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:

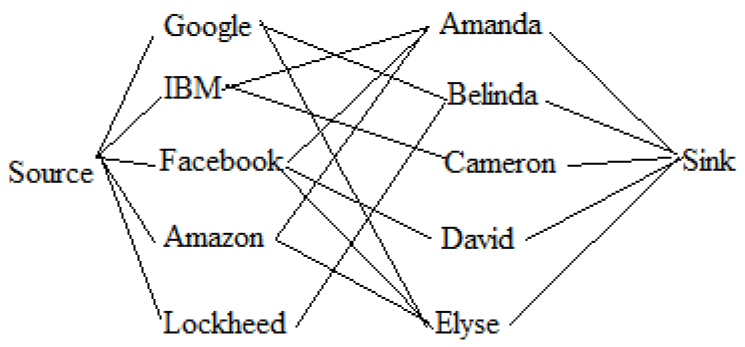
Input: two mutually exclusive sets of equal size U and V, along with a list of ordered pairs of the form (u, v) where $u \in U$ and $v \in V$, indicating pairs of members, one of each set that can be "paired" together.

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.

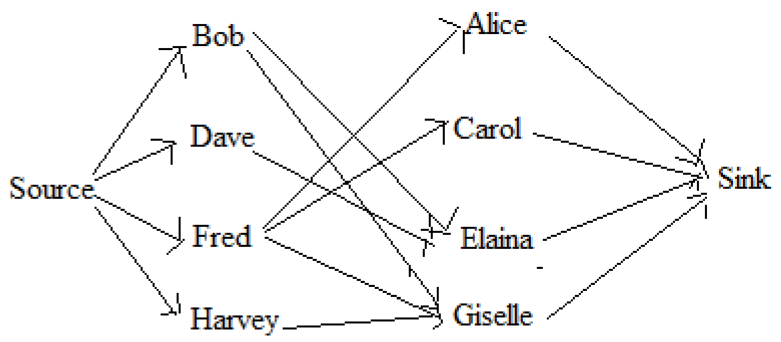
Solution: Let n = the size of each input set. Set up a graph with 2n+2 vertices. Create one vertex for each item in each set and add source and sink vertices. Add an edge from the source to each item in set U with capacity 1. Add an edge from each item in set V to the sink with capacity 1. Add an edge 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.)

<u>Instance of Bipartite Matching</u>
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) }



<u>Second Instance of Bipartite Matching</u>
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.

<u>Network Flow for other matching problems</u>

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.

<u>Example Problem: Grand Dinner Problem (from ACMUVA)</u>

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

<u>Solution using Network Flow</u>

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 t_i. Create edges from each table vertex to the sink vertex with capacity s_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 can not be.

<u>**Indirect use of bipartite matching**</u>

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.

<u>**Cow Steeplechase Problem (from USACO) – Utilizes bipartite matching indirectly**</u>

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.

<u>Solution</u>

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 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.

Max-Flow = Min-Cut

Sometimes it's easier to characterize the solution to a problem as the minimum cut of a flow network. Because the minimum cut of a flow network equals its maximum flow, we can solve these problems by calculating the maximum flow of the desired flow network.

Consider the Problem E Funhouse from the 2012 South East Regional Division 1 Problem Set:

You are given a topology of a funhouse of connected rooms, some of the rooms have entrances where people can enter the funhouse. Other rooms have exits. Some pairs of rooms are connected with a door. There may be many paths that visitors take, but we want to guarantee that no matter what path a visitor takes, they visit at least one room with shaker floors. The cost of installing a shaker floor for each room is known and our goal is to minimize the cost of the shaker floors installed while guaranteeing that all guests experience at least one room with a shaker floor.

We set up the flow network as follows:

We create a source that connects to each entrance, and a sink such that each exit connects to the sink. We also draw edges between all pairs of rooms that are connected with a door. The capacity of an edge will be the cost of installing a shaker floor in the destination vertex of that edge. The capacity of the edges going to the sink will be unbounded.

Notice that the minimum cut of this graph represents the subset of edges across a cut between any entrance and any exit. But, because the minimum cut equals the maximum flow, we can just set up the graph and calculate the maximum flow through the graph to solve the problem.

Note: I've simplified the problem to isolate the network flow portion of the problem. The real problem had a harder geometry problem embedded into it. None of the connections or weights described were given in the input. Instead they had to be calculated based on a set of line segments that defined all of the rooms, entrances, exits and doors.

Museum Guard Problem: 2009 SER - Use of Binary Search with Network Flow

A museum employs guards that work in 30 minute shifts: 12am - 12:30am, 12:30am-1am, ..., 11:30pm-12 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 7:30am and from 4:29pm to 8:01pm. In this case, the guard could work the 3am-3:30am and 7:30am-8am shifts, but not the 4pm - 4:30pm shift or 8pm-8:30pm 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 at 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.

Image Segmentation Type Problems (2013 South Central: Drop Zone)

In an image segmentation problem, you're given an image (usually you can think of it as a 2d grid of values of some sort) and you want to determine the fewest number of "barricades" you need to place to enclose a particular region or item. For example, consider being given the following two D grid:

```
XXX..XXX
XXX..XXX
.....XXX
XXX..XXX
XDDDD.XX
XDDDD...
XXXXXXXX
```

And being asked to place either horizontal or vertical boundaries to make it impossible to reach and of the spots marked 'D' traveling from the outside via the dots(.). The 'X' character represents an illegal square. The goal is to place the fewest number of these boundaries. For this picture, three boundaries will suffice:

```
XXX..XXX
XXX..XXX
.....XXX
XXX--XXX
XDDDD.XX
XDDDD.|.
XXXXXXXX
```

We can model this as a network flow problem! We place edges between each vertex that we can travel between. We place edges from the source to each entrance vertex. Finally, we place edges from each vertex corresponding to a square with a D to the sink. The capacities of most of the edges are 1, except for edges from the source and edges leading to the sink. The former should equal the number of boundaries the square has with the outside (representing potential sides to block) and the latter should just be large.

The maximum flow through this network represents the maximum number of independent paths that can be formed from the outside to any of the characters 'D'. This is also the minimum cut, the precise item for which we needed an answer.

Repeated Use of Flow to find the lexicographically first maximum matching (2012 South East Regional Problem F: A Terribly Grimm Problem)

The full problem is as follows:

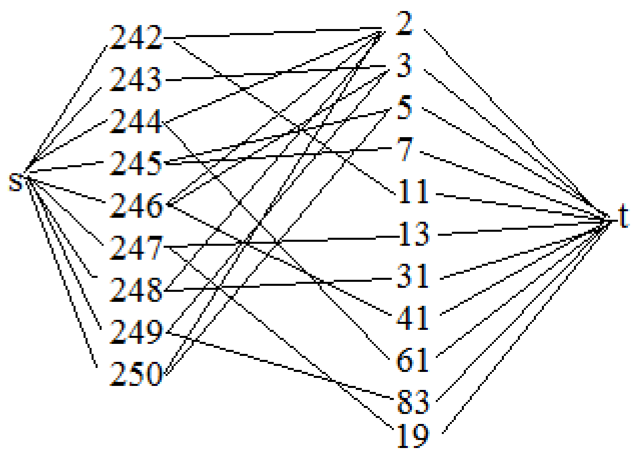
Grimm's conjecture states that to each element of a set of consecutive composite numbers one can assign a distinct prime that divides it.

For example, for the range 242 to 250, one can assign distinct primes as follows:

242	243	244	245	246	247	248	249	250
2	3	61	7	41	13	31	83	5

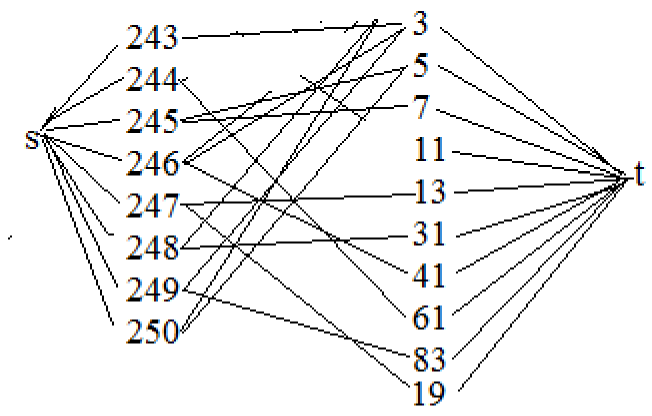
Given the lower and upper bounds of a sequence of composite numbers, find a distinct prime for each. If there is more than one such assignment, output the one with the smallest first prime. If there is still more than one, output the one with the smallest second prime, and so on.

The bounds of the problem were such that the lower and upper bounds were as large as 10^{10}, but that all numbers in the range were composite. If we wanted to know whether or not any solution existed at all, we could set up a bipartite matching between the composite numbers on the top row and all possible prime numbers, on the bottom row. Here is what the graph would look like for our example:



All of the capacities would be 1. Basically, if the flow of this network is equal to 9 (250 - 242 + 1), then that means a solution exists. As the problem states, this will always be the case for any range of consecutive composite integers up to 10^{10}. Thus, we know that if we were to run flow on the induced network for any of these cases, our answer would indeed equal the number of values in the composite range.

Nonetheless, the flow characterization of the problem is still useful to obtain the lexicographically first solution. We proceed as follows. We know that 242 MUST be linked to either 2 or 11. Of the two options, we prefer 2. Thus, let's force 242 to map to 2 and remove both vertices from the graph, giving us this graph:



Now, unlike the previous graph, it's not guaranteed that this graph has a maximum flow of 8 (250 - 243 + 1). Thus, we find the maximum flow through the network. If it's 8, then we know there exists a valid solution matching 242 to 2 and that we must fix this matching. Alternatively, if the maximum flow of this network is less than 8, we simply try our next matching for 242. We are guaranteed that at least one of the possible matchings of 242 will lead to a subgraph that has a maximum flow of 8. We simply match 242 to the smallest of these values, by checking each subgraph in order.

Once we match 242, we continue to match each subsequent number in the same fashion. (Note: In code, rather than forming a new graph topography, it may be easier to simply put 0 capacities on each composite-prime pair edge from source and to sink that have already been matched.)

This general idea works for any problem where we desire the lexicographically first maximum bipartite matching.

Maximal Anti-Chain (2015 SER D1: Airport Checker)

The Airports problem from this year's regionals is as follows:

An airline company offers flights out of ***n*** airports. The flight time between any given pair of airports is known, but may differ on direction due to things like wind or geography. Upon landing at a given airport, a plane must be inspected before it can be flown again. This inspection time is dependent on the airport at which the inspection is taking place.

Given a set of ***m*** flights that the airline company must realize, determine the minimum number of planes that the company needs to purchase. The airline may add unscheduled flights to move the airplanes around if that would reduce the total number of planes needed.

The maximal anti-chain of a set of items is the longest sequence of events, e_1, e_2, e_3, … , e_k such that no two elements in the sequence are comparable to each other. For this problem, two flights are "comparable" if the same plane can fly both flights. Thus, we desire the largest set of flights such that no two can be flown by the same plane. If we were able to obtain this set, we'd have proof that we would need at least that many planes. Dilworth's theorem tells us that whatever the size of this largest set is, we can partition all of the items into exactly that many sets where each consecutive pair of items in an ordered sequence are comparable. Thus, for this problem, if we can find the maximal set of flights such that no two can be flown by the same plane, we'll have also solved the minimization problem of the finding the fewest number of planes necessary to fly all of the flights.

To find the desired maximum, we can set up a flow network. We create a source, sink and two nodes for each flight. We connect the first node for flight i to the second node for flight j if and only if the same plane can first fly flight i and then follow it with flight j. Each unit of flow we send through this graph represents the reduction, by one, of the planes needed to fly all the flights. Namely, if the flow were 0, we would need a plane for every flight. But, as we add an edge to this graph, we can remove one plane and use the plane that flew flight i to now also fly flight j. This logic continues for each unit of flow added in the graph.

It follows that the final result is simply the number of flights minus the maximum flow of this graph.

Baseball Elimination Problem

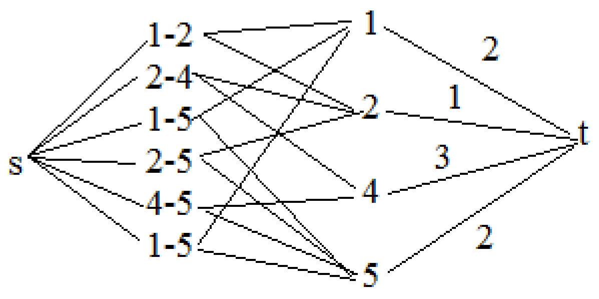
Imagine the situation where we have mostly completed a baseball season and have a few games to play. We want to know whether or not our beloved team can win their division! A simple calculation one can make is see the number of wins the current team has and see if we were to win our remaining games, if we'd reach that number. This is a good approximation for whether or not we have a shot, but it's not 100% accurate. In some cases, while we might have enough games to play to equal the number of wins of the current leader, it may turn out that no matter what the outcome of other games are, some team will end up winning some more games preventing your team from claiming the division title.

Let's say we know each team's current record and that we know each pair of games that remains. For simplicity, assume that the only outcomes of a game are a won or a loss.

It's easy for us to calculate our maximal record; it's simply equal to the current number of wins plus the number of games we have left to play. For simplicity, let's just assume it's good enough if we tie for the best record in the division. This means that all of the other teams can NOT exceed this number of wins.

Our network flow graph will have a source, a sink, a node for each game remaining, and a node for each team. The source will connect to each game with a capacity of 1, representing the one game to play. Each game will connect to the two teams in question, each with a capacity of 1. Thus, the unit of flow that starts at the source and goes to a game must either go to one team or the other. Finally, each team links to the sink with a capacity equal to the maximum number of games our beloved team can win minus their current number of wins. (If we wanted to win the division outright, we would set this capacity to be one less for all other teams…) Of course, we don't put our team in the flow network! We automatically assign wins for each of our team's games!!!

For example, imagine that our team is team 3 and there are 6 remaining games in the season (except for ours). Also, imagine that we want team 1 to win at most 2 of these games, team 2 to win at most 1 of these games, team 4 to win at most 3 of these games and team 5 to win at most 2 of these games. (All edges not labeled have a capacity of 1.) We would get the following flow graph:


s
1-2
2-4
1-5
2-5
4-5
1-5
1
2
4
5
t
2
1
3
2