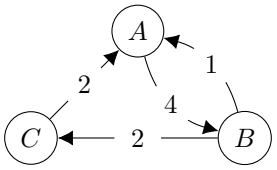
Algorithms on Graphs: Flow

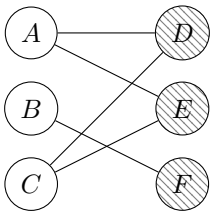
Originally prepared by Mark
Adapted by Curtis and Sunny

Part 1: Review of Graphs

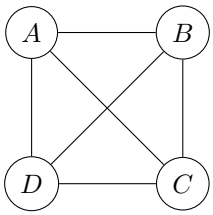
Definition 1:
A *graph* is a set of vertices and a set of edges where any two distinct vertices are either not connected by an edge or connected by exactly one edge. A *directed graph* is a graph where edges have direction. In such a graph, edges (A, B) and (B, A) are distinct. A *weighted graph* is a graph that features weights on its edges.
A weighted directed graph is shown below.



Definition 2:
We say a graph is *bipartite* if its nodes can be split into two groups[1] L and R, where no two nodes in the same group are connected by an edge:



Here is an example of a non-bipartite graph:



[1] A fancy mathematical term for this is *partitioned into two nonempty sets.*

Part 2: Network Flow

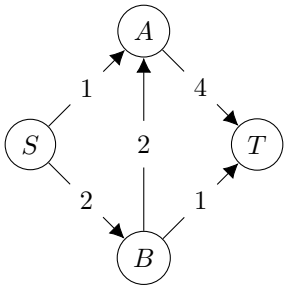
Networks

A **network** is a graph where the nodes and edges have certain useful attributes. For our purposes, the nodes have distinct labels and the edges have weights and directons. A *simple network* is a network in which no two vertices are joined by more than one edge and there are no loops.
Say have a network: a sequence of pipes, a set of cities and highways, an electrical circuit, etc.

We can draw this network as a directed weighted graph. If we take a transportation network, for example, we can crudely model this by taking edges to represent highways[2] and nodes to be cities. There are a few conditions for a valid network graph:

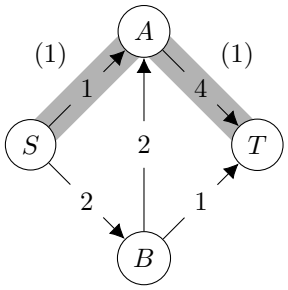
- The weight of each edge represents its capacity, e.g, the number of lanes in the highway going the direction of the edge.
- Edge capacities are always positive integers.[3]
- Node S is a *source*: it produces flow.
- Node T is a *sink*: it consumes flow.
- All other nodes *conserve* flow. The sum of flow coming in must equal the sum of flow going out.

Here is an example of such a graph:



Flow

In our city example, traffic represents *flow*. Let's send one unit of traffic along the topmost highway:



There are a few things to notice here:

- Highlighted edges carry one unit of traffic or our flow.
- Numbers in parentheses tell us how much flow each edge carries.
- The flow along an edge is always positive or zero.
- Flow comes from S and goes towards T.
- Flow is conserved: all flow produced by S enters T.

The *magnitude* of a flow is the number of "flow-units" that go from S to T.
We are interested in the *maximum flow* through this network: what is the greatest amount of flow we can get from S to T?

Problem 1:

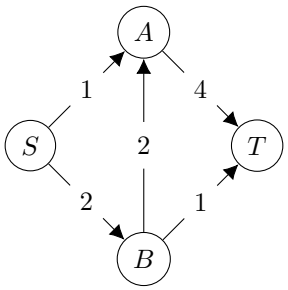
What is the magnitude of the flow above?

---

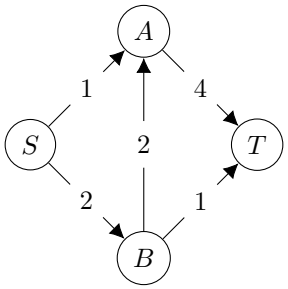
[2] In practice, highways tend to have equal capacity in both directions, but for our purposes, this network consists of many one-way highways.

[3] An edge with capacity zero is equivalent to an edge that does not exist; An edge with negative capacity is equivalent to an edge in the opposite direction. This is an extension of our definition of an edge being labeled by zero or a negative number.

Problem 2:
Find a flow with magnitude 2 on the graph below.

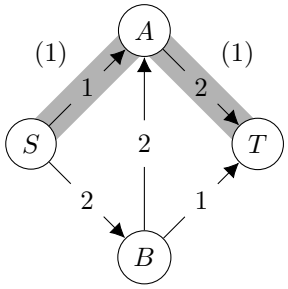


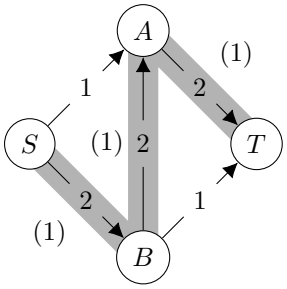
Problem 3:
Find a maximal flow on the graph below.
Hint: The total capacity coming out of S is 3, so any flow must have magnitude ≤ 3.



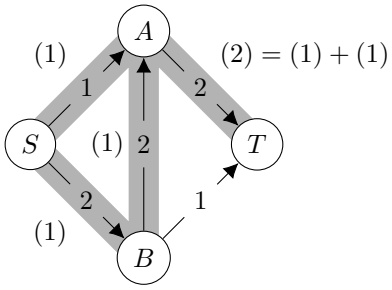
Part 3: Combining Flows

It is fairly easy to combine two flows on a graph. All we need to do is add the flows along each edge. Consider the following flows:





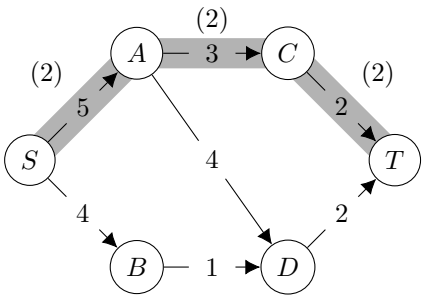
Combining these, we get the following:

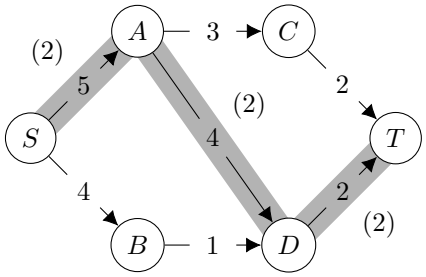


When adding flows, we must respect edge capacities.

For example, we could not add these graphs if the magnitude of flow in the right graph above was 2, since the capacity of the top-right edge is 2, and $2 + 1 > 2$.

Problem 4:
Combine the flows below, ensuring that the flow along each edge remains within capacity.



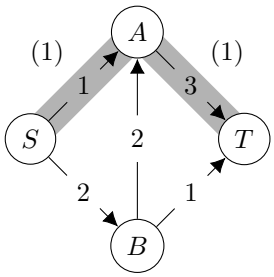


Part 4: Residual Graphs

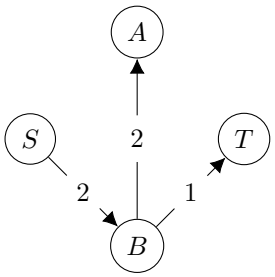
It is hard to find a maximum flow for a large network by hand.
We need to create an algorithm to accomplish this task.

The first thing we'll need is the notion of a *residual graph*. Let's construct a residual graph given a network and flow as follows:

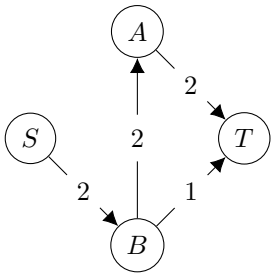
We'll start with the following network and flow:



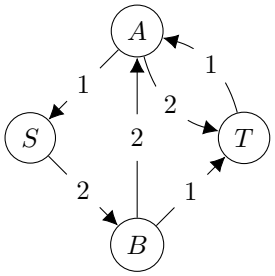
First, we'll copy all nodes and "unused" edges:



Then, we'll add the unused capacity of "used" edges: (Note that $3 - 1 = 2$)



Finally, we'll add "used" capacity as edges in the opposite direction:



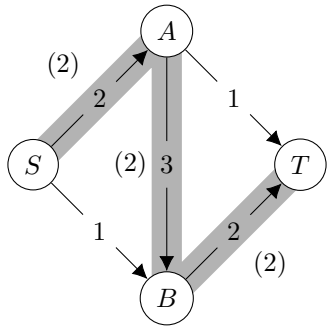
This graph is the residual of the original flow.

You can think of the residual graph as a "list of possible changes" to the original flow.
There are two ways we can change a flow:

- We can add flow along a path
- We can remove flow along another path

A residual graph captures both of these actions, showing us where we can add flow (forward edges) and where we can remove it (reverse edges). Note that "removing" flow along an edge is equivalent to adding flow in the opposite direction.

Problem 5:
Construct the residual of this flow.



Problem 6:
Is the flow in Problem 5 maximal?
If it isn't, find a maximal flow.
Hint: Look at the residual graph. Can we add flow along another path?

Problem 7:
(Bonus) Prove the following:

- A maximal flow exists in every network with integral edge weights.
- Every edge in this flow carries an integral amount of flow.

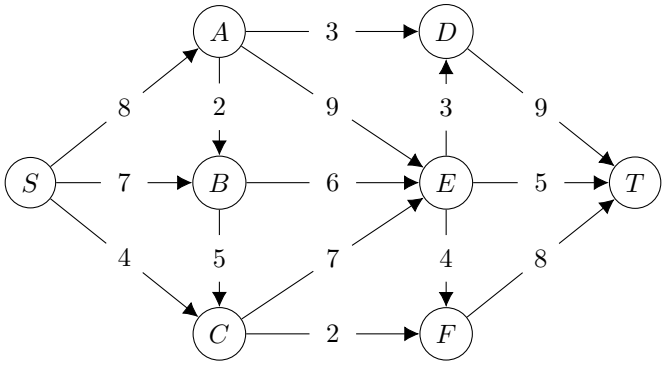
Part 5: The Ford-Fulkerson Algorithm

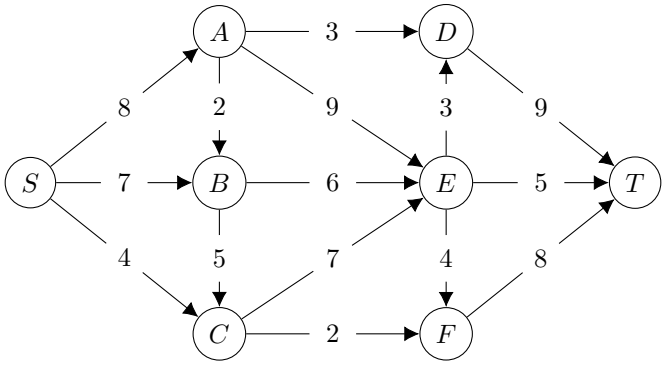
Problem 8:
We now have all the tools we need to construct an algorithm that finds a maximal flow. But first, take a look at the graph below. Just using your intuition, what do you think the maximal flow is?

Okay, now here's the Ford-Fulkerson algorithm:

```
00 Take a weighted directed graph G.
01 Find any flow F in G
02 Calculate R, the residual of F.
03     If S and T are not connected in R, F is a maximal flow. HALT.
04 Otherwise, find another flow F_0 in R.
05 Add F_0 to F
06 GOTO 02
```

Problem 9:
Run the Ford-Fulkerson algorithm on the following graph to determine the maximal flow. There is extra space on the next page.



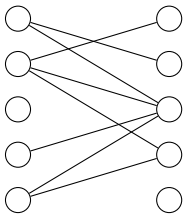

A
D
S
B
E
T
C
F
3
8
2
9
3
9
7
6
5
4
5
7
4
8
2

Problem 10:
You are given a large network. How can you quickly find an upper bound for the number of iterations the Ford-Fulkerson algorithm will need to find a maximum flow? Is this the least upper bound? If not, try to find it.

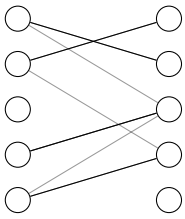
Part 6: Applications (Bonus)

Problem 11: Maximum Cardinality Matching
A *matching* is a subset of edges in a bipartite graph. Nodes in a matching must not have more than one edge connected to them.
A matching is *maximal* if it has more edges than any other matching.

Initial Graph



Maximal Matching

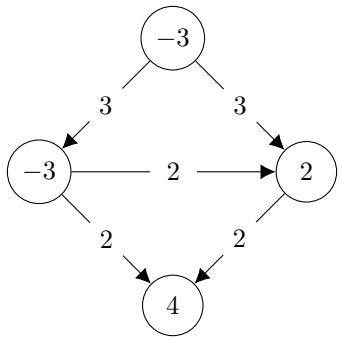


Create an algorithm that finds a maximal matching in any bipartite graph.
Find an upper bound for its runtime.
Hint: Can you modify an algorithm we already know?

Problem 12: Supply and Demand
Say we have a network of cities and power stations. Stations produce power; cities consume it. Each station produces a limited amount of power, and each city has limited demand.

We can represent this power grid as a graph, with cities and stations as nodes and transmission lines as edges.

A simple example is below. There are two cities (2 and 4) and two stations (both -3).
We'll represent station capacity with a negative number, since they *consume* a negative amount of energy.



We'd like to know if there exists a *feasible circulation* in this network—that is, can we supply our cities with the energy they need without exceeding the capacity of our power plants and transmission lines?

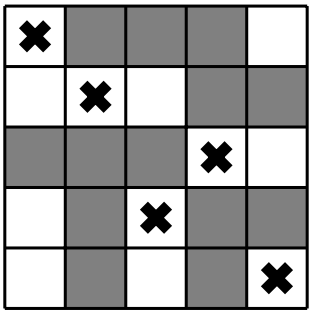
Your job: Devise an algorithm that solve this problem.

Bonus: Say certain edges have a lower bound on their capacity, meaning that we must send *at least* that much flow down the edge. Modify your algorithm to account for these additional constraints.

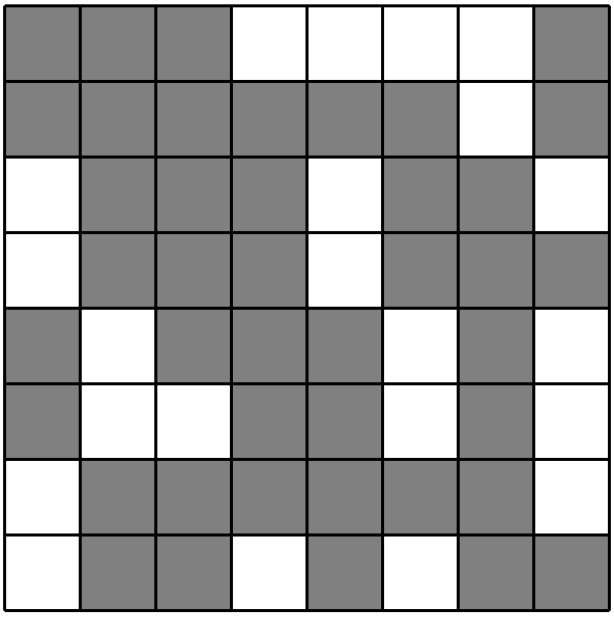
Part 7: Crosses

You are given an $n \times n$ grid. Some of its squares are white, some are gray. Your goal is to place n crosses on white cells so that each row and each column contains exactly one cross.

Here is an example of such a grid, including a possible solution.



Problem 13:
Find a solution for the following grid.



Problem 14:
Turn this into a network flow problem that can be solved with the Ford-Fulkerson algorithm.

Part 8: Cheese Network

My latest abomination. Play this with your classmates:

- Designate one player as MAX and another as MIN. MAX always makes the first move.
- Draw a source node somewhere on your page.
- Take turns adding at most one node and one unweighted directed edge to the network. You may do just one or neither. The players must ensure the following at the end of each turn:
  - The network is connected.
  - The network is *simple*.
  - Each node has degree at most 4.
- MAX gets to go both first and last, so after MAX has gone 7 times and MIN 6 times, the network is complete.
- MIN gets to choose one node to be the cheese.
- Place a mouse at the source. You can use a random object like a ball of eraser dust, or just keep track with your pencils.
- Take turns moving the mouse. On each turn, you may move the mouse along the direction of an adjacent edge to another node. MAX wants to get the mouse to the cheese, MIN must prevent this at all costs.
- The game ends either when the mouse arrives at the cheese or it gets stuck somehow and MAX gives up.
- MAX is scored on how many turns it took to reach the cheese, which may be ∞. Players should switch roles and compete for the lowest score.

I encourage you to experiment with variations on these rules. Good luck!