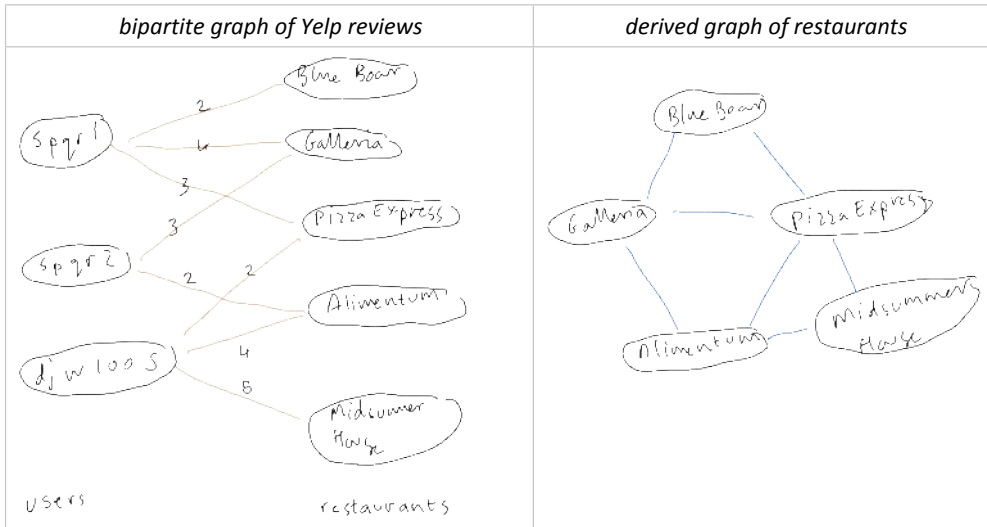


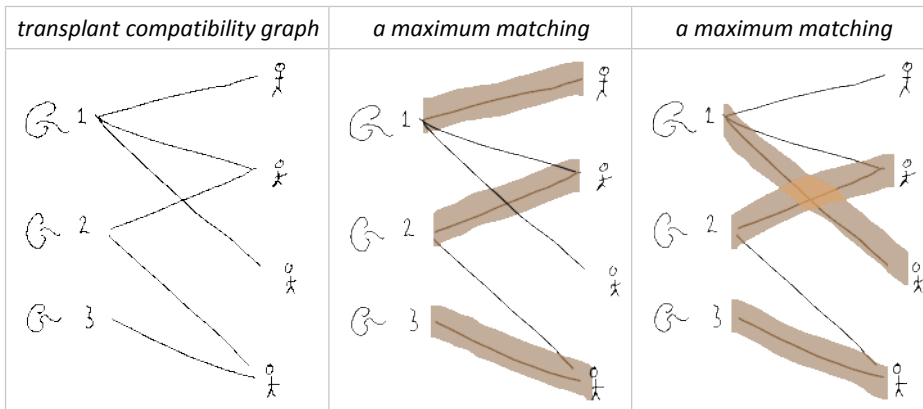
5.11. Matchings

A **bipartite graph** is one in which the vertices are split into two sets, and all the edges have one end in one set and the other end in the other set. We'll assume the graph is undirected. For example

- Vertices for medical school graduates, vertices for hospitals offering residency training, and edges for each application the medic has made to a hospital.
- Vertices for Yelp users, vertices for restaurants, and edges labelled with weights to indicate the user's rating of the restaurant. (We could derive further graphs from this by "hiding" one set of vertices, e.g. to produce a graph with vertices for restaurants, and edges between restaurants that have been rated by the same users.)



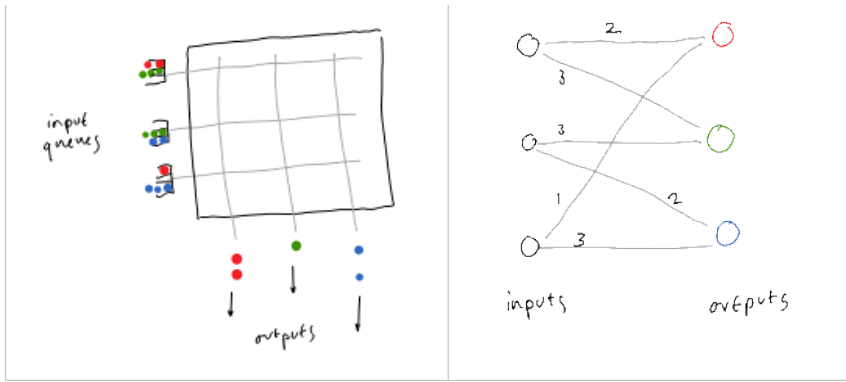
A **matching** in a bipartite graph is a selection of some or all of graph's edges, such that no vertex is connected to more than one edge in this selection. For example, kidney transplant donors and recipients, with edges to indicate compatibility. A **maximum matching** is one with the largest possible number of selected edges. There may be several maximum matchings.



Application: Internet routing

The silicon in the heart of an Internet router has the job of forwarding packets from inputs to outputs. Every clock tick, it can take at most one packet from each input, and it can send at most one packet to each output---in other words, it selects a matching from inputs to outputs. It turns out to be useful to weight the edges by the number of packets waiting to be sent, and to pick a matching with the highest possible total weight.





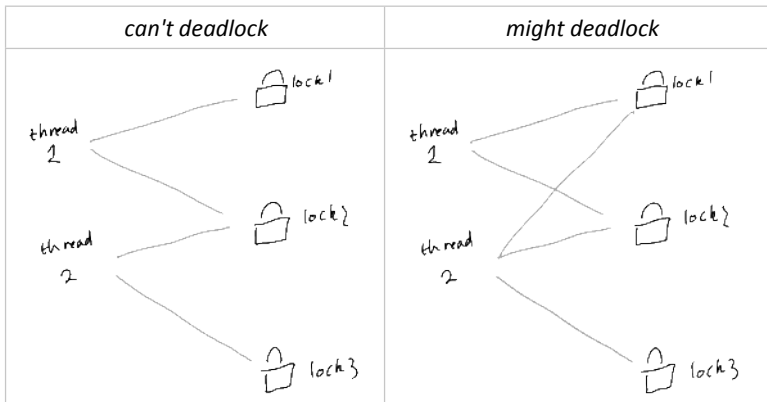
Application: taxi scheduling

A company like Uber has to match taxis to passengers. When there are passengers who have made requests, which taxis should they get? The greedy strategy "pick the nearest available taxi as soon as a request is made" might lead to shortages on the edge of the network, or to imbalanced workloads among drivers. A different strategy is "once a minute, draw an edge from each passenger to the ten nearest available vehicles, and look for a maximum matching". In this as in many other applications, matchings are computed over and over again, and a bad choice will have knock-on effects.

Application: deadlock analysis

A *concurrent* program is one with several simultaneous threads of execution (as opposed to the *sequential* algorithms we have studied so far). If there is a variable that is shared by several threads, this can lead to problems: one thread might be in the middle of updating a variable's value when another thread tries to read it. We can prevent these collisions using *locks*: a thread requests a lock on a variable, then it reads or writes that variable, then it releases the lock. If one thread has the lock and another thread requests it, the second thread is *blocked* until the first thread releases it. A concurrent program can run into *deadlock*, when each of a group of threads is waiting for some other thread in that group to release a lock. Deadlock is notoriously hard to reason about, and many programmers superstitiously avoid concurrency.

In this bipartite graph view of a concurrent system, one can prove the following. For any subset I of threads, create a subgraph consisting of these threads and the locks that may be requested by at least two threads in I , and draw edges between threads and the locks they might request. Call I a *blocking set* if this subgraph has a maximum matching involving all the resources. If the concurrent system has no blocking sets, then it cannot deadlock.



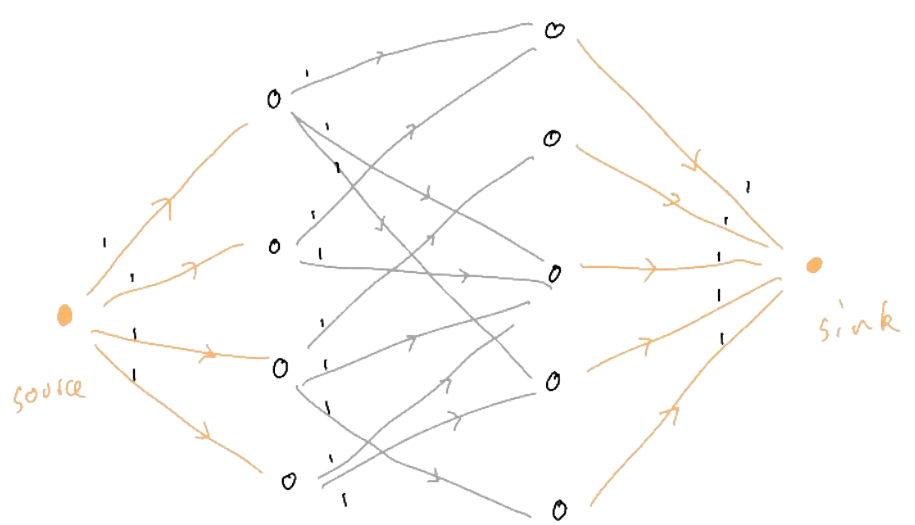
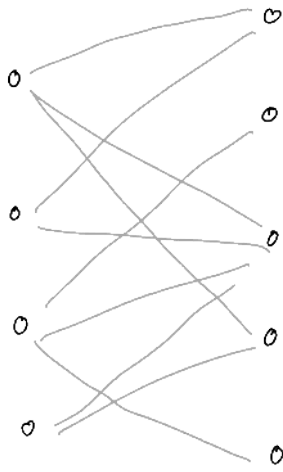
Implementation

An good way to find a maximum matching is to turn it into what looks like a harder problem, the *maximum flow problem*. **Read Section 5.12 now.** The translation is as follows.

A bipartite graph



A flow problem: add a source with edges to each left-hand vertex; add a sink with edges from each right-hand vertex; turn the original edges into directed edges from left to right; give all edges capacity 1.



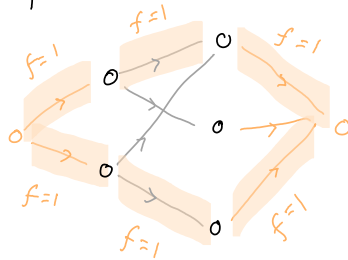
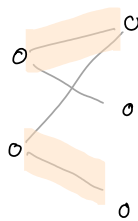
Analysis

Theorem 1 in Section 5.12 tells us that the max flow algorithm terminates, and produces a flow that is integer on all edges. Since the capacities are all 1, the flow must be 0 or 1 on all edges. We can straightforwardly translate it into a matching: just select all the edges from the original graph that got flow=1. The capacity constraints on the extra edges we added ensure that this selection is a matching.

Theorem. The matching obtained in this way is a maximum matching.

Proof

(1) Any matching can be translated into a flow:



Why is this a flow?
It's certainly ≥ 0 ;
and the definition of
"matching" implies the
flow conservation equation.

(2) The size of the matching (#edges selected) is clearly equal to the value of the flow (net flow out of source).

(3) If there were a larger matching than the one obtained by Ford-Fulkerson, it would correspond to a larger flow, which cannot be.

□

5.12. Ford-Fulkerson algorithm

To describe a transportation network, we can use a directed graph with edge weights: vertices for the junctions, edges for the roads or railway links or water pipes or electrical cables, whatever it may be that is being transported. An interesting question is: how much stuff can be carried by this network?

Let's suppose there is a single **source** vertex s producing stuff, and a single **sink** vertex t to which stuff is to be transported. (It's easy to generalise to multiple sources and sinks, rather harder to generalise to multiple types of stuff.) Let each edge $u \rightarrow v$ have a **capacity** $c(u \rightarrow v) \geq 0$.

A **flow** is a set of edge weights $f(u \rightarrow v)$ with $0 \leq f(u \rightarrow v) \leq c(u \rightarrow v)$ on every edge, such that at all vertices v , other than the source s and the sink t , flow is "conserved" i.e. as much stuff comes in as goes out:

$$\text{for all } v \in V \setminus \{s, t\},$$

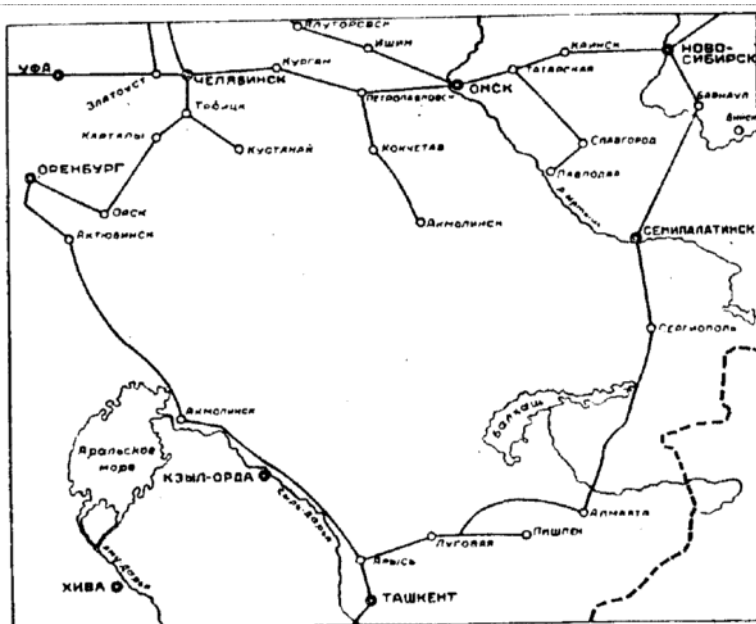
$$\sum_{u: u \rightarrow v} f(u \rightarrow v) = \sum_{w: v \rightarrow w} f(v \rightarrow w)$$

What flow will transport the largest amount of stuff from source to sink?

Applications

We've already seen how a matching problem can be turned into a flow problem (and then solved!). Now here is a pair of flow problems that inspired the algorithm we'll describe shortly.

The Russian applied mathematician A.N.Tolstoy was the first to formalize the flow problem. He was interested in the problem of shipping cement, salt, etc. over the rail network. Formally, he posed the problem "Given a graph with edge capacities, and a list of source vertices and their supply capacities, and a list of destination vertices and their demands, find a flow that meets the demands."



From "Methods of finding the minimal total kilometrage in cargo-transportation planning in space", A.N.Tolstoy, 1930.

In this illustration, the circles mark sources and destinations for cargo, from Omsk in the north to Tashkent in the south.

The US military was also interested in flow networks during the cold war. If the Soviets were to attempt a land invasion of Western Europe through East Germany, they'd need to transport fuel to the front line. Given their rail network, and the locations and capacities of fuel refineries, how much could they transport? More importantly, which rail links should the US Air Force strike and how much would this impair the Soviet transport capacity?

NOTICE: THIS DOCUMENT CONTAINS INFORMATION AFFECTING THE NATIONAL DEFENSE OF THE UNITED STATES WITHIN THE MEANING OF THE ESPIONAGE LAWS, TITLE 18, U.S.C., SECTIONS 793 and 794. THE TRANSMISSION OR THE REVELATION OF ITS CONTENTS IN ANY MANNER TO AN UNAUTHORIZED PERSON IS PROHIBITED BY LAW.

"Fundamentals of a method for evaluating rail net capacities", T.E. Harris and F.S. Ross, 1955.

A report by the RAND Corporation for the US Air Force, declassified by the Pentagon in 1999.

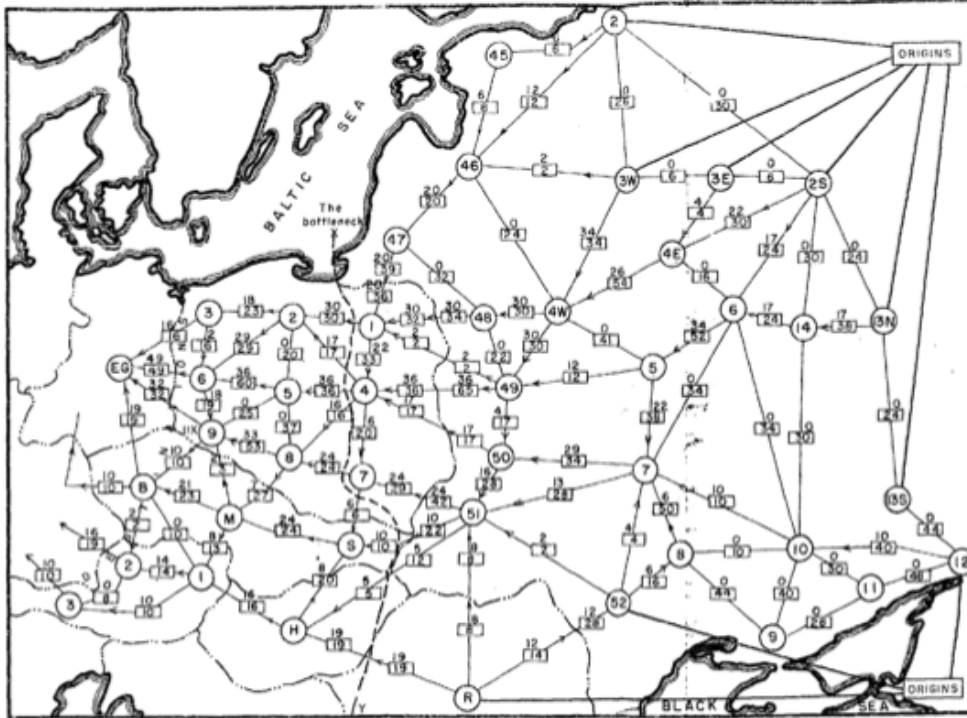


Fig. 7 - Traffic pattern: entire network available

SECRET 10-3773 10-24-55 -53-

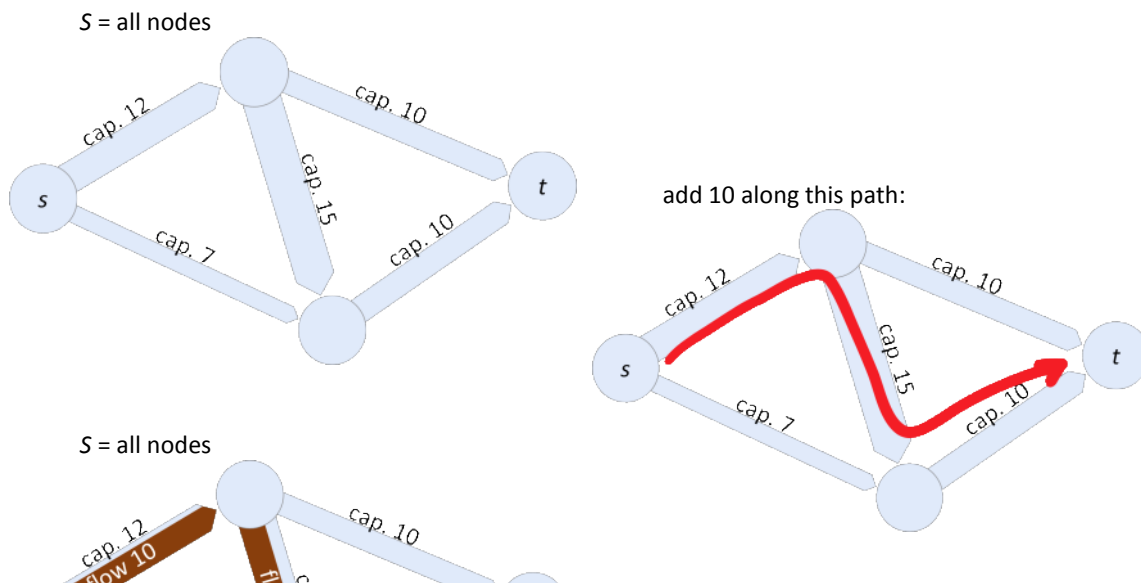
Legend:
 - - - International boundary
 (B) Railway operating division
 (12) Capacity: 12 each way per day. Required flow of 9 per day toward destinations (in direction of arrow) with equivalent number of returning trains in opposite direction

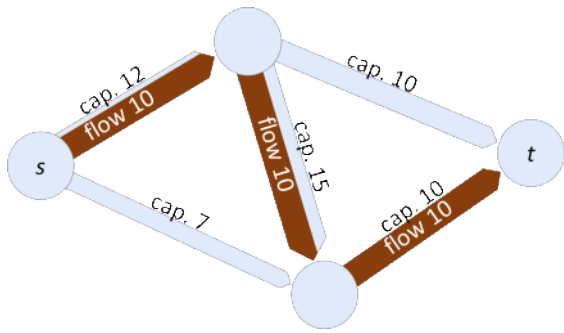
All capacities in trains /1000's of tons each way per day
 Origins: Divisions 2, SW, SE, 25, 13N, 13S, 12, 52 (USSR), and Roumania
 Destinations: Divisions 3, 6, 9 (Poland); 8 (Czechoslovakia); and 2, 3 (Austria)
 Alternative destinations: Germany or East Germany
 Note 11X of Division 9, Poland

General idea

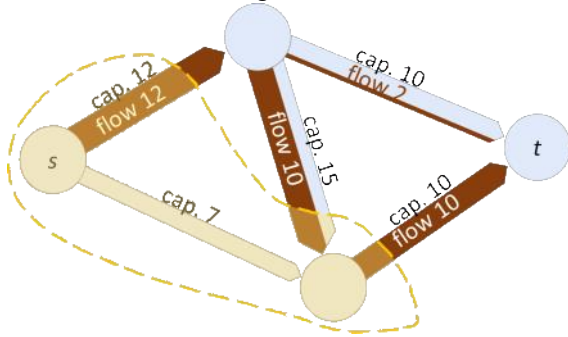
An obvious place to start is with a simple greedy algorithm. We'll build up a set S of vertices to which we could send more flow.

1. Start with just the source s , $S = \{s\}$.
2. Repeatedly, look at the neighbours of S and add any to which we could increase flow. In mathematical notation, we look for $v \in S$, $w \notin S$ and $f(v \rightarrow w) < c(v \rightarrow w)$, and add any such w to S .
3. If S grows to include the sink t , then pick out a path from source to sink along which we can add capacity. We'll call this an *augmenting path*. Add as much flow along this path as it allows, and go back to step 2.
4. If S does not include the sink, we're done.

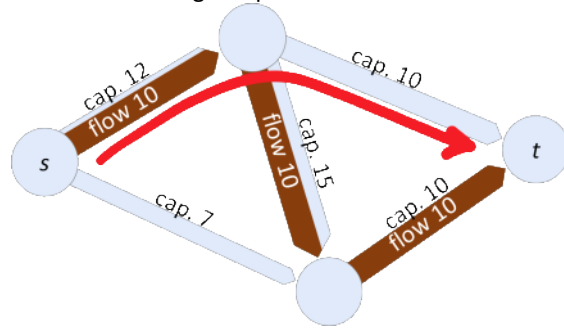




$S =$ two nodes, algorithm is done.



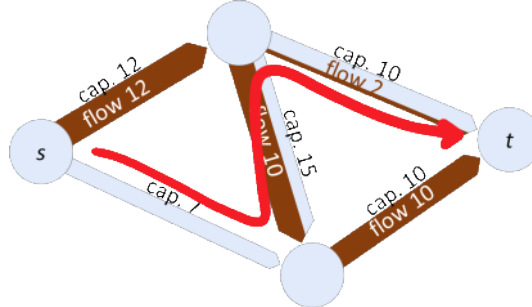
add 2 along this path:



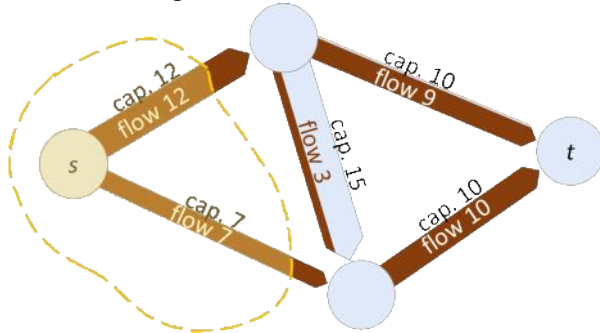
This greedy algorithm found a flow of size 12, then it finished. But we can easily see a flow of size 17 (10 on the top path, 7 on the bottom path). It turns out there is a very simple modification to step 2, to allow the algorithm to reassign flows, which enables it to find the maximum.

2. Look for $v \in S, w \notin S$ and **either** $f(v \rightarrow w) < c(v \rightarrow w)$ **or** $f(v \rightarrow w) > 0$, and add any such w to S .

add 7 along this path:



$S = \{s\}$, algorithm is done.



Problem statement

The **value of a flow** is the net flow out of the source. Given a weighted directed graph g with a source s and a sink t , find a flow from s to t with maximum value (also called a **maximum flow**).

Note: it's easy to prove that that the net flow out of the source must be equal to the net flow into the sink:



Intuition: net flow out of s

= net flow out of $\{s\} \cup \{\text{other vertices}\}$, since flow is conserved at all those vertices

= net flow into t .

All we need to do is write this intuition out with algebra:

$$\begin{aligned}
 \text{value of flow} &= \sum_v f(s \rightarrow v) - \sum_v f(v \rightarrow s) \\
 &= \sum_v f(s \rightarrow v) - \sum_v f(v \rightarrow s) + \sum_{w \neq s, t} \left[\sum_v f(w \rightarrow v) - \sum_v f(v \rightarrow w) \right] \\
 &= \sum_v f(s \rightarrow v) + \sum_{w, v} f(w \rightarrow v) - \sum_v f(s \rightarrow v) - \sum_v f(t \rightarrow v) \\
 &\quad - \sum_v f(v \rightarrow s) - \sum_{w, v} f(v \rightarrow w) + \sum_v f(v \rightarrow s) + \sum_v f(v \rightarrow t) \\
 &= \sum_v f(v \rightarrow t) - \sum_v f(t \rightarrow v) \\
 &= \text{net flow into } t
 \end{aligned}$$

where we'll take $f(u \rightarrow v) = 0$ if there's no $u \rightarrow v$ edge

this extra term is 0, since flow is conserved at all vertices other than s, t .

Algorithm

```

1  def ford_fulkerson(g, s, t):
2      # let f be a flow, initially empty
3      for u,v in g.edges:
4          f(u,v) = 0
5      while True:
6          S = Set([s]) # the set of vertices to which we can increase flow
7          while there are vertices v in S and u not in S with f(v,u) < c(v,u) or f(u,v) > 0:
8              S.add(u)
9          if t not in S: break
10         pick any path p from s to t made up of vertex pairs (v,u) discovered in line 7
11         # write p as s = v0, v1, v2, ..., vk = t
12         delta = infinity # amount by which we'll augment the flow
13         for each edge (vi, vi+1) along p:
14             if vi -> vi+1 is an edge of g:
15                 delta = min(c(vi -> vi+1) - f(vi -> vi+1), delta)
16             else:
17                 delta = min(f(vi+1 -> vi), delta)
18         # assert: delta > 0
19         for each edge (vi, vi+1) along p:
20             if vi -> vi+1 is an edge of g:
21                 f(vi -> vi+1) = f(vi -> vi+1) + delta
22             else:
23                 f(vi+1 -> vi) = f(vi+1 -> vi) - delta
24         # assert: f is still a flow (according to definition at top of this section)

```

ford_fulkerson

This pseudocode doesn't tell us how to choose the path in line 10. One sensible idea is "pick the shortest path", and this version is called the Edmonds-Karp algorithm. Another sensible idea is "pick the path that makes δ as large as possible", also due to Edmonds and Karp.

Analysis of running time

The assertions on lines 18 and 24 are both simple to verify.

Be scared of the while loop in line 5: how can we be sure it will terminate? In fact the example sheet asks you to step through the algorithm for a simple graph with irrational capacities where the algorithm does *not* terminate. On the other hand:

Theorem 1. If all capacities are integers then the algorithm terminates, and the resulting flow on each edge is an integer.

Proof. Initially, the flow on each edge is 0, i.e. integer. At each execution of lines 12--17, we start with integer capacities and integer flow sizes, so we end up with δ a positive integer. Therefore the total flow has increased by an integer after lines 19--23. The value of the flow can never exceed the sum of all capacities, so the algorithm must terminate. QED.

Now let's analyse running time, under the assumption that capacities are integer. We execute the while loop at most f^* times, where f^* is the value of maximum flow. Lines 12--23 involve some operations per edge of the augmenting path, which is $O(V)$ since the path is of length $\leq V$. The logic involving the set S can actually be accomplished using a stack and a per-vertex attribute $v.is_in_S$, and so lines 6--8 can be accomplished in running time $O(E)$, the time it takes to scan the edges of the graph. Therefore the total running time is $O(E V f^*)$.

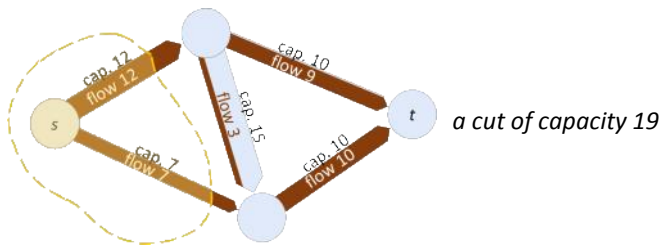
It is unsatisfactory that the running time we found depends on the values in the input data (via f^*) rather than just the size of the data. This is unfortunately a common feature of many optimization algorithms, and of machine learning algorithms.

The Edmonds-Karp version of the algorithm can be shown to have running time $O(E^2 V)$.

Analysis of correctness: the max-flow min-cut theorem

A **cut** is a partition of the graph's vertices into two sets, $V = S \cup S^c$, with $s \in S$ and $t \in S^c$. The algorithm looks for a cut, and if it breaks at line 9 it has found one. The **capacity of a cut** is the sum of edge capacities from S to S^c ,

$$\text{capacity of cut } (S, S^c) = \sum_{\substack{u \in S, v \in S^c \\ u \rightarrow v}} c(u \rightarrow v)$$



Lemma. For any flow and any cut, the value of the flow is \leq the capacity of the cut.

Theorem 2. If the algorithm terminates, then the value of the flow it has found is equal to the size of the cut found in lines 6--8.

Thus, if the algorithm terminates then it must have found a maximum flow---since by the lemma no other flow could have value greater than the capacity of the cut found in lines 6--8. We'll call this the **bottleneck cut**. The RAND report shows a bottleneck cut, and suggests it's the natural target for an air strike.

Proof of Lemma

let f be a flow, and (S, S^c) a cut.

$$\text{value of flow} = \sum_u f(s \rightarrow u) - \sum_u f(u \rightarrow s) \quad \text{ie net flow out of } S$$

$$= \sum_{v \in S} \left(\begin{array}{c} \sum_u f(v \rightarrow u) \\ - \sum_u f(u \rightarrow v) \end{array} \right) \quad \begin{array}{l} \text{since flow is conserved at} \\ \text{all other nodes in } S, \text{ so they contribute} \\ 0 \text{ to the sum} \end{array}$$

$$= \sum_{v \in S, u \in S} f(v \rightarrow u) + \sum_{v \in S, u \notin S} f(v \rightarrow u) - \sum_{v \in S, u \in S} f(u \rightarrow v) - \sum_{v \in S, u \notin S} f(u \rightarrow v) \quad \begin{array}{l} \text{splitting the sum into} \\ \text{two parts, } u \in S \text{ and } u \notin S \end{array}$$

$$\leq \sum_{v \in S, u \notin S} f(v \rightarrow u) \quad \text{since } f \geq 0 \quad (1)$$

$$\leq \sum_{v \in S, u \notin S} c(v \rightarrow u) \quad \text{since } f \leq c \quad (2)$$

= capacity of cut.

Proof of Theorem 2 let f be the flow it found, and let (S, S^c) be the cut it produced.

By the condition in line 7, $f(u \rightarrow v) = 0$ for $v \in S, u \notin S$, so inequality (1) is equality.

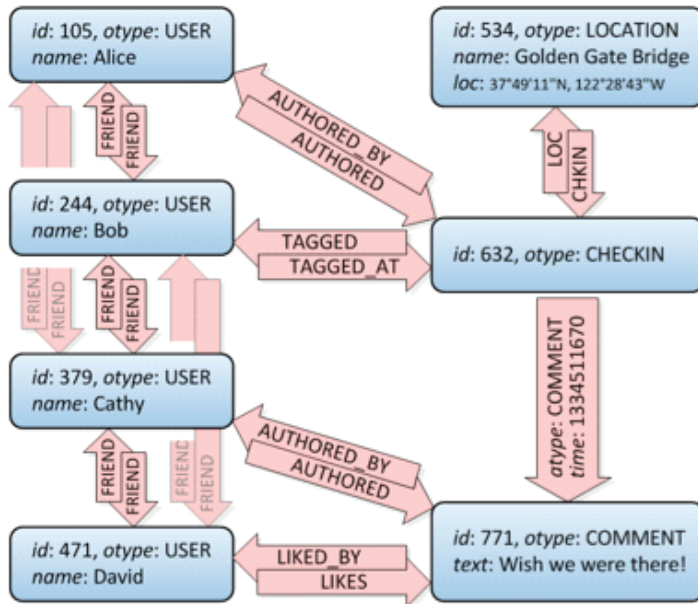
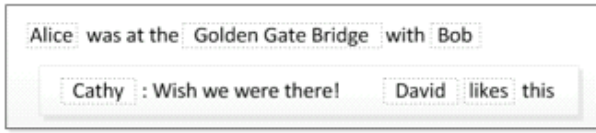
By the same condition, $f(v \rightarrow u) = c(v \rightarrow u)$ for $v \in S, u \notin S$, so inequality (2) is equality.

Thus value of f = capacity of (S, S^c)

5.13. Big graphs

Facebook

Facebook sees the world as a graph of objects and associations, their *social graph*:



Facebook represents this internally with classic database tables:

id	otype	attributes
105	USER	{name: Alice}
244	USER	{name: Bob}
379	USER	{name: Cathy}
471	USER	{name: David}
534	LOCATION	{name: Golden Gate Bridge, loc: (38.90,-77.04)}
632	CHECKIN	
771	COMMENT	{text: Wish we were there!}

from_id	to_id	edge_type
105	244	FRIEND
105	379	FRIEND
105	632	AUTHORED
244	105	FRIEND
244	379	FRIEND
244	632	TAGGED_AT
⋮	⋮	⋮

Why not use an adjacency list? Some possible reasons:

- Backups! Years of bitter experience have gone into today's databases, and they are very good and reliable for mundane tasks like backing up your data. The data is too big to fit in memory, and database tables are a straightforward way to store it on disk.
- Database tables can be indexed on many keys. If I have a query like "Find all edges to or from user 379 with timestamp no older than 24 hours", and if the edges table has indexes for columns from_id and to_id and timestamp, then the query can be answered quickly. In an adjacency list representation, we'd just have to trawl through all the edges.

When you visit your home page, Facebook runs many queries on its social graph to populate the page. It needs to ensure that the common queries run very quickly, and so it has put considerable effort into indexing and caching.

Twitter also has a huge graph, with vertices for tweets and users, and edges for @mentions and follows. Like Facebook, it has optimized its graph database to give rapid answers to "broad but shallow" queries on the graph, such as "Who is following both the tweeter and the @mentioned user?"

TAO: Facebook's Distributed Data Store for the Social Graph (Bronson et al.) Usenix 2013.

<https://dev.twitter.com/rest/public/search>

Google, Spark

Google first came to fame because they had a better search engine than anyone else. The key idea, by Brin and Page when they were PhD students at Stanford, was this: a webpage is likely to be "good" if other "good" webpages link to it. They built a search engine which ranked results not just by how well they matched the search terms, but also by the "goodness" of the pages. They use the word *PageRank* rather than goodness, and the equation they used to define it is

$$PR_v = \frac{1-\delta}{|V|} + \delta \sum_{u:u \rightarrow v} \frac{PR_u}{\#edges\ out\ of\ u}$$

where $\delta=0.85$ is put in as a "dampening factor" that ensures the equations have a well-behaved unique solution.

How do we solve an equation like this, on a giant graph with one vertex for every webpage? Google said in 2013 that it indexes more than 30 trillion unique webpages, so the graph needs a cluster of machines to store it, and the computation has to be run on the cluster.

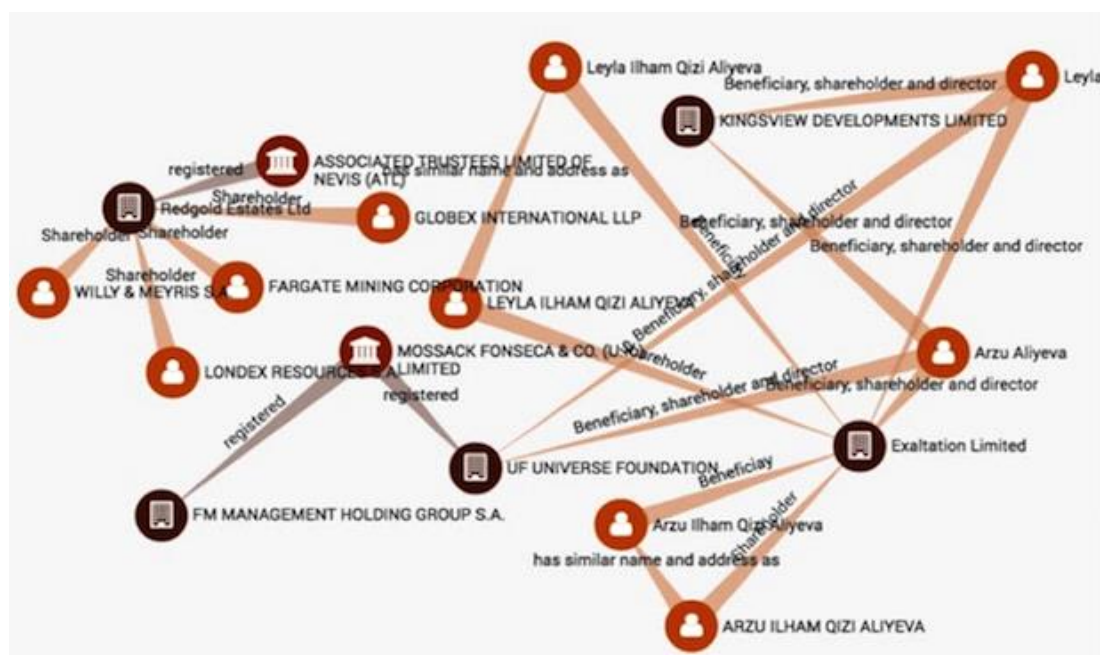
A popular platform for distributed computation (as of writing these notes in 2017) is called Spark. It has a library that is tailor-made for distributed computation over graphs, and friendly tutorials.

<http://spark.apache.org/docs/latest/graphx-programming-guide.html#pagerank>

neo4j

Late night conversations often devolve into questions like "What was that movie about submarines with the actor who was in that movie with that other actor that played the lead in Gone With the Wind?" This question can be thought of as searching for a subgraph with certain properties, in a graph with vertices for actors, movies, topics, etc.

Graph databases are databases that are specialized for storing graphs and for answering this sort of subgraph query. When the Panama Papers dataset was leaked, uncovering a complex network of offshore trusts, the journalists who got hold of it used a graph database called neo4j to help them understand it.



We don't yet know if it's possible to build a general-purpose graph database that works well on more than a single machine, which is why Facebook and Google have their own custom solutions.

<https://offshoreleaks.icij.org/pages/database>