Network Flow

G&T 16.1, 16.2, K&T §7.1, 7.2

Problem: Given a directed graph $G = (V, E)$, and a **positive integer** capacity c_e for each edge, along with distinguished vertices s (the "source") and t (the "sink"), and this definition of "flow":

Definition An $s - t$ flow puts f_e flow on each edge e such that:

- Each capacity is obeyed; this is called the "capacity constraint"

$$0 \leq f_e \leq c_e$$

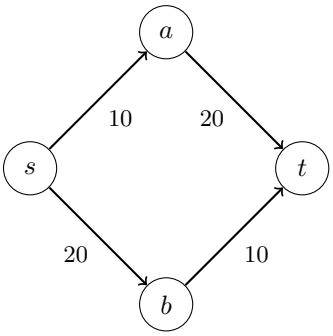
- "Conservation of flow": only the source can create flow, and only the sink can consume it. More formally, for each $v \neq s, t$:

$$\sum_{e \ into \ v} f_e = \sum_{e \ out \ of \ v} f_e$$

- The source can create any amount of flow, and the sink can consume any amount. We say the *value* of a given flow f is:

$$v(f) = \sum_{e \ out \ of \ s} f_e$$

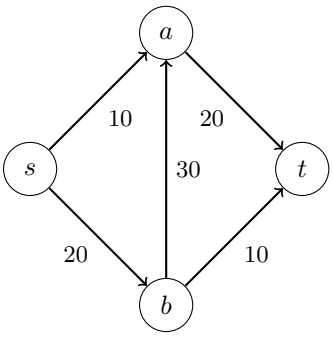
Our goal is to find a valid flow with maximum possible $v(f)$, the "maximum flow."



Let's first try a basic greedy algorithm: as long as we can add flow, do so.

$\forall_e \ \ f_e = 0$
while $\exists$ path p from s to t in G with all $c_e > 0$ **do**
  p = any simple s to t path in G, using only edges with $c_e > 0$
  m = min capacity edge on p
  for all edges e in p **do**
    $f_e = f_e + m$
    $c_e = c_e - m$
for all edges e **do**
  Final flow along e is f_e

Let's try it with another graph and see if it works:



What we end up needing is a way to "take back" flow that we put on an edge.

We will define a *residual graph* G_f, for any given flow f, where the capacities on each edge are :

- If $e = (u, v) \in G$, then c_e in G_f is $c_e - f_e$. In other words, the capacity is the remaining capacity that has yet to be used by the flow. We call these **forward** edges.
- If $e = (u, v) \in G$, define $e' = (v, u)$. $c_{e'}$ in G_f is f_e. In other words, the capacity of e' is the flow that can be "taken back" – that was tentatively sent from u to v. We call these **backward** edges.

The changes we make will result in this algorithm, known as FORD-FULKERSON.

$\forall_e \quad f_e = 0$
while $\exists$ path p from s to t in G_f **do**
  p = any simple s to t path in G_f.
  b = min residual capacity edge on p (the "bottleneck" edge)
  for all edges $e = (u, v)$ in p **do**
    if e is forward **then**
      $f_e = f_e + b$
    else
      $e' = (v, u)$
      $f_{e'} = f_{e'} - b$

Let's talk a little bit about this algorithm:

1. Recall that all c_e values are positive integers. Will any f_e values ever be non-integer? Why or why not?

2. What is the minimum increase in $v(f)$ each iteration of the while loop?

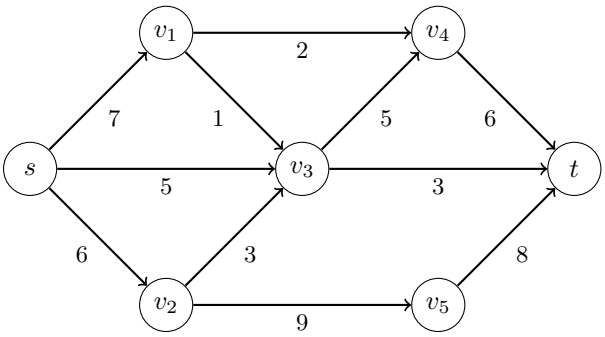
3. Define $C = \sum c_e$ over all $e = (s, v)$. How many iterations will the **while** loop take in the worst case?

4. How long does each iteration of the **while** loop take?

5. What is the total running time of the algorithm?

6. Is this polynomial time?
   Recall that polynomial time is a formalism that fits our intuition for an algorithm that scales efficiently as the input size doubles. Suppose the integer variables for all capacities is a C++ ***int*** *(32 bits) and then we change it to allow for* ***long****s (64 bits) instead, thus doubling the amount of space used to represent edge capacities. What effect does this have, if any, on the variables in the running time found in question 5?*

Review: Find Maximum Flow



Proof of Correctness

Let's prove that the Ford-Fulkerson algorithm actually finds the maximum flow. To do this, we need one more definition:

Define (A,B) cut: a partition of V into sets A and B such that $s \in A$, $t \in B$. We say that the *capacity* of the cut is:

$$c(A, B) = \sum_{e \ out \ of \ A} c_e$$

The following facts will be useful to us in proving that it is a max flow:

1. If f is any $s-t$ flow and (A, B) is any $s-t$ cut, then $v(f) = f^{out}(A) - f^{in}(A)$. This shouldn't surprise you: intuitively, the total that flows from s to t is the net amount that exits from set A.
2. If f is any $s-t$ flow and (A, B) is any $s-t$ cut, then $v(f) = f^{in}(B) - f^{out}(B)$.

3. If f is any $s-t$ flow and (A, B) is any $s-t$ cut, then $v(f) \leq c(A, B)$
4. If we can show a flow and a cut of equal value, it follows that the flow is a maximum and the cut is a minimum.

Concept Exercises

1. Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given an integer maximum $s-t$ flow in G, defined by a flow value f_e on each edge e.

   Now suppose we pick a specific edge $e \in E$ and increase its capacity by one unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m+n)$.

2. Suppose a concert has just ended and C cars are parked at the venue. We would like to determine how long it takes for all of them to leave the area. For this problem, we are given a graph representing the road network where all cars start at a particular vertex s (the parking lot) and several vertices $(t_1, t_2, \ldots, t_k)$ are designated as exits. We are also given capacities (in cars per minute) for each road (directed edges). Give a polynomial-time algorithm to determine the amount of time necessary to get all cars out of the area.

Applying and Interpreting Network Flow

Relevant Reading: G&T 16.3, K&T §7.5, 7.6

Finding the maximum flow in a flow network isn't just a fun problem; the algorithms that solve it (and there are many) can be applied to give an efficient solution to many other problems.

7.14 We define the *Escape Problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads). A certain collection of nodes $X \subset V$ are designated as *populated nodes*, and a certain other collection $S \subset V$ are designated as *safe nodes*, with X and S being disjoint. In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

   (a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

(b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus we change (iii) to say "the paths do not share any *nodes*."
With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

Bipartite Matching

Relevant reading: G&T 16.3, K&T §7.5, 7.6

Consider the following problem:

Some cities have restrictions on pet ownership per household. Suppose you are in a city that limits each household to at most one dog, and you work for a pet adoption center. Dogs can only be adopted one day of the year, and that day is fast approaching. You have a collection of n households that want a dog and an equal number of puppies; each household that wants a dog has provided a list of which dog(s) they are interested in adopting. Show how to maximize the number of dogs that can be adopted this year.

1. Let's use network flow to solve this problem.

   (a) Define a graph G that we will use in future steps. What property in the graph are we looking for?

   (b) Create a source and sink for G; also, add edge capacities.

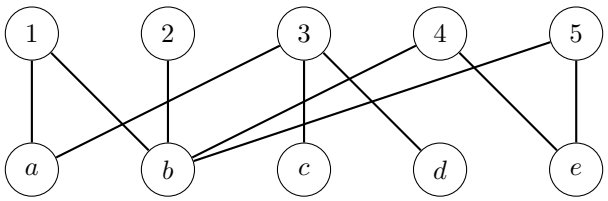
   (c) Suppose we find max flow on G. What does $v(f^*)$ tell us about the solution to the problem? How can we use it to prepare the adoptions?

2. Suppose every dog can be adopted by a distinct household. It should be very easy to convince someone of this fact. How?

3. Suppose it isn't possible, and at least one dog won't be adopted. Someone who hasn't taken CSCI 570 probably won't accept a "proof" of this fact that amounts to "my algorithm returned a less than perfect match."
   How could we use a similar algorithm such that the output can convince such a person of our unfortunate situation?

To solve this, we're going to use **Hall's Theorem**. For any $A \subseteq V$, we define $\Gamma(A)$ (the "neighborhood" of A) to be the set of vertices that are adjacent to one or more elements of A. If we have a bipartite graph with partition $L \cup R$, where $|L| = |R|$, then either the graph has a perfect matching or there is an $A \subseteq V$ such that $|\Gamma(A)| < |A|$.

For example, prove that the following bipartite graph *does not* have a perfect matching:



If our graph from the puppy adoption problem *does not* have a perfect matching, how can we use the result from running network flow to output a proof of this?

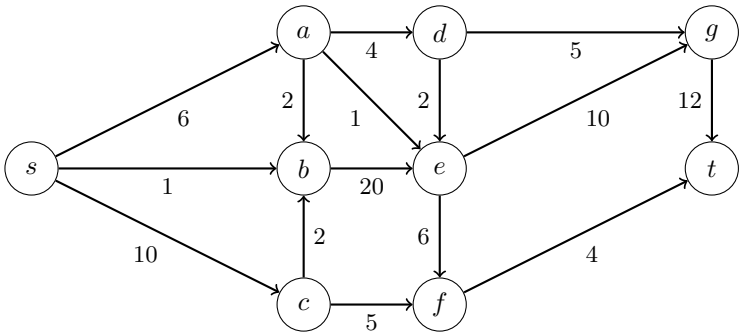
Achieving a Polynomial Running Time

See also: K/T §7.3

Recall the algorithm for computing a maximum flow, FORD-FULKERSON.

$\forall_e \quad f_e = 0$
while $\exists$ path p from s to t in G_f **do**
  p = any simple s to t path in G_f.
  b = min residual capacity edge on p (the "bottleneck" edge)
  for all edges $e = (u, v)$ in p **do**
    if e is forward **then**
      $f_e = f_e + b$
    else
      $e' = (v, u)$
      $f_{e'} = f_{e'} - b$

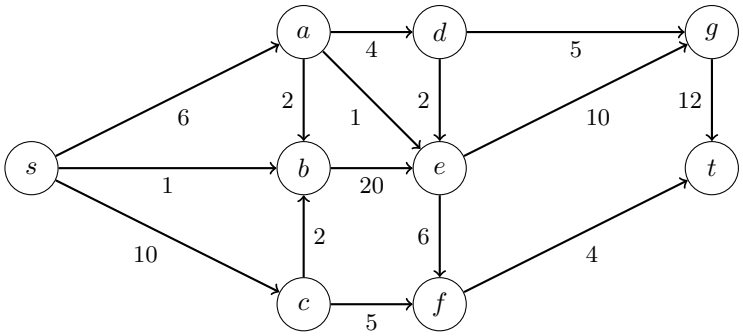
1. Find the maximum flow $v(f^*)$ in this graph[1]:



2. What is residual graph G_f after $v(f^*)$ is found?

[1]The graph in this handout is from the textbook "Algorithms" by Sanjoy Dasgupta, Umesh Vazirani, and Christos Papadimitriou ; page 240

3. What is the minimum cut on the same graph?



4. Suppose I want to select a "good" path if one exists. How can I find out if a path with residual capacity *at least* Δ (for some given integer Δ) remains in G_f?

5. We previously saw an algorithm (Ford-Fulkerson) with running time $O(Cm)$, which we also saw was not polynomial time. Let's modify the algorithm for it to be polynomial time.

```
∀_e  f_e = 0
X = maximum c_e out of s
for Δ = 2^⌊log_2 X⌋ ; Δ ≥ 1 ; Δ = Δ/2 do
  while ∃ path p from s to t in G_f(Δ) do
    p = any simple s to t path in G_f(Δ)
    b = min residual capacity edge on p (the "bottleneck" edge)
    for all edges e = (u, v) in p do
      if e is forward then
        f_e = f_e + b
      else
        e' = (v, u)
        f_e' = f_e' − b
```

6. What is that algorithm's running time?

   (a) How many iterations of the outer loop will happen?

   (b) How long will each iteration of the inner **while** loop take?

   (c) How many iterations of the inner **while** loop will happen during each iteration of the outer loop?

Supply, Demand, and Lower Bounds

K&T §7.7

1. Suppose we have a directed graph $G = (V, E)$, with capacities c_e on the edges, as per network flow. Unlike network flow, though, we don't have a distinguished source s nor sink t. Instead, some $F \subseteq V$ are marked as factories; for each factory i, we have a supply s_i of units of product that the factory has produced. We also have some $R \subseteq V$ marked as retail stores; for each store i, we believe that store can sell d_i units of the product. Our goal is to find a way to ship products from the factories so that they end up at the stores, in such a way that each store gets exactly d_i units, and each factory sends out its entire supply. Fortunately, $\sum_i s_i = \sum_i d_i$.

   Show how to use network flow to determine if this is possible in a given graph.

   (G, {Ce}, {Si}, {di})

2. We need to assign students to sections in a large freshman calculus class. Each section should have *at most thirty* students. Sections are held at various times, and each student submits a list of the sections his or her schedule permits attending. Every student must be assigned to *exactly one* section. Use network flow techniques to design a polynomial-time algorithm to determine an assignment of students to sections such that each student is assigned to a section that his or her schedule permits attending, or that determines if no such assignment is possible.

   students Sections

   .

   .

   .

3. We need to assign students to sections in a large freshman calculus class. Each section should have *between twenty and thirty* students. Sections are held at various times, and each student submits a list of the sections his or her schedule permits attending. Every student must be assigned to *exactly one* section. Use network flow techniques to design a polynomial-time algorithm to determine an assignment of students to sections such that each student is assigned to a section that his or her schedule permits attending, or that determines if no such assignment is possible.

Baseball Elimination

G&T §16.4, K&T §7.12

Suppose T is a set of teams in a sports league. At any point during the season, each time i has some number of wins, w_i and some number of games left to play g_i (for each team j, team i has $g_{i,j}$ games left against j). Our goal is to determine if team i still has a chance to finish in first place. Note that sometimes a team cannot do this, even if they win every remaining game.

Team	Wins	Games Left	Schedule ($g_{i,j}$)			
i	w_i	g_i	LA	Oak	Sea	Tex
Los Angeles	81	8		1	6	1
Oakland	77	4	1		0	3
Seattle	76	7	6	0		1
Texas	74	5	1	3	1	

Let $W_k = w_k + g_k$ be the maximum number of wins possible for team k. There are two ways for team k to be eliminated:

- $W < w_i$ for some team i. That is, even if k wins out and i loses out, i still finishes with more wins.
- Some combination of games "mathematically eliminate" team k; that is, there is a set of teams T that will finish with a *strictly higher* average number of wins than W. Note that this means the league does not allow games to end in a tie. More formally:

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{x,y} > W \cdot |T|$$

Let's build a graph to determine if a given team k has been eliminated. The idea is going to be to use network flow to determine if there is a way to allocate wins across every series such that no team has strictly more wins than team k.

Airline Scheduling

K&T §7.9

Suppose you're managing a fleet of airplanes for an airline and would like to create a flight schedule for them. You have identified a set of m flight segments that you would like to serve; each has an origin airport, a destination airport, a departure time, and an arrival time. We have a fleet of k airplanes available to us.

For example, we might have the following six possible flights:

- Boston (6AM) to Washington D.C. (7AM)
- Philadelphia (7AM) to Pittsburgh (8AM)
- Washington D.C. (8AM) to Los Angeles (11AM)
- Philadelphia (11AM) to San Francisco (2PM)
- San Francisco (2:15 PM) to Seattle (3:15PM)
- Las Vegas (5PM) to Seattle (6PM)

There are various ways an airline might determine if a plane can be reused between flights, but for this problem, let's say that flight j is either *reachable* from flight i or it isn't, and that this information is somehow available to us.

We will need to represent the following *ideas* in the graph:

- Each flight on the list must be served
- The same plane can sometimes perform two flights
- Any flight can be the first of the day for any given plane.
- Any flight can be the last of the day for any given plane.
- If we can service every flight with fewer than k planes, we are under no obligation to use all k.