18.415/6.854 Advanced Algorithms September 8th, 2008

Network Flows

Lecturer: Michel X. Goemans

1 Introduction

In the previous lecture, we introduced Fibonacci heaps, which is a data structure that provides an efficient implementation of priority queues. In this lecture, we switch our attention from efficiency to algorithm design. In particular, for the next few lectures we study **Network Flows**.

Network flows are a family of problems that are concerned with a directed graph and properties of functions defined on the graph. A flow is an abstraction of elements which typically do not disappear while travelling through the edges of the directed graph; it could be current in an electrical network, packets in a computer network, cars/trains in a transportation network, or some purely abstract object. In the maximum flow problem, we try to obtain a flow on the graph such that the flow going from a given source vertex to a given sink vertex is maximized.

In today's lecture, we focus on two instances of network flow problems: the **Shortest Path Problem** and the **Maximum Flow Problem**. There are other variants of network flow problems that we cover later in this class. For example, we will talk about the minimum cost flow or minimum cost circulation problem, which is a generalization of both the shortest path problem and the maximum flow problem. We will also cover the bipartite matching problem, which has two versions: cardinality bipartite matching (a special case of the maximum flow problem) and weighted bipartite matching (a special case of the minimum cost flow problem). There are still other network flow problems that we do not discuss such as the multi-commodity flow problem. Figure 1 illustrates how these network flow problems are related to one another.

2 Shortest Path Problem

Let $G = (V, E)$ be a directed graph, where V denotes the set of vertices and E denotes the set of edges. Let $\ell\colon E \to \mathbb{R}$ be a length function defined on the edges of G.[1] Given two vertices s and t in V, the $s - t$ shortest path problem is the problem of finding a simple directed path on G from s to t of *minimum total length.* The length of a path P is defined to be the sum of the lengths of all the edges in P:

$$\ell(P) = \sum_{(v,w)\in P} \ell(v, w).$$

In this problem, we refer to s as the "source" vertex and t as the "sink" vertex.

We note that if the length function $\ell(e)$ is non-negative for every edge $e \in E$, then Dijkstra's algorithm using Fibonacci heaps provides a $O(m + n \log n)$ solution to this problem, where $m = |E|$ and $n = |V|$. On the other hand, if some edges of G have negative lengths, but the graph has the property that for every cycle C the total length of the cycle is non-negative, then we can use the Bellman-Ford algorithm to solve the $s - t$ shortest path problem in polynomial time. For more information on Dijkstra's and the Bellman-Ford algorithm, see Chapter 24 in [CLRS].

[1] For $v, w \in V$, we use the notation $\ell(v, w)$ to mean the length of the edge $e = (v, w)$. In these notes, we use the two notations $\ell(e)$ and $\ell(v, w)$ interchangeably.

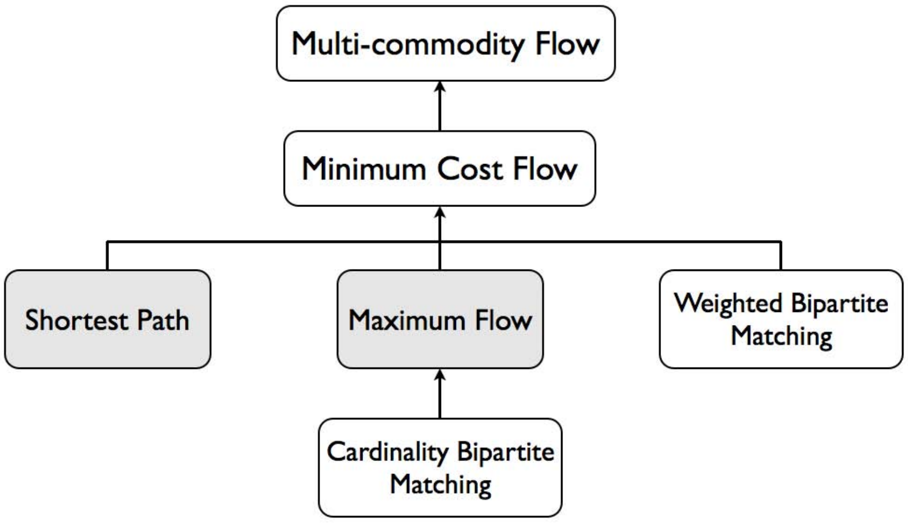


Figure 1: Some instances of network flow problems and how they are related to one another, where the arrow indicates "is a special case of". In this lecture we only cover the shaded boxes: the shortest path problem and the maximum flow problem.

Remark 1: In this lecture, we consider directed graphs only. For undirected graphs with non-negative edge lengths, we can still apply Dijkstra's algorithm by transforming every (undirected) edge into two edges of opposite directions with the same length, as illustrated in Figure 2.

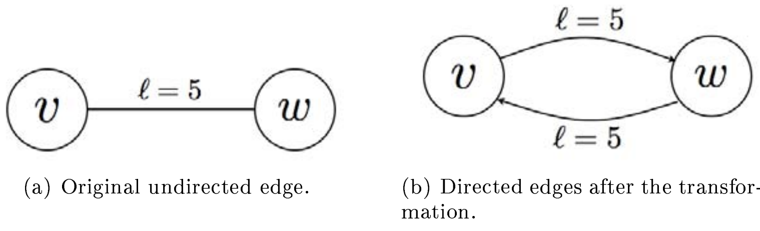
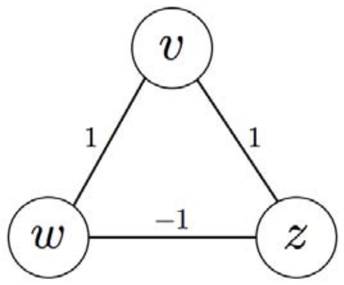

v
$\ell = 5$
w
(a) Original undirected edge.
$\ell = 5$
v
w
$\ell = 5$
(b) Directed edges after the transformation.

Figure 2: Transformation of an undirected edge into two directed edges to apply Dijkstra's algorithm.

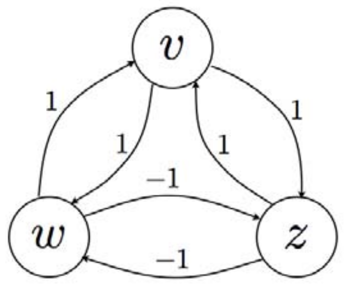
However, the same trick does not apply for the Bellman-Ford algorithm, because even if the original undirected graph satisfies the constraint that every cycle has non-negative length, the new directed graph resulting from the transformation might violate this constraint. An example of this case is given in Figure 3.

The problem of finding the shortest path between two vertices in an undirected graph where every cycle has non-negative length is still solvable in polynomial time, but it is a much harder problem. We will discuss this problem later in the class if time permits.

Remark 2: In directed graphs with non-negative, given a shortest path P between two vertices, the path between any two vertices in P is also the shortest path between those two vertices. However, this is not necessarily true in the case of undirected graphs (and this prevents the use of a transformation to a directed graph). For example, in the graph given in Figure 3(a), the shortest path between v and w is $P = \{(v, z), (z, w)\}$ with length 0. However, the shortest path between v and z is not $\{(v, z)\}$ as it appears in P, but rather $\{(v, w), (w, z)\}$ with length 0.



(a) Original undirected graph. Every cycle has non-negative length.



(b) Directed graph after the transformation. The cycle $\{(w,z),(z,w)\}$ has negative length.

Figure 3: An example where the given transformation creates a negative cycle so that the Bellman-Ford algorithm cannot be applied.

3 Maximum Flow Problem

The second instance of network flow problems that we study in this lecture is the maximum flow problem. In this problem, we want to find a flow from a source vertex to a sink vertex with maximum flow value.

More precisely, we define the problem framework as follows. Let $G = (V, E)$ be a directed graph, where V is the set of vertices and E is the set of edges of G. Let n denote the cardinality of V and m denote the cardinality of E. Given a vertex $v \in V$, let $N^+(v)$ (resp. $N^-(v)$) denote the set of endpoints of edges coming out (resp. into) v:

$$N^+(v) = \{w \in V : (v,w) \in E\},$$
$$N^-(v) = \{w \in V : (w,v) \in E\}.$$

Furthermore, let $u\colon E \to \mathbb{R}_+$ be a capacity function that limits the amount of flow that we can send through each edge of G. We refer to the graph G and the capacity function u collectively as the **network** G. Given a source vertex $s \in V$ and a sink vertex $t \in V$, we are interested in determining how much flow we can push from s to t through this network.

3.1 Notions of Flow

Loosely speaking, a flow is an assignment of quantity to the edges of G under certain constraints. There are two notions of flow that we use in this class: raw flow and net flow.

Definition 1 *A* **raw flow** *on a network G is a function* $r\colon E \to \mathbb{R}$ *satisfying the following properties:*

1. **Capacity constraint:** *For all* $(v,w) \in E$, $0 \le r(v,w) \le u(v,w)$.
2. **Conservation constraint:** *For all* $v \in V \setminus \{s,t\}$,

$$\sum_{w \in V:(v,w)\in E} r(v,w) - \sum_{w \in V:(w,v)\in E} r(w,v) = 0.$$

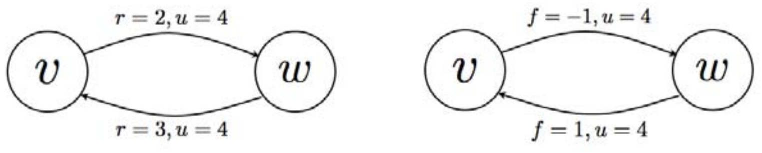
Given a raw flow r, the **flow value** of r is defined to be the total excess of flow at the source vertex s, i.e.

$$|r| = \sum_{w \in N^+(s)} r(s,w) - \sum_{w \in N^-(s)} r(w,s).$$

We now give the second definition of flow, which is the one we primarily use for the rest of these notes.

Definition 2 *Given a raw flow r on a network G, the* **net flow** *f with respect to r is the function $f\colon E \to \mathbb{R}$ given by $f(v,w) = r(v,w) - r(w,v)$.*

An example of raw flow and the corresponding net flow is illustrated in Figure 4.



(a) Raw flow. (b) Net flow.

Figure 4: An example of a raw flow and its corresponding net flow.

Before we go any further, we first note that from the definition given above, to compute $f(v,w)$ we need both $r(v,w)$ and $r(w,v)$. However, there is a slight difficulty because even if $(v,w) \in E$, (w,v) might not be an edge of G. To resolve this issue, we assume that the graph G has the property that if $(v,w) \in E$ then $(w,v) \in E$. Given a directed graph G, we can achieve this property by modifying G as follows:

1. Consider the set $E' = \{(v,w) \in E : (w,v) \notin E\}$.
2. For every $(v,w) \in E'$, create a new edge (w,v) with edge capacity 0 and add it to E.

Similar to the definition of the flow value of raw flow, the **flow value** of f is defined to be the total amount of net flow that comes out from the source vertex s:

$$|f| = \sum_{w \in N(s)} f(s,w), \tag{1}$$

where we now use $N(s)$ to denote $N^+(s) = N^-(s)$, the common set of out-neighbors and in-neighbors of s.

From the definition of net flow, it is easy to check that the net flow f satisfies the following properties:

1. **Skew symmetry:** For all $(v,w) \in E$, $f(v,w) = -f(w,v)$.
2. **Capacity constraint:** For all $(v,w) \in E$, $f(v,w) \le u(v,w)$.
3. **Flow conservation:** For all $v \in V \setminus \{s,t\}$, $\sum_{w \in N(v)} f(v,w) = 0$.

Note that, unlike r, the flow f has no restriction on being negative. In fact, f will be negative for some edges, unless it is the 0 flow everywhere. For example, if the original graph G has an edge (v,w) with positive raw flow $r(v,w)$ such that (w,v) is not an edge, then in the modified graph, the edge (w,v) has negative net flow $f(w,v) = -r(v,w)$. Note that this does not violate the capacity constraint since $f(w,v) \le u(w,v) = 0$. Figure 5 illustrates an example of a net flow.

For the maximum flow problem, we use the notion of net flow. For the rest of these notes, unless specified otherwise, the term flow refers to net flow. We can now define the maximum flow problem properly.

Definition 3 (Maximum Flow Problem) *Given a network G, a source vertex $s \in V$, and a sink vertex $t \in V$, the maximum flow problem is the problem of finding a flow through G of maximum flow value.*

Notice that modifying G by adding to E the new edges needed to define the net flow does not affect the maximum flow problem, since the new edges all have zero capacity.

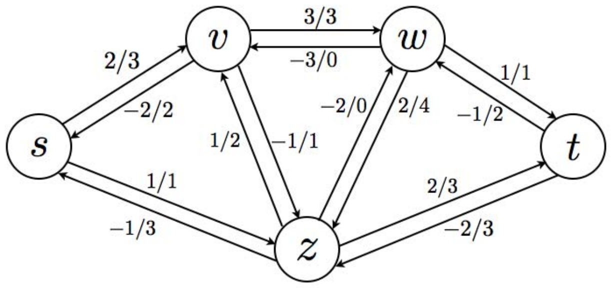


Figure 5: An example of a flow of a network. The label x/y on each edge e is such that $x = f(e)$ and $y = u(e)$. Here the flow value is $|f| = 3$.

3.2 $s - t$ Cut

We now define the notion of *cut*, which helps us to construct the solution of the maximum flow problem.

Definition 4 *Suppose that we have a network G with source vertex s and sink vertex t. Let S be a subset of V such that $s \in S$ and $t \notin S$, and let $\overline{S} = V \setminus S$. Then the $s - t$* **cut** *with respect to S is defined to be*

$$(S : \overline{S}) = \{(v, w) \in E : v \in S \text{ and } w \in \overline{S}\}.$$

We can also denote an $s-t$ cut by $\delta^+(S)$ or $\delta^-(\overline{S})$, but in this class the preferred notation is $(S : \overline{S})$ as introduced above. Figure 6 shows an example of an $s - t$ cut.

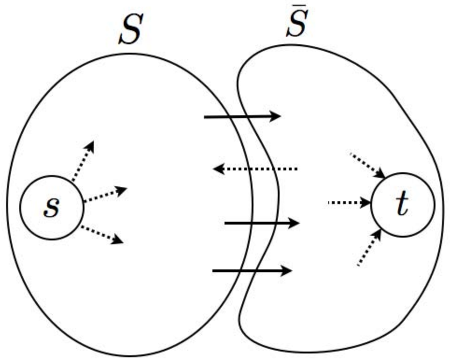


Figure 6: An example of an $s - t$ cut. The solid arrows represent the edges in $(S : \overline{S})$.

Definition 5 *Given an $s - t$ cut $(S : \overline{S})$, then its* **cut capacity** *is defined to be the total capacity of the edges across the cut:*

$$u(S : \overline{S}) = \sum_{(v,w)\in(S:\overline{S})} u(v, w).$$

3.3 Connection between Flows and Cuts

We have the following lemma that connects flows and cuts.

Lemma 1 *Let G be a network with source s and sink t. Then for every flow f and every $s-t$ cut $(S:\overline{S})$, we have*

$$|f| = \sum_{(v,w)\in(S:\overline{S})} f(v,w). \tag{2}$$

In particular, this implies that $|f| \leq u(S:\overline{S})$.

Proof: From the flow conservation property of f, for every vertex $v \in S \setminus \{s\}$, we have

$$\sum_{w\in N(v)} f(v,w) = 0.$$

Taking the sum over all vertices $v \in S \setminus \{s\}$ gives us

$$\sum_{v\in S\setminus\{s\}} \sum_{w\in N(v)} f(v,w) = 0.$$

Adding the definition of the flow value of f (Eq. (1)) to the equation above yields

$$|f| = \sum_{w\in N(s)} f(s,w) + \sum_{v\in S\setminus\{s\}} \sum_{w\in N(v)} f(v,w).$$

Now notice that if an edge (v,w) appears in either of the summations above and $w \in S$, then (w,v) also appears in the summations. Therefore, we can rewrite the equation above in a slightly different way:

$$|f| = \sum_{(v,w)\in(S:\overline{S})} f(v,w) + \sum_{v\in S} \sum_{w\in S} f(v,w).$$

By the skew-symmetry property of f, the second summation in the equation above is equal to 0 since $f(v,w)$ and $f(w,v)$ cancel each other out. Therefore, we conclude that

$$|f| = \sum_{(v,w)\in(S:\overline{S})} f(v,w),$$

as desired.

Furthermore, by the capacity constraint of f, we can write

$$|f| = \sum_{(v,w)\in(S:\overline{S})} f(v,w) \leq \sum_{(v,w)\in(S:\overline{S})} u(v,w) = u(S:\overline{S}).$$

This completes the proof of the lemma. □

In particular, if we take $S = V \setminus \{t\}$ and $\overline{S} = \{t\}$, then Eq. (2) from Lemma 1 tells us that the flow coming from s is equal to the flow going to t. In other words, there is no loss in the flow of the network.

An important corollary to Lemma 1 comes from the observation that since the value of any flow f is always less than equal to the capacity of any $s-t$ cut $(S:\overline{S})$, then it also holds for the case when f is a maximum flow and $(S:\overline{S})$ is a minimum cut. This fact is known as the Weak-Duality Lemma.

Corollary 2 (Weak-Duality Lemma) *Let G be a network with source vertex s and sink vertex t. Then*

$$\max_f |f| \leq \min_{(S:\overline{S})} u(S:\overline{S}),$$

where the maximum is taken over all possible flows and the minimum is taken over all possible $s-t$ cuts in G.

4 The Max-Flow and Min-Cut Theorem

In this section, we show that the inequality in the Weak-Duality Lemma is actually an equality, that is, the maximum value of a net flow is *equal* to the minimum value of an $s-t$ cut. This fact was first discovered in 1956 by by Elias, Feinstein, and Shannon (see [EFS]), and independently by Ford and Fulkerson in the same year.

Theorem 3 (Duality Theorem/Maxflow Mincut Theorem) *In a network G, the following equality holds:*

$$\max_f |f| = \min_{(S:\overline{S})} u(S:\overline{S}).$$

In order to prove the theorem, we first have to introduce some new definitions. The first one is residual capacity, which denotes the extent to which a flow on some edge is less than the capacity on that edge.

Definition 6 *The* **residual capacity** *of G with respect to f is the function $u_f\colon E \to \mathbb{R}$ defined by $u_f(v,w) = u(v,w) - f(v,w)$ for all (v,w) in E. Hence, the residual capacity on the edge (v,w) is the amount of additional flow that we can push from v to w, without violating the capacity constraint.*

We observe that the capacity constraint implies that $u_f(v,w) = u(v,w) - f(v,w) = u(v,w) + f(w,v) \leq u(v,w) + u(w,v)$. Moreover, since f is a flow, $u(v,w) \geq f(v,w)$, so that $u_f(v,w) \geq 0$. Hence, the following inequality holds for any edge (v,w) in E:

$$0 \leq u_f(v,w) \leq u(v,w) + u(w,v).$$

All the edges with positive residual capacities are members of a set that we call the residual arcs.

Definition 7 *The* **residual arcs** *E_f of G with respect to f is the set given by $E_f = \{(v,w) \in E : u_f(v,w) > 0\}$. Intuitively, the residual arcs is the subset of E that contains those edges through which we can push a non-zero additional flow.*

Given the vertices of a network G, its residual arcs, and its residual capacity, we can make a new network, the residual network.

Definition 8 *The* **residual network** *G_f of the network G with respect to f is the network given by the graph $G_f = (V, E_f)$ together with the capacity function u_f.*

The residual network is used to understand to what extent a flow is not maximal, and we do that by defining a certain kind of path in the residual network that we call augmenting path.

Definition 9 *An* **augmenting path** *of G with respect to f is a directed simple path from the source s to the sink t in the residual network G_f.*

In fact, the existence of an augmenting path in a residual network for a given flow indicates that the flow is not maximal, as we prove in the following lemma.

Lemma 4 *If a residual network G_f has at least one augmenting path P, then f is not a maximum flow.*

Proof: By definition, the residual network G_f includes only edges with non-zero residual capacity with respect to f. Therefore, an augmenting path P of G_f is a path through which we can push more flow in the original network G, and the additional amount of flow is upper bounded by the "bottleneck" of P.

More precisely, consider the quantity given by

$$\epsilon(P) = \min_{(v,w)\in P} u_f(v,w).$$

Observe that $\epsilon(P) > 0$, because $P \subset E_f$ so that P is a finite set of positive real numbers.

Then, construct the flow f' given by

$$f'(v,w) = \begin{cases} f(v,w) + \epsilon(P) & \text{if } (v,w) \in P, \\ f(v,w) - \epsilon(P) & \text{if } (w,v) \in P, \\ f(v,w) & \text{otherwise.} \end{cases}$$

Note that f' is satisfies all the flow constraints for G. Moreover, $|f'| = |f| + \epsilon(P) > |f|$, so that the flow f is not a maximum flow. □

Using Lemma 4 and the Weak-Duality Lemma, we prove now the Maxflow Mincut Theorem.

Proof of Theorem 3: Let f be a flow of maximal value for $G = (V, E)$. By Lemma 4, the residual network G_f has no augmenting path, since, if it did, then f would not be of maximal value.

Consider the set S of vertices $v \in V$ such that there exists a directed path from the source s to v in G_f. By definition, $s \in S$. Moreover, G_f has no augmenting path, so that $t \notin S$. Therefore, $(S : \overline{S})$ is an $s - t$ cut.

Now notice that $u_f(v,w) = 0$ for any $(v,w) \in (S : \overline{S})$. By definition, $u_f(v,w) = u(v,w) - f(v,w)$, so that $f(v,w) = u(v,w)$ for any $(v,w) \in (S : \overline{S})$. Thus, we can compute that

$$|f| = \sum_{(v,w)\in(S:\overline{S})} f(v,w) = \sum_{(v,w)\in(S:\overline{S})} u(v,w) = u(S : \overline{S}).$$

The Weak-Duality Lemma tells us that the value of any flow is upper bounded by the capacity of any $s - t$ cut, so we can conclude that

$$\max_f |f| = \min_{(S:\overline{S})} u(S : \overline{S}).$$

□

We summarize all of the results in the following theorem.

Theorem 5 (Max-Flow Min-Cut Theorem) *Let G be a network and f be a flow on G. Then, the following statements are equivalent:*

1. *f is a flow of maximal value;*
2. *G_f has no augmenting path; and*
3. *$|f| = u(S : \overline{S})$ for some $s - t$ cut $(S : \overline{S})$.*

Proof: We prove the equivalence of the statements by showing that (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (1), that is:

- (1) $\Rightarrow$ (2): This implication is the contrapositive of the implication proved in Lemma 4.
- (2) $\Rightarrow$ (3): This implication follows from the proof of the Maxflow Mincut Theorem.
- (3) $\Rightarrow$ (1): This implication follows from the Weak Duality Lemma.

□

5 The Ford-Fulkerson Algorithm

In 1956 Ford and Fulkerson used the Max-Flow Min-Cut Theorem to design an algorithm, called the Ford-Fulkerson algorithm, to compute the maximal flow of a network (see [FF]). The idea of their algorithm is very simple: as long as there is an augmenting path in the residual network we push more flow along that path in the original network. This idea is illustrated as pseudocode below.

```
FORD-FULKERSON($G$)
1  start with a zero flow $f$ (or any feasible flow)
2  while $G_f$ has an augmenting path $P$
3        do push $\epsilon(P)$ more units of flow through $P$, so that $|f| \leftarrow |f| + \epsilon(P)$
```

Before we declare the idea above an algorithm, there are two issues that need to be addressed:

1. Does the algorithm ever halt?
2. If there is more than one augmenting path in the residual network, which one should we choose? And how does our decision affect the correctness and running time of the algorithm?

We consider three cases.

Case 1: Assume that the capacity function u of G is **integer** valued. Then we can make the following observations:

1. At every iteration of FORD-FULKERSON, the flow f is integer valued, and therefore so are the residual capacities. Indeed, this is the case at the beginning when $f = 0$, and by induction, this is maintained since $\epsilon(P)$ is the minimum of a set of positive integers and thus a positive integer, and therefore the resulting flow after an augmentation is also integer valued.

   Furthermore, since $\epsilon(P) \geq 1$ (being a positive integer) and since the minimum-cut value (and thus the maximum flow value) is finite, it follows that the FORD-FULKERSON always halts.

2. Since the algorithm halts and every intermediate flow is integer valued, the maximum flow output will also be integer valued. That is, *if the capacities of a network are integral then there is a maximum flow that is also integral.* This is a very useful property that has many applications. One such application is the cardinality bipartite problem, as we will see in the next lecture.

3. The number of iterations is bounded by $|f| \leq |N(s)|U \leq nU$, where $U = \max\{u(s,w) : w \in N(v)\}$. Note that U may *not* be polynomial in the size of G. In fact, Figure 7 shows an example of a graph where FORD-FULKERSON takes exponential time to halt. The dotted and dashed lines represent paths from the source to the sink. The algorithm might choose alternatively and repeatedly the two paths as augmenting paths. In such a case, the algorithm will take $O(2^L)$ time to terminate. Thus, we need a better policy to choose the augmenting path.

Case 2: Assume that the capacity function u of G is **rational** valued. Then, a similar discussion as the one carried out in Case 2 shows that FORD-FULKERSON always halts, that the value of the maximal flow is rational, and that there exists an example of a network for which the running time is exponential. The arguments are similar because the rational capacities behave like integers if we consider them all as written with the same least common multiple.

Case 3: Assume that the capacity function u of G is **real** valued. In the general case (i.e. $u(E) \subset \mathbb{Q}_+$ is not necessarily true) there exist instances of networks such that FORD-FULKERSON never halts. Moreover, in such cases, the value of $|f|$ may converge to a *sub-optimal* value.

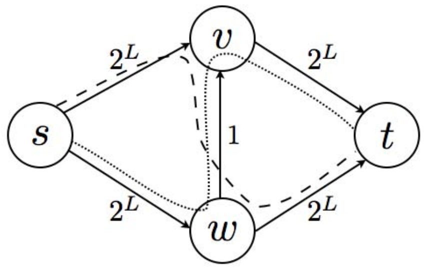


Figure 7: An example of a network for which the Ford-Fulkerson algorithm may not halt in polynomial time (the reverse edges and the corresponding flows are not shown for clarity).

6 Fixing the Ford-Fulkerson Algorithm

The problems of the Ford-Fulkerson algorithm that we examined at the end of Section 5 can be addressed, at least in part, by specifying a policy for choosing the augmenting path at every iteration. A good policy must satisfy two properties:

1. It is possible to efficiently (e.g. in polynomial time) find the augmenting path specified by the policy; and
2. The maximum number of augmentations (and thus the total time) is polynomial.

In fact, we should be precise when we say that a running time is "polynomial", because it means different things depending on the model of computation. Also, ideally, we would like algorithms for which the number of operations does not depend on the size of the numbers involved in the input (e.g. the capacities in a maximum flow instance); such algorithms could be used even if the data was irrational (provided our model allows (arithmetic) operations on irrational data).
Given an instance I of a number problem (a computational problem involving numbers as input), let $size(I)$ denote the number of bits needed to represent the input and $number(I)$ denote the number of numbers involved in the input. For example, for a maximum flow instance, $number(I)$ corresponds to the number m of edges while $size(I)$ corresponds to the number of bits needed to represent all edge capacities. For the solution of an $n \times n$ system of linear equations, $number(I)$ will be $n^2 + n$ (n^2 for the matrix and n for the right-hand-side) while $size(I)$ is the sum of the binary sizes of all the entries of the matrix and the right-hand-side.
We say that an algorithm A running on an instance I is (weakly) ***polynomial*** if

- the number of operations performed by A is at most polynomial in $size(I)$ and
- the size of any number obtained during the execution of A is at most polynomial in $size(I)$.

For an algorithm to be ***strongly polynomial***, we require that

- the number of operations performed by A is at most polynomial in $number(I)$ and
- the size of any number obtained during the execution of A is at most polynomial in $size(I)$.

Thus, the two notions differ only in whether the number of operations performed depends on the size of the numbers in the input. For example, Gaussian elimination can be shown to be strongly polynomial for solving a system of equations (it is clear that the number of operations is at most $O(n^3)$, but one can also show that the size of the numbers obtained through the algorithm are polynomially bounded in the size of the input). On the other hand, Euclid's algorithm for computing the gcd is clearly not strongly polynomial (as only 2 numbers are involved), but is polynomial.

We now consider two policies for choosing the augmenting path in the Ford-Fulkerson algorithm. Both were proposed by Edmonds and Karp in 1972 [EK]. Both lead to polynomial algorithms, while the second leads to a strongly polynomial algorithm.

Pick the Fattest: Suppose that, in the case of integral capacities, at every iteration of the Ford-Fulkerson algorithm, we pick the "fattest" augmenting path, that is, a path P such that $\epsilon(P)$ is maximized. Given this policy:

- By adapting Dijkstra's algorithm to find this *bottleneck* path rather than the shortest path, it is possible to find the augmenting path that maximizes $\epsilon(P)$ in $O(m + n \log n)$ time;
- It can be shown that the number of iterations is $O(m \log U)$, where U is a bound for the capacity function, yielding a running time for this *fattest augmenting path algorithm* of $O((m + n \log n) m \log U)$.

A similar argument works for rational capacities as well. However, for irrational capacities, the time complexity given above does not apply, and this analysis does not even show whether the algorithm terminates.

Pick the Shortest: Suppose that, in the case of integral capacities, at every iteration of the Ford-Fulkerson algorithm, we pick the "shortest" augmenting path, that is, a path P such that its number of edges is minimized. Given this policy, we observe that:

- Using breadth-first search, it is possible to find the augmenting path with a minimum number of edges in $O(m)$ time (by breadth-first-search);
- It can be shown that the number of iterations is $O(nm)$, yielding a running time for the algorithm of $O(nm^2)$. Thus this *shortest augmenting path* algorithm is **strongly polynomial** and therefore halts even if capacities are irrational.

Next time we will discuss more network flow problems.

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition, MIT Press and McGraw-Hill, 2001.

[EFS] P. Elias, A. Feinstein, and C. E. Shannon, *Note on maximum flow through a network*, IRE Transactions on Information Theory IT-2, 117–119, 1956.

[EK] Jack Edmonds, and Richard M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of the ACM 19 (2): 248–264, 1972.

[FF] L. R. Ford, D. R. Fulkerson, *Maximal flow through a network*, Canadian Journal of Mathematics 8: 399–404, 1956.