormen [1, p. 492] writes that "no algorithms for [the single-pair shortest path] problem are known that run asymptotically faster than the best single-source algorithms in the worst case."

We can also express Dijkstra's Algorithm in pseudocode as follows. Some of the details of this code borrow from [6, sec. 9.3].

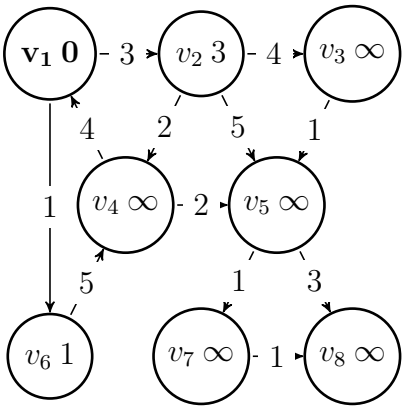
```
Path dijkstra(Graph g, Vertex start, Vertex end) {
  for each Vertex v in g {
    v.dist = INFINITY;
    v.known = false;
  }

  start.dist = 0;
  start.known = true;

  while(true) {
    v = unknown Vertex with smallest distance

    // if the shortest distance to the end vertex is known, exit the loop
    if(v == end)
      break;

    v.known = true;

    for each Vertex w where (v, w) is an edge {
      if(v.dist + Cost(v, w) < w.dist) {
        // update the distance to w
        w.dist = v.dist + Cost(v, w);

        // remember that we traveled through v to get to w
        w.path = v;
      }
    }
  }

  // the shortest distance to the end vertex is known, so iteratively construct the path
  // from end to start
  Path p = new Path;
  p.insertAtFront(end);

  Vertex current = end;
  while(current != start) {
    // move to the next vertex on the shortest path to the starting vertex
    current = current.path;
    p.insertAtFront(current);
  }

  return p;
}
```

3.2 Example Using Dijkstra's Algorithm

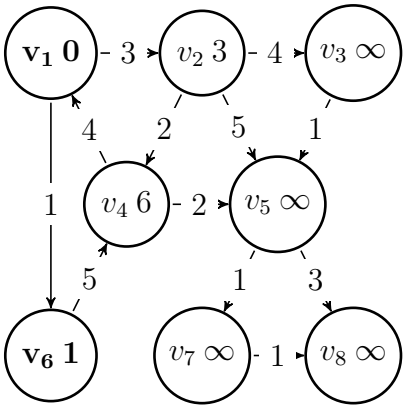
Here we will step through an example application of Dijkstra's Algorithm. In this example, we would like to find the shortest path between the vertices v_1 and v_8 in the pictured graph. To show the process of the algorithm, we will use the value next to the name of each vertex to denote its current distance d, and we will mark a vertex as known by displaying its label in bold. For the sake of space, the value of p(v) is not shown. As each new vertex is marked known, the distances of its neighboring vertices are updated to reflect the new possible path to them.



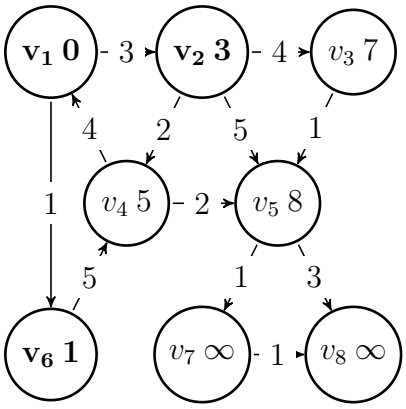
(a) Set up distances



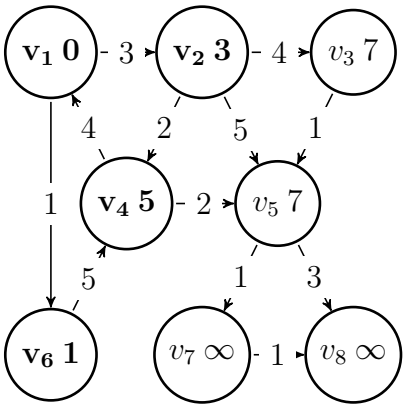
(b) v_1 is marked as known



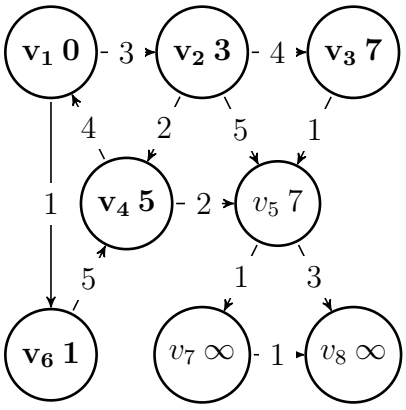
(c) v_6 is marked as known



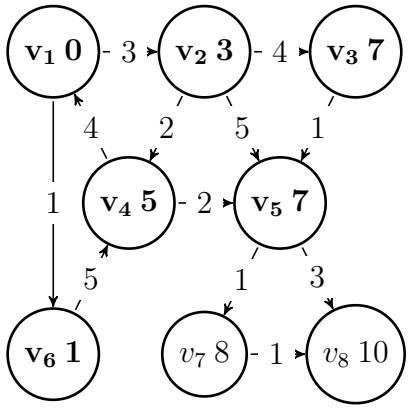
(d) v_2 is marked as known



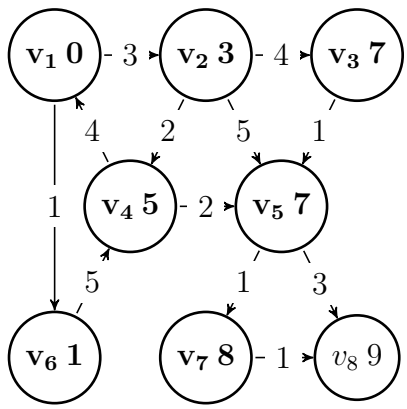
(e) v_4 is marked as known



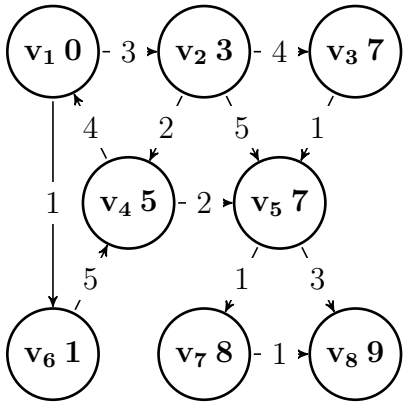
(f) v_3 is marked as known



(g) v_5 is marked as known



(h) v_7 is marked as known



(i) v_8 is marked as known

At this point, the algorithm would trace the shortest path from v_8 back to v_1, which in this case is $(v_1, v_2, v_4, v_5, v_7, v_9)$. As it turned out, the algorithm marked that every other vertex was "known" before it marked v_8 as known, so the algorithm explored every vertex possible before proceeding. If the weight of edge (v_5, v_8) were exchanged with the weight of (v_7, v_8), though, the algorithm would have marked v_8 as known before v_7 and hence proceeded to constructing the shortest path without actually exploring every node.

3.3 The Correctness of Dijkstra's Algorithm

We know turn our attention to a proof that Dijkstra's Algorithm will succeed in finding the shortest path in general. The idea for the following proof is similar to that presented in [1, sec. 24.3]. To aid in the development of this proof, we will first define the function $\delta : V \times V \to \mathbb{Z}$ to be the minimum-cost path from its first parameter to the second (0 if the two parameters are the same vertex), or ∞ if no path exists. We will also define Relax(u, v) to be the operation of setting $\mathrm{d}(v) = \min\{\mathrm{d}(v), \mathrm{d}(u) + \mathrm{w}(u, v)\}$ and setting $\mathrm{p}(v) = u$ if $\mathrm{d}(v)$ changed. What we would like to show is that for a starting vertex v_1 and an ending vertex v_2, at the completion of Dijkstra's algorithm we have $\mathrm{d}(v_2) = \delta(v_1, v_2)$. To do this, we will first prove some lemmas concerning shortest paths and the relaxation of edges.

3.3.1 Lemma (Triangle Inequality for Graphs)

Let $G(V, E)$ be a directed, weighted graph with non-negative edge weights, and suppose that $v_1 \in V$ is the starting vertex of G, where there is at least one path from v_1 to any other vertex. Then for all edges $(u, v) \in E$, $\delta(v_1, v) \leq \delta(v_1, u) + \mathrm{w}(u, v)$.

Proof: Let $v \in V$. Since there is path from v_1 to v, there is a shortest path P from v_1 to v, the total cost of which is less than or equal to any other path from v_1 to v. So if $(u, v) \in E$, then the cost of any path from v_1 to u and then from u to v is no less than the cost of P, and hence the shortest path from v_1 to v that passes through u has at least as much cost as P. In particular, $\delta(v_1, v) \leq \delta(v_1, u) + \mathrm{w}(u, v)$, which is what we wanted to show.

3.3.2 Lemma (Upper-Bound Property)

Let $G(V, E)$ be a directed, weighted graph with non-negative edge weights, and suppose that $v_1 \in V$ is the starting vertex of G, where there is at least one path from v_1 to any other vertex. Initialize $\mathrm{d}(v_1) = 0$, $\mathrm{d}(v) = \infty$ for $v \neq v_1$. Then $\mathrm{d}(v) \geq \delta(v_1, v)$ for all $v \in V$, and this property holds even if we apply any sequence of operations $\mathrm{Relax}(v_j, v_k)$ to $(v_j, v_k) \in E$. Furthermore, if $\mathrm{d}(v) = \delta(v_1, v)$ at some point in this sequence, then $\mathrm{d}(v) = \delta(v_1, v)$ after each subsequent operation.

Proof: We will proceed by induction to show that the statement $\mathrm{d}(v) \geq \delta(v_1, v)$ for all vertices $v \in V$ holds after each operation $\mathrm{Relax}(v_j, v_k)$ in the sequence.

Base case: Before any edges are relaxed, we have $\mathrm{d}(v_1) = 0 \geq \delta(v_1, v_1) = 0$. $\mathrm{d}(v) = \infty \geq \delta(v_1, v)$, since there is a path from v_1 to v, so the statement holds.

Inductive step: Now assume that the statement holds for all operations $\mathrm{Relax}(v_i, v_j)$ in the sequence prior to the operation $\mathrm{Relax}(v_k, v_l)$. We know that $\mathrm{d}(v) \geq \delta(v_1, v)$ for all $v \in V$ prior to this operation by the inductive hypothesis, and since $\mathrm{Relax}(v_k, v_l)$ can only change $\mathrm{d}(v_l)$, we need only to show that $\mathrm{d}(v_l) \geq \delta(v_1, v_l)$. Since $\mathrm{d}(v_l) \geq \delta(v_1, v_l)$ before the operation $\mathrm{Relax}(v_k, v_l)$ by the inductive hypothesis, suppose that $\mathrm{Relax}(v_k, v_l)$ changes the value of $\mathrm{d}(v_l)$. Then

$$\begin{aligned}
\mathrm{d}(v_l) &= \mathrm{d}(v_k) + \mathrm{w}(v_k, v_l) \\
&\geq \delta(v_1, v_k) + \mathrm{w}(v_k, v_l) && \text{(By the inductive hypothesis)} \\
&\geq \delta(v_1, v_l), && \text{(By the Triangle Inequality for Graphs)}
\end{aligned}$$

which completes the inductive proof.

Furthermore, at any step in the sequence of relaxations, if $\mathrm{d}(v) = \delta(v_1, v)$, then $\mathrm{d}(v)$ will remain unchanged past that point. This follows from the fact that $\mathrm{Relax}(v_j, v_k)$ sets $\mathrm{d}(v_k)$ to the minimum of $\mathrm{d}(v_k)$ and $\mathrm{d}(v_j) + \mathrm{w}(v_j, v_k)$, neither of which can be less than $\delta(v_1, v_k)$ by the definition of δ, and relaxing the edge cannot increase $\mathrm{d}(v_k)$.

3.3.3 Lemma

Let $G(V, E)$ be a directed, weighted graph with non-negative edge weights, and let $(u, v) \in E$. Then immediately after the operation $\mathrm{Relax}(u, v)$, we have $\mathrm{d}(v) \leq \mathrm{d}(u) + \mathrm{w}(u, v)$.

Proof: Suppose that just before relaxing the edge (u, v), $\mathrm{d}(v) > \mathrm{d}(u) + \mathrm{w}(u, v)$. Then $\mathrm{d}(v) = \mathrm{d}(u) + \mathrm{w}(u, v)$ afterward, so the desired inequality holds. Now suppose that $\mathrm{d}(v) \leq \mathrm{d}(u) + \mathrm{w}(u, v)$ just before the edge (u, v) is relaxed. Then neither $\mathrm{d}(u)$ nor $\mathrm{d}(v)$ changes during the operation, so $\mathrm{d}(v) \leq \mathrm{d}(u) + \mathrm{w}(u, v)$ afterward as well, which is what we wanted to show.

3.3.4 Lemma

Let $G(V, E)$ be a directed, weighted graph with non-negative edge weights. Let $P = (v_1, v_2, \ldots, v_n)$ be a shortest path from v_1 to v_n, and denote the subpath $(v_i, v_{i+1}, \ldots, v_j)$ as P_{ij}. Then P_{ij} is the shortest path from v_i to v_j.

Proof: Notice that $\mathrm{w}(P) = \mathrm{w}(P_{1i}) + \mathrm{w}(P_{ij}) + \mathrm{w}(P_{jn})$. Assume that there is some path P'_{ij} from v_i to v_j with weight $\mathrm{w}(P'_{ij}) < \mathrm{w}(P_{ij})$. Then $P_{1i} \cup P'_{ij} \cup P_{jn}$ is a path from v_1 to v_n with weight $\mathrm{w}(P_{1i}) + \mathrm{w}(P'_{ij}) + \mathrm{w}(P_{jn}) < \mathrm{w}(P)$. But this is a contradiction, since P is the shortest path from v_1 to v_n, so P_{ij} is the shortest path from v_i to v_j as we wanted to show.

3.3.5 Lemma (Convergence Property)

Let $G(V, E)$ be a directed, weighted graph with non-negative edge weights. Suppose that $v_1 \in V$ is the starting vertex of G, and let P be the shortest path from v_1 to v, where u precedes v on the path. Further suppose that we initialize $\mathrm{d}(v_1) = 0$, $\mathrm{d}(v) = \infty$ for $v \neq v_1$ and then apply $\mathrm{Relax}(v_j, v_k)$ to a subset of edges $(v_j, v_k) \in E$ containing (u, v). Then if $\mathrm{d}(u) = \delta(v_1, u)$ at some point before this series of operations, $\mathrm{d}(v) = \delta(v_1, v)$ after the series of operations.

Proof: By the Upper-Bound Property, if $\mathrm{d}(u) = \delta(v_1, u)$ at some point before the operation $\mathrm{Relax}(u, v)$, then the same equality holds after all subsequent relax operations. After the edge (u, v) is relaxed, we have

$$\begin{aligned} \mathrm{d}(v) &\leq \mathrm{d}(u) + \mathrm{w}(u, v) && \text{(By Lemma 3.3.3)} \\ &= \delta(v_1, u) + \mathrm{w}(u, v) \\ &= \delta(v_1, v) && \text{(By Lemma 3.3.4)} \end{aligned}$$

3.3.6 Theorem (Correctness of Dijkstra's Algorithm)

We now have the means to prove the validity of the Dijkstra's Algorithm. Let G = (V, E) be a graph, either undirected or directed, with non-negative edge weights, and let $v_1, v_2 \in V$ such that v_1 and v_2 are path-connected. Then the path P produced by Dijkstra's Algorithm has the lowest total cost of any path connecting v_1 and v_2.

Proof: We proceed by examining the state of the known vertices denoted S, and show that each iteration of Dijkstra's Algorithm, $\mathrm{d}(v) = \delta(v_1, v)$ for all $v \in S$. For the sake of notation, we will assume that v_1 is path-connected to every other $v \in V$, since Dijkstra's Algorithm only operates on path-connected vertices anyway. Note that at the beginning of Dijkstra's Algorithm, the first vertex to be marked as known is v_1, and $\mathrm{d}(v_1)$ is set to 0, so $\mathrm{d}(v_1) = \delta(v_1, v_1) = 0$ as we want.

We now want to show that in each proceeding iteration of Dijkstra's Algorithm, if the unknown vertex $v \in V - S$ is marked as known, then $\mathrm{d}(v) = \delta(v_1, v)$. In order to do this, we

will suppose that this is not true and establish a contradiction, so let $u \in V$ be the first vertex added to S for which $\mathrm{d}(u) \neq \delta(v_1, u)$. We already know that $\mathrm{d}(v_1) = \delta(v_1, v_1) = 0$, so $u \neq v_1$, and hence $S \neq \emptyset$. v_1 and u are path-connected, so there must be a shortest path P from v_1 to u. Before u was marked as known, P connected the vertex $v_1 \in S$ to the vertex $u \in V - S$, so we have a vertex y along P such that $y \in V - S$ and a predecessor $x \in S$. Notice that $x \in S$, so since u was chosen as the first vertex with $\mathrm{d}(u) \neq \delta(v_1, u)$, we know that $\mathrm{d}(x) = \delta(v_1, x)$ when x was added to S. We also applied $\mathrm{Relax}(x, y)$ when x was added to S, so by the Convergence Property, $\mathrm{d}(y) = \delta(v_1, y)$.

Because y occurs on the path P and all of the edges of E are non-negative, we have that $\delta(v_1, y) \leq \delta(v_1, u)$ and hence $\mathrm{d}(y) = \delta(v_1, y) \leq \delta(v_1, u) \leq \mathrm{d}(u)$ by the Upper-Bound Property. Since both u and y were in $V - S$ when u was chosen, however, we have that $\mathrm{d}(u) \leq \mathrm{d}(y)$, so in fact $\mathrm{d}(y) = \delta(v_1, y) = \delta(v_1, y) = \mathrm{d}(u)$. But then $\delta(v_1, u) = \mathrm{d}(u)$, which is a contradiction of how u was chosen. So $\mathrm{d}(u) = \delta(v_1, u)$ when u is added to S, and hence $\mathrm{d}(v) = \delta(v_1, v)$ for all $v \in S$ at each step of the algorithm. In particular, $\mathrm{d}(v_2) = \delta(v_1, v_2)$, so the path that Dijkstra's Algorithm constructs from v_1 to v_2 is in fact the shortest possible path.

4 Maximum Flow Problem

With Dijkstra's Algorithm, we have solved one of the major problems that we posed at the beginning of this paper. We have shown that given a graph with a certain cost associated to each edge, we can find the path of lowest cost from one vertex to any other vertex, or more specifically, that given a network with a certain speed associated to each connection between components, we can find the fastest possible route from one component to another.

Sending all of a network's traffic over a single path, however, can cause performance degradation for all the connections that use it. For each connection between components, there is a limit to how much traffic can pass over it at once without negatively impacting its speed or reliability, which in computer science is known as *channel capacity*. In this section, we will explore the problem of maximizing the flow of data over a network from one component to another given a channel capacity for each connection. More generally, this is known as the **maximum flow** problem for flow networks.

4.1 Terminology

We will begin the discussion of maximum flow with some terminology. A *flow network* is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a *capacity* $c(u, v) \geq 0$. For edges $(u, v) \notin E$, we define $c(u, v) = 0$. As opposed to start and end vertices, we instead refer to the origin and destination vertices in a flow network as the *source* and the *sink*, respectively. We will assume that in a flow network $G = (V, E)$ with source s and sink t, for any vertex $v \in V$ there is a path from s to t that passes through v.

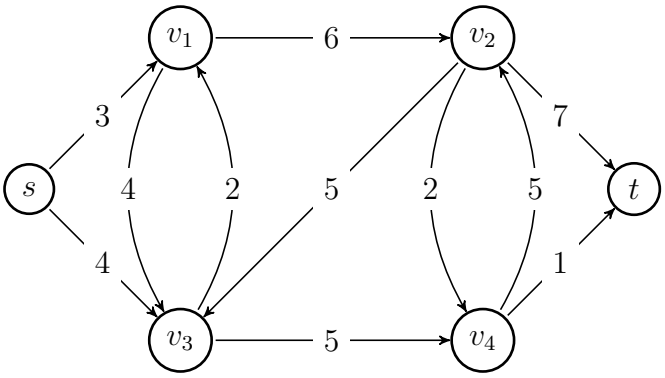
A $flow$ in a flow network $G = (V, E)$ with source s and sink t is a function $f : V \times V \to \mathbb{R}$ that satisfies the following properties:

1. **Capacity Constraint**: For all $u, v \in V$, $f(u, v) \leq c(u, v)$.
2. **Skew Symmetry**: For all $u, v \in V$, $f(u, v) = -f(v, u)$.

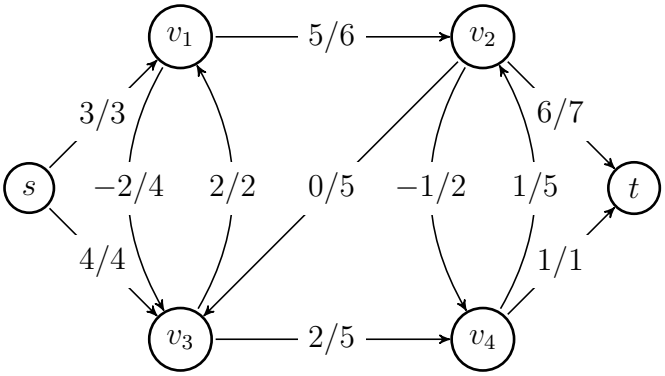
3. **Flow Conservation**: If $u \in V$ and $u \neq s$, $u \neq t$, then $\sum_{v \in V} f(u, v) = 0$.

We say that $f(u, v)$ is the flow from vertex u to vertex v. For two sets of vertices X and Y and a flow function f, we define $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$ and $c(X, Y) = \sum_{x \in X} \sum_{y \in Y} c(x, y)$. The *value* of a flow f, not to be confused with absolute value or norm, is denoted $|f|$ and defined as $|f| = \sum_{v \in V} f(s, v)$; i.e. the total flow out of the source.

The following is an example of a flow network that illustrates these concepts:



(a) A flow network $G = (V, E)$ where each edge $(u, v) \in E$ is labeled with $c(u, v)$.

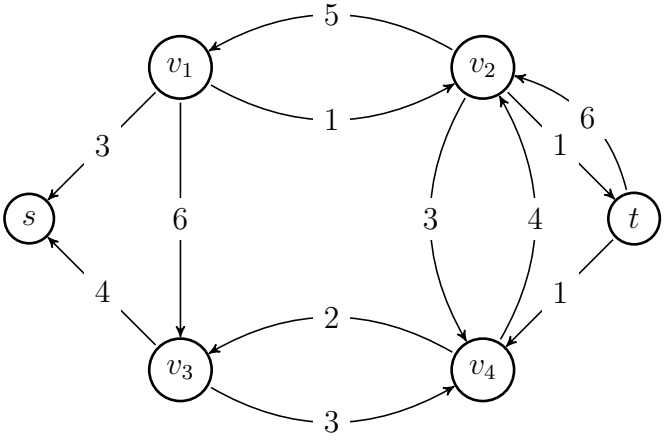


(b) The same flow network G, where each edge $(u, v) \in E$ is labeled with $f(u, v)/c(u, v)$ for a flow function f.

In the first figure, we have constructed a flow network $G = (V, E)$ with vertices $V = \{s, v_1, v_2, v_3, v_4, t\}$ and edges E as indicated, where the capacity $c(u, v)$ of each edge (u, v) is labeled. In the second figure, we have labeled the flow across each edge of a flow function f to the left of each edge's capacity. Though not pictured for every edge, we set $f(v, u) = -f(u, v)$ for all $(u, v) \in E$. The reader may quickly verify that f satisfies the Capacity Constraint and the Skew Symmetry properties, and by summing the flows into and out of each vertex, that f also satisfies the Flow Conservation property of flow functions. The value of the flow f, in this case, is $|f| = 7$, which is the total flow out of the source s and into the sink t.

Additionally, for a flow network $G = (V, E)$ and a flow f, we define the *residual capacity* of

an edge $(u, v) \in E$ to be $c_f(u, v) = c(u, v) - f(u, v)$; that is, the difference between the flow currently being sent across an edge and the capacity of the edge. The *residual network* of G induced by the flow f is $G_f = (V, E_f)$, where $E_f = \{(u, v)|u, v \in Eandc_f(u, v) > 0\}$. The residual network G_f of the flow network given in the above example is pictured below:



(c) The residual network G_f of the flow network pictured above. Each edge (u, v) is labeled with $c_f(u, v)$.

4.2 Statement of the Maximum Flow Problem

With this terminology in mind, we can formulate a precise statement of the maximum flow problem: For a flow network $G = (V, E)$ with source s and sink t, what flow function f maximizes $|f|$? We will solve this problem using the Ford-Fulkerson algorithm, which was first presented by Lester Ford, Jr. and Delbert Fulkerson in 1956 [5]. Similar to the presentation of Dijkstra's Algorithm, we will first give the Ford-Fulkerson Algorithm as a sequence of steps and then again in pseudocode.

4.3 Ford-Fulkerson Algorithm

Let $G = (V, E)$ be a flow network with source s and sink t. We claim that it produces a flow function f that maximizes $|f|$. Note that $c_f(P)$ is simply a temporary variable for the residual capacity of the path P.

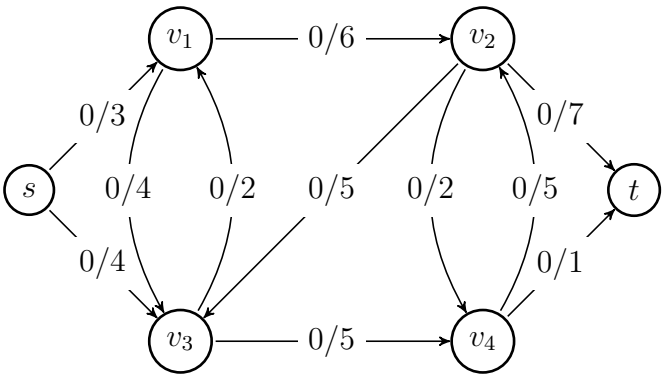
1. For each edge $(u, v) \in E$, initialize $f(u, v) = f(v, u) = 0$.
2. If there is a path P from s to t in the residual network G_f, continue to step 3, otherwise terminate.
3. Set $c_f(P) = \min_{(u,v)\in P} c_f(u, v)$.
4. For each $(u, v) \in P$, set $f(u, v) = f(u, v) + c_f(P)$ and $f(v, u) = -f(u, v)$.
5. Return to step 2.

In pseudocode, we can express this algorithm as follows:

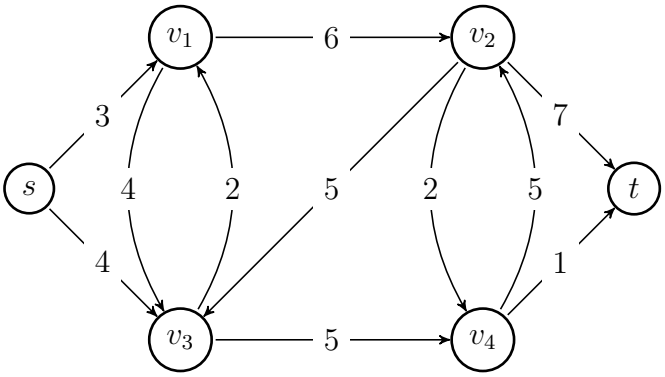
```
FlowFunction fordFulkerson(FlowNetwork G, Vertex s, Vertex t) {
  FlowFunction f = new FlowFunction();

  // initialize the values of f for each edge in G to 0
  for each Edge e in G {
    f.setValue(e.from, e.to) = 0;
    f.setValue(e.to, e.from) = 0;
  }

  // recall that the residual network of G has the same vertices as G, but
  // contains only those edges e of G with a positive residual capacity cf(e)
  ResidualNetwork Gf = G.residualNetwork(f);

  // here we assume that Gf.getPath(s, t) returns any path from s to t in Gf
  // (it does not matter which one) or null if no such path exists
  Path P = Gf.getPath(s, t);
  while(P != null) {
    // find the minimum residual capacity of the edges on P
    int cfmin = NaN;
    for each Edge e in P {
      if(cfmin == NaN || e.residualCapacity(f) < cfmin)
   cfmin = e.residualCapacity(f);
    }

    // update the flow of f across each edge on P to be its current flow plus
    // the minimum residual capacity of the edges on P
    for each Edge e in P {
      f.setValue(e.from, e.to) = f.getValue(e.from, e.to) + cfmin;
      f.setValue(e.to, e.from) = -f.getValue(e.from, e.to);
    }
  }

  return f;
}
```

4.4 Example Using the Ford-Fulkerson Algorithm

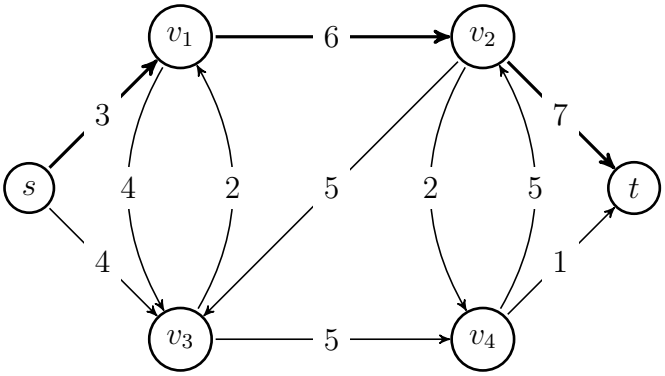
Before we prove the correctness of the Ford-Fulkerson Algorithm, we will show an example of its application using the flow network $G = (V, E)$ from the figures above. At each step where we select a new path P, we will mark it in bold.



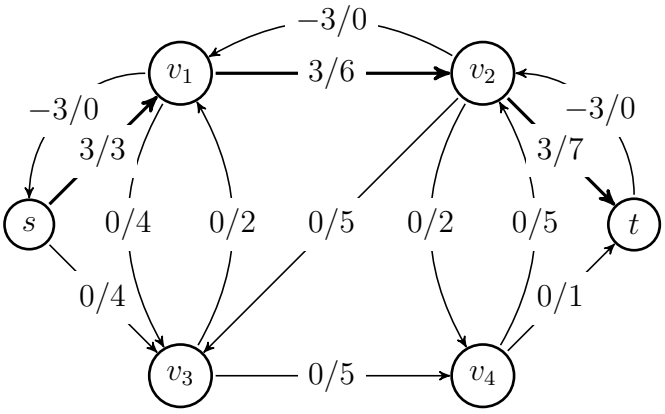
(a) Initialize the value of f for each edge to 0. Here the flow network G is shown with each edge (u, v) labeled as $f(u, v)/c(u, v)$.



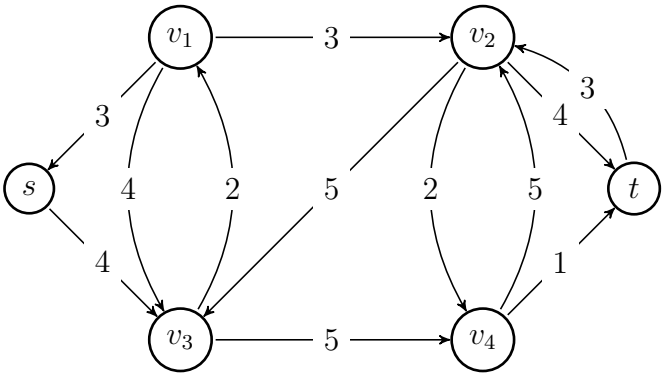
(b) The flow network G_f, with each edge (u, v) labeled as $v_f(u, v)$.



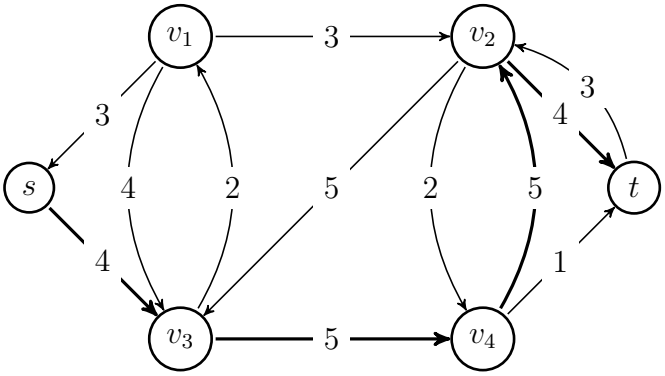
(c) Select a path P from s to t. In this case, $c_f(P) = 3$. Here the values of $c_f(u, v)$ are shown on each edge.



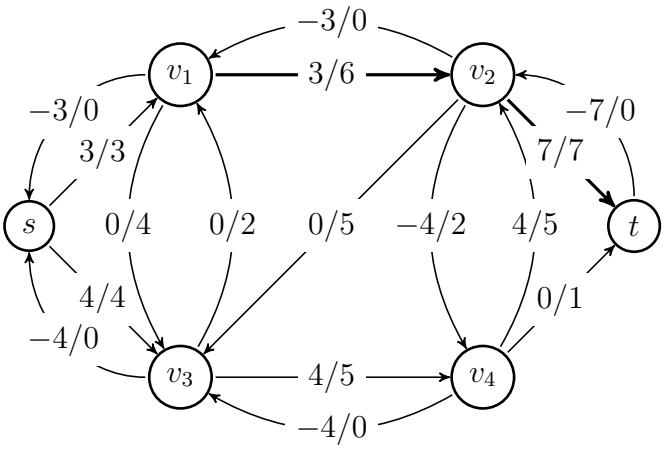
(d) Update the values of f along P. Here each edge (u, v) is labeled as $f(u, v)/c(u, v)$ We have included some edges with zero capacity simply to show the flow across them.



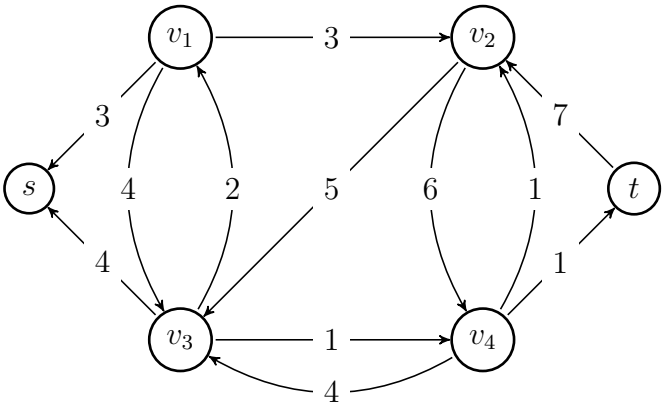
(e) The resulting residual network G_f, with edges (u, v) labeled as $c_f(u, v)$.



(f) Select a new path P from s to t. Here $c_f(P) = 4$.



(g) Update the values of f for each edge along the path. Each edge (u, v) is labeled as $f(u, v)/c(u, v)$.



(h) The resulting residual network G_f, with edges (u, v) labeled as $c_f(u, v)$. Since there is no longer a path from s to t, the algorithm terminates.

4.5 The Correctness of the Ford-Fulkerson Algorithm

The correctness of the Ford-Fulkerson Algorithm is actually a result of the Max-Flow Min-Cut Theorem, which we will prove here after introducing a few lemmas and their consequences. Before we begin, however, we must first introduce a few more terms.

First, for two flow function f_1, f_2 over the set of edges E, we define their *flow sum* $f_1 + f_2$ to be the function $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$ for $(u, v) \in E$. The *cut* of a flow network $G = (V, E)$ is a partition of V into two sets S and T, where $s \in S$, $t \in T$, and $T = V - S$. The *net flow* across the cut (S, T) is $f(S, T)$, and the capacity of the cut is $c(S, T)$. A *minimum cut* is a cut with the minimum possible capacity of any cut.

4.5.1 Lemma

Let $G = (V, E)$ be a flow network with source s, sink t, and flow function f. Then we have

1. For all $X \in V$, $f(X, X) = 0$.
2. For all $X, Y \in V$, $f(X, Y) = -f(Y, X)$.
3. For all $X, Y, Z \in V$ with $X \cap Y = \emptyset$, $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

4.5.2 Lemma

Let $G = (V, E)$ be a flow network with source s, sink t, and flow function f. Let G_f be the residual network of G induced by f, and let f' be a flow function of G_f. Then the flow sum $f + f'$ is a flow in G with value $|f + f'| = |f| + |f'|$.

Proof: To show that the flow sum is a flow in G, we must verify the properties of flow

functions. For the Capacity Constraint, we have

$$\begin{aligned}(f+f')(u,v) &= f(u,v) + f'(u,v) \\ &\leq f(u,v) + (c(u,v) - f(u,v)) \qquad \text{(Since } f'(u,v) \leq c_f(u,v)\text{)} \\ &= c(u,v),\end{aligned}$$

so the first property holds. For Skew Symmetry, we have

$$\begin{aligned}(f+f')(u,v) &= f(u,v) + f'(u,v) \\ &= -f(v,u) - f'(v,u) \\ &= -(f(v,u) + f'(v,u)) \\ &= -(f+f')(v,u),\end{aligned}$$

so the second property holds. For flow conservation, for all $u \in V - s, t$ we have

$$\begin{aligned}\sum_{v \in V}(f+f')(u,v) &= \sum_{v \in V}(f(u,v) + f'(u,v)) \\ &= \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) \\ &= 0 + 0 \\ &= 0,\end{aligned}$$

so the third property holds and thus the flow sum $f + f'$ is a flow of G. Finally, the value of the flow sum is

$$\begin{aligned}|f+f'| &= \sum_{v \in V}(f+f')(s,v) \\ &= \sum_{v \in V}(f(s,v) + f'(s,v)) \\ &= \sum_{v \in V} f(s,v) + \sum_{v \in V} f'(s,v) \\ &= |f| + |f'|,\end{aligned}$$

which concludes the proof.

4.5.3 Lemma

Let $G = (V, E)$ be a flow network with source s, sink t, and flow function f, and let G_f be the residual network induced by f. Let P be a path from s to t in G_f. Then the function defined by

$$f_p(u,v) = \begin{cases} c_f(P) & \text{If } (u,v) \in P, \\ -c_f(P) & \text{If } (v,u) \in P, \\ 0 & \text{If } (u,v) \notin P \text{ and } (v,u) \notin P \end{cases}$$

is a flow in G_f. Furthermore, it has value $|f_p| = c_f(P) > 0$, since G_f has edges with positive capacities.

4.5.4 Lemma

Let $G = (V, E)$ be a flow network with source s, sink t, and flow function f, and let G_f be the residual network induced by f. Let P be a path from s to t in G_f, and let f_P as defined in Lemma 4.5.3. Define the flow f' as $f' = f + f_P$.

Then by Lemma 4.5.2 and Lemma 4.5.3, f' is a flow in G with value $|f'| = |f| + |f_P| > |f|$.

4.5.5 Lemma

Let $G = (V, E)$ be a flow network with source s, sink t, and flow function f, and let (S, T) be a cut of G. Then the net flow $f(S, T) = |f|$.

Proof: By Flow Conservation, $f(S - \{s\}, V) = 0$. Applying Lemma 4.5.1, we have

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) && \text{(By part 3)} \\ &= f(S,V) && \text{(By part 1)} \\ &= f(s,V) + f(S-s,V) && \text{(By part 3)} \\ &= f(s,V) && \text{(By Flow Conservation)} \\ &= |f|. \end{aligned}$$

4.5.6 Lemma

Let $G = (V, E)$ be a flow network with flow function f. Then for any cut (S, T) of G, $|f| \leq c(S, T)$.

Proof: By Lemma 4.5.5 and the Capacity Constraint, we have

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T). \end{aligned}$$

4.5.7 Theorem (Max-Flow Min-Cut Theorem)

We are finally able to prove the Max-Flow Min-Cut Theorem, which shows that once the flow network G_f of a flow network G and a flow function f no longer has a path from the source to the sink, that f has the maximum value of any possible flow function for G. Since the Ford-Fulkerson Algorithm iterates over the paths in G_f, improving upon the flow function f until no path from exists from s to t in G, proving the Max-Flow Min-Cut Theorem is equivalent to showing that the Ford-Fulkerson Algorithm is correct. We state the theorem as follows:

Let $G = (V, E)$ be a flow network with source s, sink t, and flow function f, and let G_f be the residual network of G induced by f. Then the following statements are equivalent:

1. f is a maximum flow in G.

2. G_f has no path from s to t.

3. $|f| = c(S,T)$ for some cut (S,T) of G.

Proof: First we will show that if (1) holds, then (2) also holds. Let f be a flow function that maximizes the flow in G, and suppose that there is still some path P in G_f from s to t. Let f_P be defined as in Lemma 4.5.3. Then by Lemma 4.5.4, the flow sum $f + f_P$ has value $|f + f_P| = |f| + |f_P| > |f|$, which contradicts the assumption that f maximizes the flow in G. So (1) implies (2) as we wanted.

Now we will show that (2) implies (3). Suppose that G_f has no path from s to t, and let $S = \{v \in V |$ there is a path from s to v in $G_f\}$, $T = V - S$. Then (S,T) is a cut of G, since $s \in S$, and there is no path from s to t in G_f, so $t \notin S$ and hence $t \in T$. Let $u \in S$, $t \in T$. Then $f(u,v) = c(u,v)$, since if this were not the case, (u,v) would be an edge of G_f and hence v would be in S. By Lemma 4.5.5, then, $|f| = f(S,T) = c(S,T)$ as we wanted to show.

Finally we will show that (3) implies (1). By Lemma 4.5.6, we have $|f| \leq c(S,T)$ for all cuts (S,T) of G. Since $|f| = c(S,T)$ by (3), f is a maximum flow function of G. Thus we have shown the equivalence of (1), (2), and (3), which completes the proof.

5 Conclusion

In this paper, we have solved two related problems in network theory. First, given a network where the speed of the connection between every two components is known, we can find the fastest possible connection from one component, such as a computer, to another, such as a web server, by applying Dijkstra's Algorithm to the corresponding graph. Second, given a network where there is a limit to how much traffic can pass over each connection between components, we can use the Ford-Fulkerson Algorithm to determine a flow function that gives us the maximum possible traffic between one component and another.

These two algorithms present but a partial picture of network theory, however. We can find the optimum path from one component to another, but what if one component along the route were to freeze? It would be helpful to know the second- and third-fastest routes to the destination as well. In addition, we have taken an omniscient view of the graphs that we have examined so far; how can we have the individual components in a network communicate effectively to solve the shortest-path or max-flow problems? There are also more aspects than speed and channel capacity to consider in finding the "best" route between components, since reliability, i.e. the number of packets a connection successfully transmits versus how many are sent over it, is just as or perhaps more important than speed in many cases.

For further reading, *Introduction to Algorithms* [1] is a great resource on the use and runtime of graph algorithms, and *Data Structures and Algorithm Analysis in Java* [6] presents graph algorithms and their implementations in code from a computer science perspective.

References

[1] Cormen, Thomas H. *Introduction to Algorithms*. 2nd ed. Cambridge, Massachusetts: MIT, 2001.

[2] Dechter, Rina, and Judea Pearl. "Generalized Best-first Search Strategies and the Optimality of A*." *Journal of the ACM* 32.3 (1985): 505-36. *ACM Digital Library*. Web. 18 May 2010.

[3] Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[4] Even, Shimon. *Graph Algorithms*. Computer Science Press, 1979.

[5] Ford, Jr., L. R., and D. R. Fulkerson. "Maximal Flow Through a Network." *Canadian Journal of Mathematics* 8 (1956): 399-404. *Canadian Mathematical Society*. Web. 2 June 2010.

[6] Weiss, Mark Allen. *Data Structures and Algorithm Analysis in Java*. 2nd ed. Boston: Pearson Addison-Wesley, 2007.

[7] Wilson, Robin. *Introduction to Graph Theory*. New York, NY: Academic Press, 1972.