Algorithms on Graphs: Flow

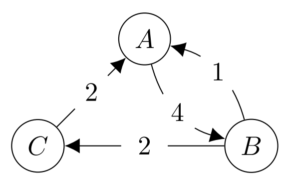
Prepared by Mark on January 24, 2025

Instructor's Handout

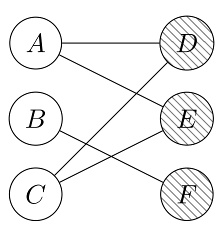
This file contains solutions and notes.
Compile with the "nosolutions" flag before distributing.
Click [here] for the latest version of this handout.

Part 1: Review

Definition 1:
A *graph* consists of a set of *nodes* $\{A, B, ...\}$ and a set of edges $\{(A, B), (A, C), ...\}$ connecting them.
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 L and R, where no two nodes in the same group are connected by an edge:



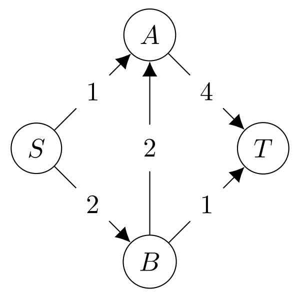
Part 2: Network Flow

Networks
Say have a network: a sequence of pipes, a set of cities and highways, an electrical circuit, etc.

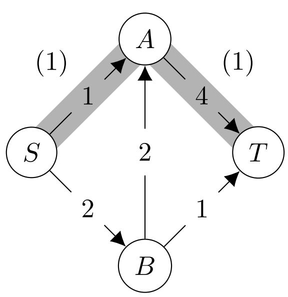
We can draw this network as a directed weighted graph. If we take a transportation network, for example, edges will represent highways and nodes will be cities. There are a few conditions for a valid network graph:

- The weight of each edge represents its capacity, e.g, the number of lanes in the highway.
- Edge capacities are always positive integers.[1]
- 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:

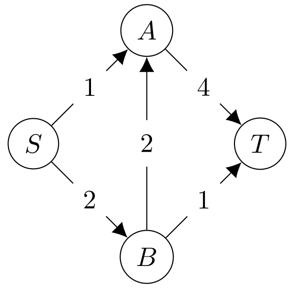
- Highlighted edges carry flow.
- Numbers in parentheses tell us how much flow each edge carries.
- The flow along an edge is always positive or zero.
- Flow comes from S and goes towards T.
- Flow is conserved: all flow produced by S enters T.

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?

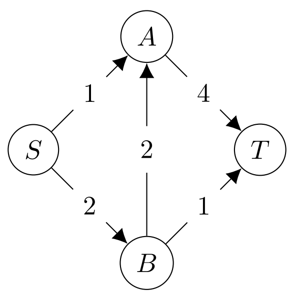
Problem 1:
What is the magnitude of the flow above?

[1] 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.

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

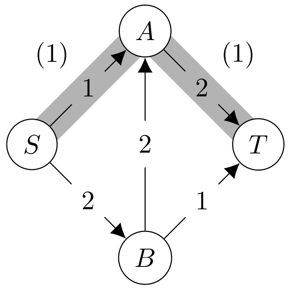


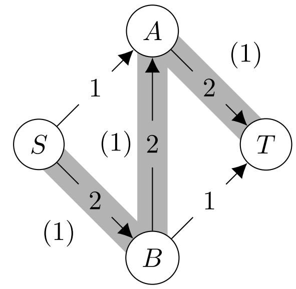
Problem 3:
Find a maximal flow on the graph below.
Hint: The total capacity coming out of S is 3, so any flow must have magnitude ≤ 3.



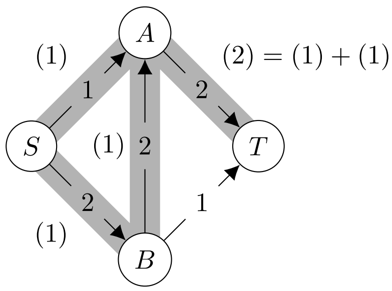
Part 3: Combining Flows

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:





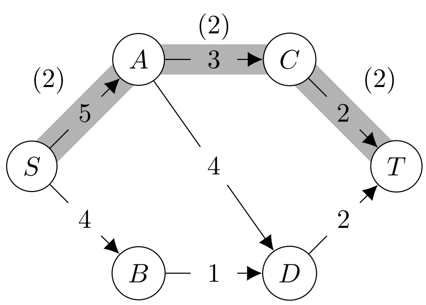
Combining these, we get the following:

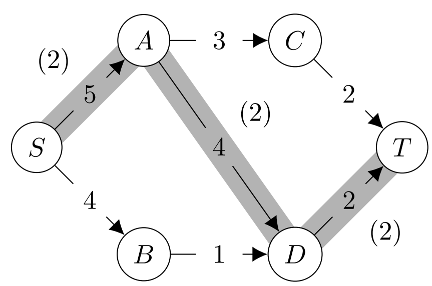


When adding flows, we must respect edge capacities.

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.



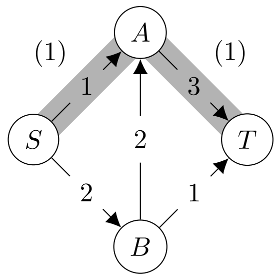


Part 4: Residual Graphs

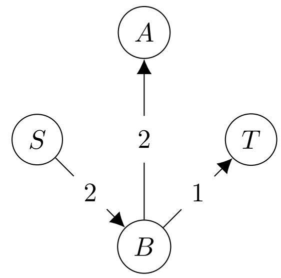
It is hard to find a maximum flow for a large network by hand.
We need to create an algorithm to accomplish this task.

The first thing we'll need is the notion of a *residual graph*.

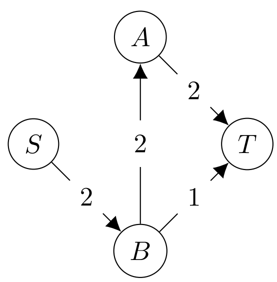
We'll start with the following network and flow:



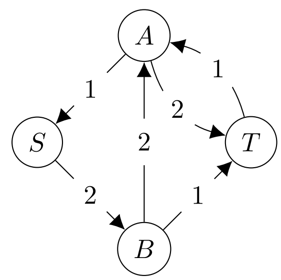
First, we'll copy all nodes and "unused" edges:



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:



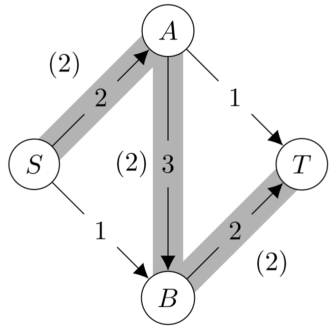
This graph is the residual of the original flow.

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:

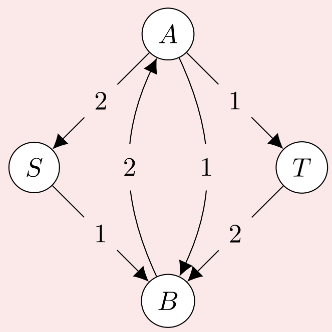
- We can add flow along a path
- We can remove flow along another path

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.



Solution



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:

Show that...

A: A maximal flow exists in every network with integral edge weights.

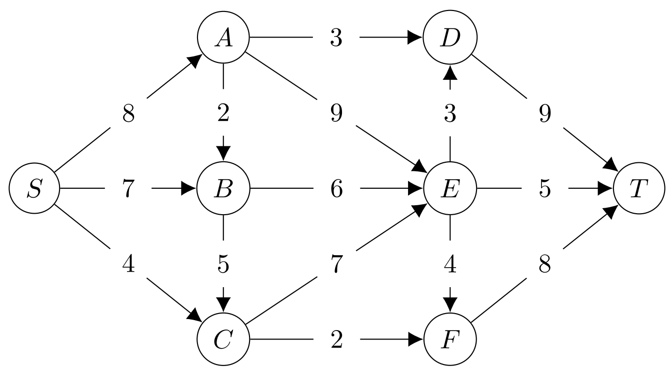
B: Every edge in this flow carries an integral amount of flow

Part 5: The Ford-Fulkerson Algorithm

We now have all the tools we need to construct an algorithm that finds a maximal flow.
It works as follows:

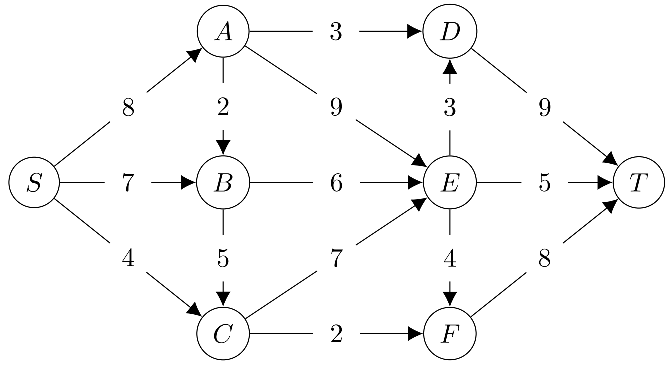
```
00 Take a weighted directed graph G.
01 Find any flow F in G
02 Calculate R, the residual of F.
03     If S and T are not connected in R, F is a maximal flow. HALT.
04 Otherwise, find another flow F0 in R.
05 Add F0 to F
06 GOTO 02
```

Problem 8:
Run the Ford-Fulkerson algorithm on the following graph.
There is extra space on the next page.



Solution

The maximum flow is 17.


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

Problem 9:
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?

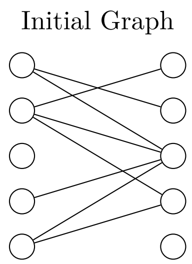
Solution

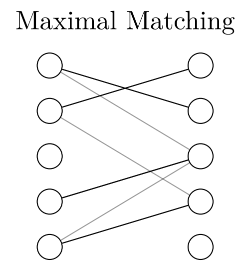
Each iteration adds at least one unit of flow. So, we will find a maximum flow in at most $\min(\text{flow out of } S,\ \text{flow into } T)$ iterations.

A simpler answer could only count the flow on S.

Part 6: Applications

Problem 10: 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.





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?

Solution

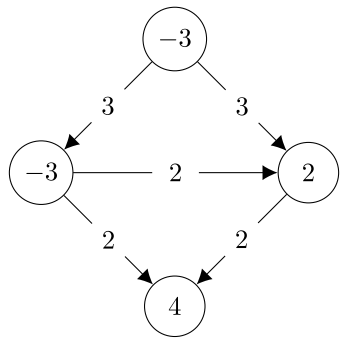
Turn this into a maximum flow problem and use FF.
Connect a node S to all nodes in the left group and a node T to all nodes in the right group.
All edges have capacity 1.

Just like FF, this algorithm will take at most $\min(\#\text{ left nodes}, \#\text{ right nodes})$ iterations.

Problem 11: 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.

Solution

Create a source node S, and connect it to each station with an edge. Set the capacity of that edge to the capacity of the station.
Create a sink node T and do the same for cities.

This is now a maximum-flow problem with one source and one sink. Apply FF.

To solve the bonus problem, we'll modify the network before running the algorithm above.

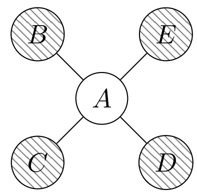
Say an edge from A to B has minimum capacity l and maximum capacity $u \geq l$. Apply the following transformations:

- Add l to the capacity of A
- Subtract l from the capacity of B
- Subtract l from the total capacity of the edge.

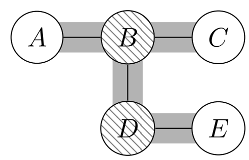
Do this for every edge that has a lower bound then apply the algorithm above.

Part 7: Reductions

Definition 3: Independent Sets
An *independent set* is a set of vertices[2] in which no two are connected. $\{B, C, D, E\}$ form an independent set in the following graph:



Definition 4: Vertex Covers
A *vertex cover* is a set of vertices that includes at least one endpoint of each edge. B and D form a vertex cover of the following graph:



[2] "Node" and "Vertex" are synonyms in graph theory.

Problem 12:
Let G be a graph with a set of vertices V.
Show that $S \subset V$ is an independent set iff $(V - S)$ is a vertex cover.
Hint: $(V - S)$ is the set of elements in V that are not in S.

Solution

Suppose S is an independent set.
$\implies$ All edges are in $(V - S)$ or connect $(V - S)$ and S.
$\implies$ $(V - S)$ is a vertex cover.

Suppose S is a vertex cover.
$\implies$ There are no edges with both endpoints in $(V - S)$.
$\implies$ $(V - S)$ is an independent set.

Problem 13:
Consider the following two problems:
- Given a graph G, determine if it has an independent set of size $\geq k$.
- Given a graph G, determine if it has a vertex cover of size $\leq k$.

Show that these are equivalent. In other words, show that an algorithm that solves one can be used to solve the other.

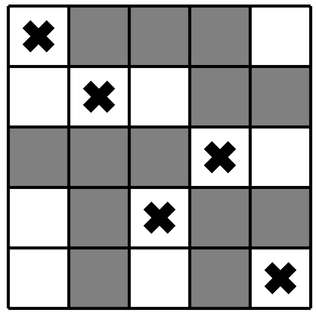
Solution

This is a direct consequence of Problem 12. You'll need to show that the size constraints are satisfied, but that's fairly easy to do.

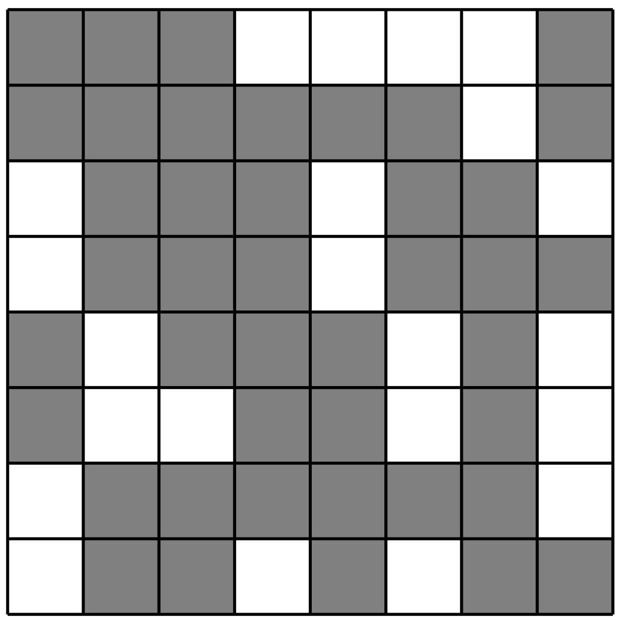
Part 8: Crosses

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 14:
Find a solution for the following grid.



Problem 15:
Turn this into a network flow problem that can be solved with the Ford-Fulkerson algorithm.