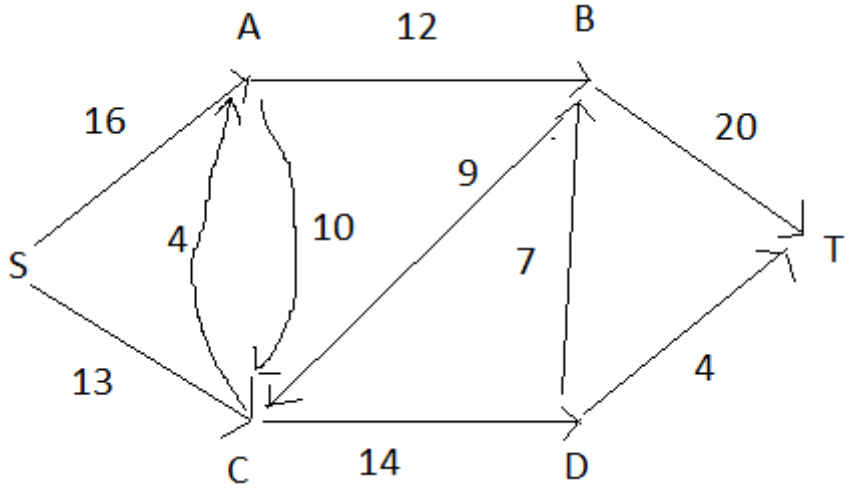
Network Flow

(Arup Guha, University of Central Florida)

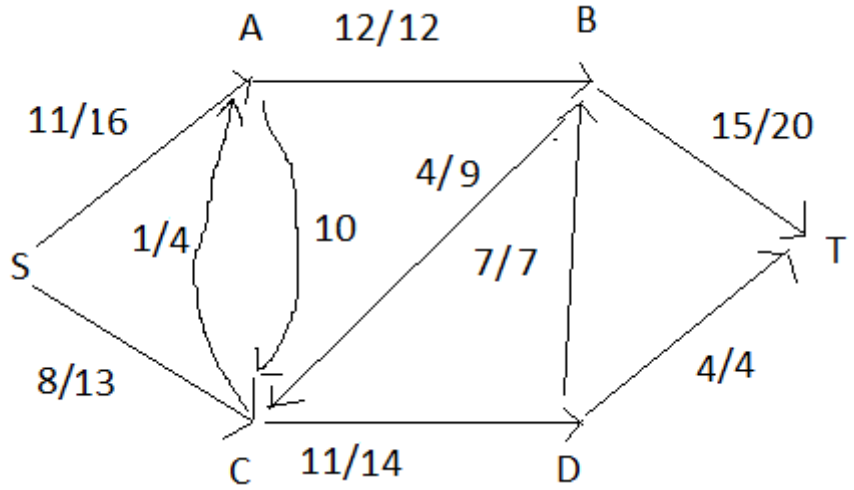
The network flow problem is as follows: Given a connected directed graph G with non-negative integer weights, (where each edge stands for the capacity of that edge), and two distinguished vertices, *s* and *t*, called the *source* and the *sink*, such that the source only has out-edges and the sink only has in-edges, find the maximum amount of some commodity that can flow through the network from source to sink.

One way to imagine the situation is imagining each edge as a pipe that allows a certain flow of a liquid per second. The source is where the liquid is pouring from, and the sink is where it ends up. Each edge weight specifies the maximal amount of liquid that can flow through that pipe per second. Given that information, what is the most liquid that can flow from source to sink per second, in the steady state?

Here is an example of an "empty" flow graph:



Here is an example of some flow going through the flow graph specified above:



In the picture on the previous page, the first number on each edge represents the current flow going through that edge and the second value represents the capacity of the edge, the maximum amount of flow that could be going through that edge. In this picture above, we have 19 units of flow going from the source (S) to the sink (T).

In order for the assignment of flows to be valid, we must have the sum of flow coming into a vertex equal to the flow coming out of a vertex, for each vertex in the graph except the source and the sink. *This is the conservation rule.* Also, each flow must be less than or equal to the capacity of the edge. *This is the capacity rule.* The *flow of the network* is defined as the flow from the source, or into the sink. For the situation above, the network flow is 19.

Cuts

A *cut* is a partition of vertices (V_s, V_t) such that $S \in V_s$ and $T \in V_t$. An edge that goes from u to v is a *forward edge* if $u \in V_s$ and $v \in V_t$. If the opposite is true, then it is *a backward edge*. Also let F equal the flow of a network.

For any cut, define the *flow across the cut* to be the sum of the flows of the forward edges minus the sum of the flows of the backwards edges.

The flow across any cut equals the flow of a network.

In our previous example, the flow was 19. Consider the flow across the cut $V_s = \{S, A, B\}$. We have three forward edges with flows of 8, 4, and 15 and two backward edges with flows of 1 and 7. Summing, we have 8+4+15-1-7 = 19, as desired. Basically, we have a total of 27 units going from "the side with S" to "the side with T" and 8 units going from "the side with T" to "the side with S", for a net positive flow of 19 units from "the side with S" to "the side with T."

Note the following observations:

1) The sum of the flows of all the vertices in V_s is the flow of the network because this sum is 0 for all non-source vertices in V_s, and equal to the flow at the source.

2) We need only to show that the flow NOT across V_s is 0. This is true simply because no edge in V_s leads into S. They all lead to each other. Thus, the net flow of all the edges within V_s is 0, because for two different vertices in V_s we add and subtract the same flow for an edge.

We can use this to show that the flow of a network can not exceed the capacity of any cut.

Simply put, our best case is if we don't have any edges with flow from "the side with T" to the "side with S." In this case, we are simply left with forward edges, each with a particular capacity. The sum of these capacities is the capacity of the cut, and an upper bound on the flow of that cut.

Finding the Maximum Flow in a Flow Network

We will use two ideas to help us determine whether or not a flow is maximum. Namely, we must show that absolutely no more flow could be added, no matter how we adjust each edge. The ideas we will look are *residual capacity* and *augmenting paths.*

The *residual capacity* of an edge from vertex *u* to vertex *v* is simply its unused capacity - the difference between its capacity and its flow in the direction of the edge. ***In the opposite direction (from v to u), this value is defined as the flow of the edge.*** The idea with this second statement is that if flow is going from *u* to *v*, there's no reason we can't "unsend" those units of flow. Mathematically, this means subtracting from the flow in the forward direction.

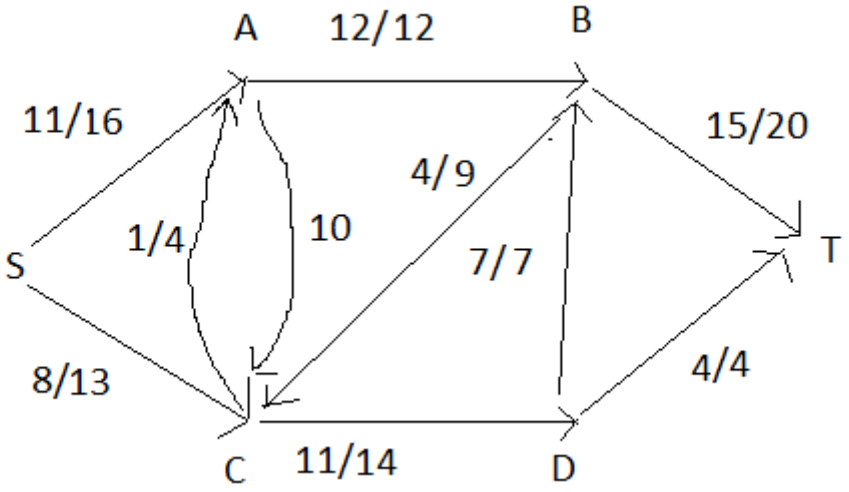
Also, the *residual capacity of a path* is defined as the minimum of the residual capacities of the edges on that path. This particular value is the maximum excess flow we can push down that particular path. Thus, an *augmenting path* is defined as one where you have a path from the source to the sink where every edge has a non-zero residual capacity. Namely, where for every edge on the path:

flow is less than capacity for each forward edge, AND
flow is greater than 0 for each backward edge.

In our example below, consider the following path:

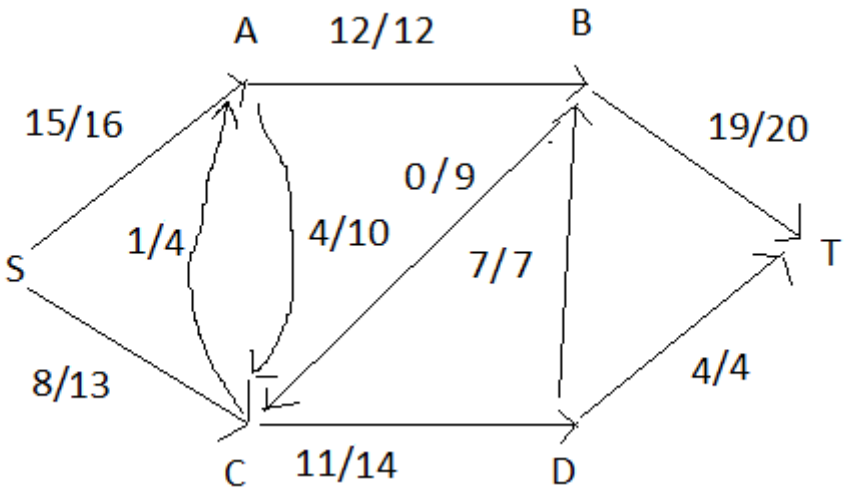
S → A → C → B → T

The forward edges on this path are SA, AC, and BT.
The backward edge is CB.



Thus, the minimum residual capacity of the path is 4, which is the limit given to us by CB, since we can "push back" at most 4 units of flow from vertex C to vertex B. From S to A, we could push forward up to 5 units of flow, from A to C we could push 10 units of flow,

and from B to T we could also push up to 5 units of flow. Thus, our limiting factor for this specific augmenting path is edge CB, where we can push back at most 4 units of flow. Now, let's augment this path:



Hopefully it should be clear that, if we find an augmenting path, adding the minimal residual capacity of that path to each edge on that path yields another valid flow. In essence, what we are doing is simply adding some fluid to a particular path in the network. ***On a backwards edge, we are actually taking away fluid.*** Intuitively, what using a backwards edge is doing is "fixing" a previous mistake. Before we added this last piece of flow, we had been sending 4 units of flow from B to C, and then C to D, where it was essentially getting stuck. So, by directing those 4 units from B to T (instead of through C), we were able to free up 4 more units to come from S to A to C to D. If one doesn't use any backwards edges in their search for augmenting paths, they lose the ability to "fix" errors that naturally occur when we guess suboptimal augmenting paths. Essentially, there's no way to know in advance what all the augmenting paths with no backward edges might look like. But, the algorithm allows us to correct our mistakes as we go, by taking backwards edges and this is the essence of how it works.

Now, we will show that if a flow network has no augmenting path, then it has a cut of maximum capacity.

If no augmenting path exists, then that means for all paths from source to sink, there is at least one edge without any residual capacity. Now, consider partitioning the vertices according to this rule: Place in V_s all vertices *v* that DO have augmenting paths from the source S. Then place all other vertices in V_t. (Note that the sink T must be in this set since there is no augmenting path to T.)

Now, consider all the edges with respect to this cut. Clearly, NONE of these edges have any residual capacity. If they did, then we would have added the vertex that they reach to the set V_s. Thus, for this particular cut, the flow at each forward edge is to its capacity and

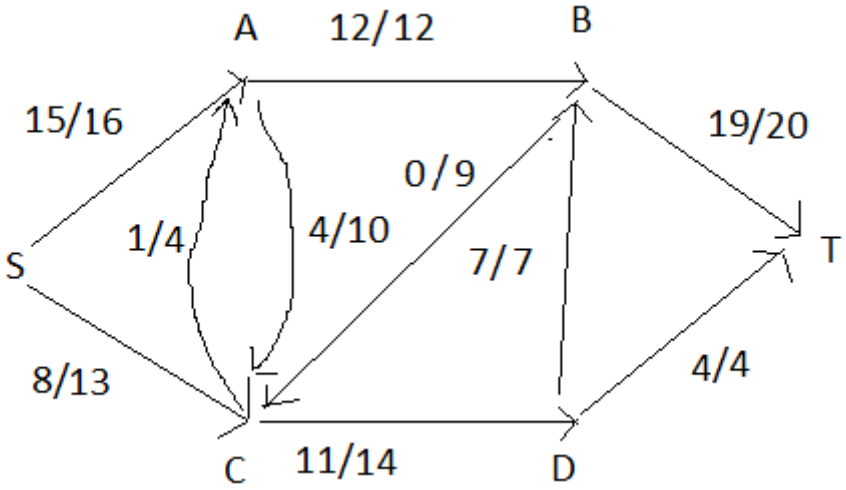
the flow at each backwards edge is 0. As we showed before, this situation represents the maximal flow possible.

Now, based upon the fact that augmenting paths can always be used to increase flow, and that the minimum capacity of a cut constrains the maximal flow, we can now claim the following:

The Max-Flow, Min-Cut Theorem: The value of the maximal flow in a flow network equals the value of the minimum cut.

With our example, we can see that there is no augmenting path from S to T. If there were, it would have to go through B and utilize edge BT. But all three edges leading to BT have no residual capacity.

Now, consider forming the set of vertices that do have augmenting paths from S. This set includes {S, A, C, D}. As mentioned before, it is impossible to reach B on an augmenting path since all edges leading to B are at full capacity already.



Now, if we look at this cut, we find that the maximal value of this cut is the sum of the capacities of the edges AB, DB, and DT which is 12+7+4 = 23***. (Notice that we did not add or subtract anything for edge BC with capacity 9, since it is a backwards edge.)***

Ford-Fulkerson Algorithm

The Ford-Fulkerson Algorithm is a basic consequent of the work above. In its simplest form, we do the following:

While there exists an augmenting path
    Add the appropriate flow to that augmenting path

We can check the existence of an augmenting path by doing any graph traversal on the network (with all full capacity edges removed.) ***Typically, DFS is used for Ford-Fulkerson.*** This graph, a subgraph with all edges of full capacity removed, is called a *residual graph*.

It is difficult to analyze the true running time of this algorithm because it is unclear exactly how many augmenting paths can be found in an arbitrary flow network. In the worst case, each augmenting path adds 1 to the flow of a network, and each search for an augmenting path takes O(E) time, where E is the number of edges in the graph. Thus, at worst-case, the algorithm takes O(|f|E) time, where |f| is the maximal flow of the network.

The Edmonds-Karp Algorithm

This algorithm is a variation on the Ford-Fulkerson method which is intended to increase the speed of the first algorithm. The idea is to try to choose good augmenting paths. In this algorithm, the augmenting path suggested is the augmenting path with the minimal number of edges. (We can find this using BFS, since this finds all paths of a certain length before moving on to longer paths.) The total number of iterations of the algorithm using this strategy is O(VE). Thus, its total running time when storing the graph as an adjacency matrix is $O(V^3E)$.

Applications of Network Flow

Obvious applications of network flow involve physical situations, such as a set of pipes moving water, or traffic in a network. For these situations, the translation of the input data into an appropriate graph is fairly intuitive.

However, a vast majority of the applications of network flow pertain to problems that don't seem to involve the physical movement of items through networks.

While we don't have time to look at all the types of applications of network flow, we will analyze one specific problem, bipartite matching, that can be solved using network flow, along with a few extra sample problems that can be solved with network flow.

Bipartite Matching

The bipartite matching problem is as follows:

Input: two mutually exclusive sets of equal size U and V, along with a list of ordered pairs of the form (u, v) where $u \in U$ and $v \in V$, indicating pairs of members, one of each set that can be "paired" together.

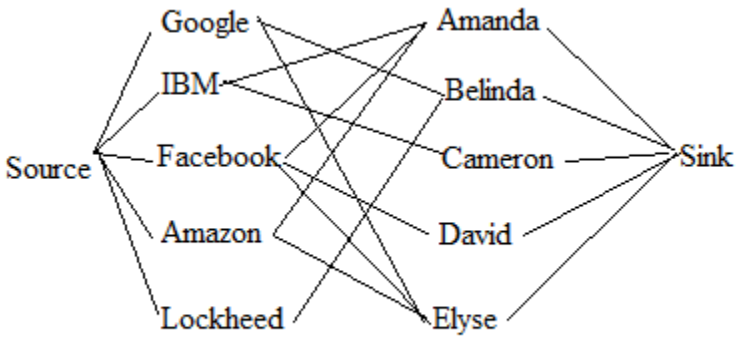
Output: True, if there exists a way to pair up each item in U with an item in V such that each item in both sets appears in exactly one pairing, and false otherwise.

Solution: Let n = the size of each input set. Set up a graph with $2n+2$ vertices. Create one vertex for each item in each set and add source and sink vertices. Add an edge from the source to each item in set U with capacity 1. Add an edge from each item in set V to the sink with capacity 1. Add an edge between each item in set U and set V that are in the set of ordered pairs with capacity 1. Calculate the maximal flow of this network. If the answer is n, then a complete matching exists, otherwise a complete matching doesn't exist. If you want the matching, keep track of each "edge" added during each iteration of the algorithm. (Note: some edges change as well.)

Instance of Bipartite Matching

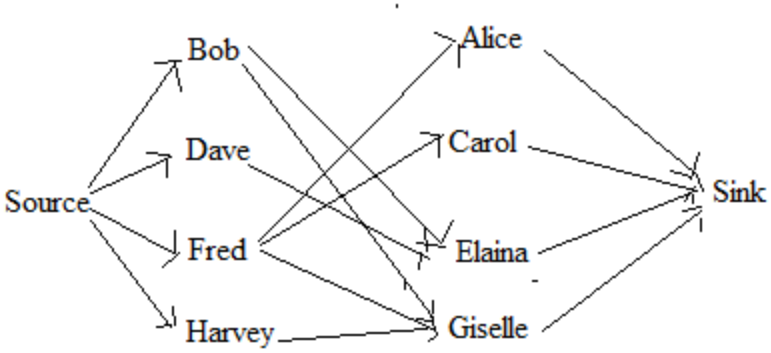
Set of companies:{Google,Microsoft, Facebook, Amazon, Lockheed}
Set of students: {Amanda, Belinda, Cameron, David, Elyse}

Set of offers: {(Google, Belinda), (Google, Elyse), (Microsoft, Amanda), (Microsoft, Cameron), (Facebook, Amanda), (Facebook, David), (Facebook, Elyse), (Amazon, Amanda), (Amazon, Elyse), (Lockheed, Belinda) }



Second Instance of Bipartite Matching

Set of boys: {Bob, Dave, Fred, Harvey}
Set of girls: {Alice, Carol, Elaina, Giselle}
Set of ordered pairs: { (Bob, Elaina), (Bob, Giselle), (Dave, Elaina), (Fred, Alice), (Fred, Carol), (Fred, Giselle), (Harvey, Giselle) }



Note: More generally, if we relax the restriction on the sizes of the two sets being equal, we can calculate the maximum number of pairs that we can form by running a Network Flow algorithm on a graph of the form described above.

Indirect use of bipartite matching

In some problems we might be asked to pick a maximum number of items so that no two items from two distinct sets "conflict". So, perhaps set A is Mrs. Weaver's class and set B is Mr. Gordon's class. We want to get the maximum number of kids in a single group from both classes so that no two kids hate each other. All the kids in each class like the other kids in their class, but they may hate some of the kids in the other class. For this type of scenario, it helps to solve the opposite problem:

Find the maximum number of pairs of kids, one from each class, such that each pair hates each other.

Note that in finding this maximum matching, which is equal to the maximum flow through the network created by linking edges between pairs of kids that hate each other, we are showing that for each of these pairs, at most one kid can be chosen. Since this matching is maximum, we CAN'T add a new pair to it. Thus, this means it's safe to add ALL the other kids not involved in any pair AND then add one person from each of these pairs. Thus, the total number of students we can choose is equal to the total number of kids in both classes minus the maximum number of pairs we can create who hate each other.

Cow Steeplechase Problem (from USACO) – Utilizes bipartite matching indirectly

Given a list of horizontal line segments (none of which intersect each other) and vertical line segments (none of which intersect each other), calculate the minimum number of line segments that must be removed, so that no two lines intersect each other, or alternatively, the most number of line segments that mutually don't intersect one another.

Solution

We can create a bipartite matching solution. Our goal is to match horizontal line segments to vertical line segments in such a way that each pair intersects. We know that if we have a set of these intersecting pairs, at the very least, one item in each pair must be removed. Thus, what we really want is the maximum matching. Once we have this (say there are 7 matching pairs in our maximal matching), then we have proof that 7 of the segments must be removed. No other matching forces us to remove more. Thus, that is how many we are forced to remove to create no intersecting line segments. Alternatively, we can calculate the maximum set of segments we can have without any two intersecting by taking the total number and subtracting out this maximum matching.

Network Flow for other matching problems

In other matching problems, we aren't always making "one to one" matchings. Instead, we might be matching several items in several groups to other items grouped together differently. In the two example problems shown below, the capacities of our edges are not always one, and these capacities indicate the number of items from one set that we can match elsewhere. These sorts of problems take many different forms, so it's best just to understand the basic structure of setting up a flow graph and be flexible to edit that structure for each new problem that you see.

Example Problem: Grand Dinner Problem (from ACMUVA)

N teams attend a dinner. Team i has p_i members. There are M tables at the dinner, with M $\geq$ N. Table i has q_i chairs. We wish to seat all teams such that no two team members are at the same table, so that we have maximum students getting to meet members of other teams. Can we do so?

Solution using Network Flow

Create a flow network with N + M + 2 vertices. Create one vertex for each team and one for each table. Create extra source and sink vertices. Create edges from the source to each team with a capacity of p_i. Create edges from each table vertex to the sink vertex with capacity q_i. Finally, add edges from each team to each table, with capacity 1, since each team can provide at most one person per table. Run the network flow algorithm. If the maximal flow equals the sum of the number of team members, the seating can be done. Otherwise, it cannot be.

Museum Guard Problem: 2009 South East Regional

A museum employs guards that work in 30 minute shifts: 12:00 AM - 12:30 AM, 12:30 AM - 1:00 AM, ..., 11:30 PM - 12:00 AM. Each guard has a list of times he/she can NOT work. (For example, a particular guard might not be able to work from 3:30 AM to 7:30 AM and from 4:29 PM to 8:01 PM. In this case, the guard could work the 3:00 AM - 3:30 AM and 7:30 AM - 8:00 AM shifts, but not the 4:00 PM - 4:30 PM shift or 8:00 PM - 8:30 PM shift.) Furthermore, each guard has a maximum number of hours they can work in a single day. Determine the maximum number of guards we can have scheduled to cover each shift without violating any of the constraints.

Solution using a flow network
As usual, we create an extra source and sink.

Each guard will be a vertex in the flow network. From the source, we connect an edge to each guard with an integer representing the maximum number of shifts that guard can work.

Each shift (there are 48) will become a node in the flow network. Add an edge with capacity 1 from each guard to each shift where the guard is able to work that shift.

Finally, if we are to set the capacities from all the shifts to the sink to 1, and we run the network flow algorithm and obtain 48, we know that we can post 1 guard for each slot. We can re-run the algorithm with all of these capacities set to 2 and see if the resulting flow is 96 or not. If so, we move onto 3, and so forth. Since there are at most 50 guards, our answer will never exceed 50 and this will run in time.

A more clever approach involves a binary search. Try capacities at 25 for each of these edges. If this doesn't work, go to 12, If it does, go to 37, and so on. Basically, rather than checking if 1 works, then 2, then 3, etc. a binary search will hone in on the answer a bit more quickly, especially in the cases that the answer is closer to 50.