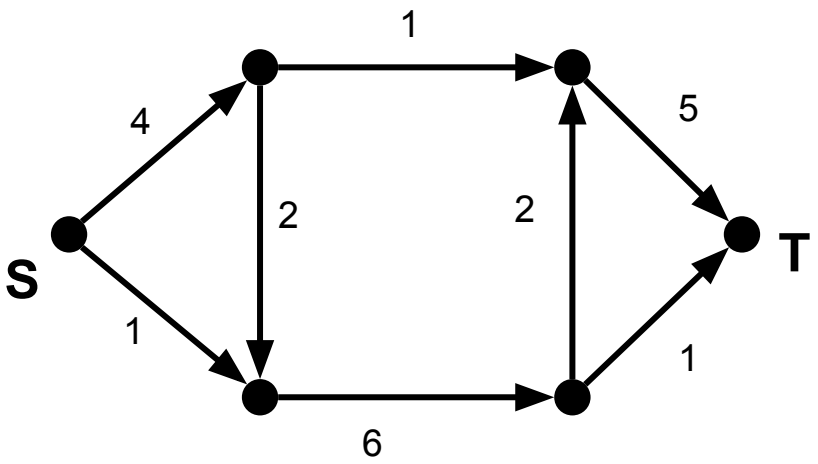
Max Flow & Min Cut

- Intro to max-flow & min-cut
  - Ford-Fulkerson
  - Edmonds-Karp
- Bipartite matching
  - Relation to vertex cover/independent set

What is maximum flow?

Given a weighted directed graph with a source vertex **S** and a sink vertex **T**, where edges represent pipes of different capacities, how much “flow” can you have from **S** to **T**?



Formally, find the maximum value *f* where:

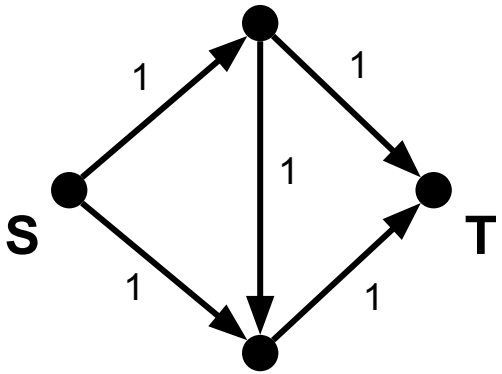
- Total flow out of **S** = *f*
- Total flow into **T** = *f*
- For all other vertices: total inflow = total outflow

Assume for now capacities and flows are integers.

A greedy approach

- Find any path from **S** to **T**, and push one unit of flow through it
- Repeat until no more paths.

Does that work? No, but almost…



“Undoing” mistakes - Augmenting paths

You can effectively “undo” a unit of flow across an edge by going across it the wrong way.

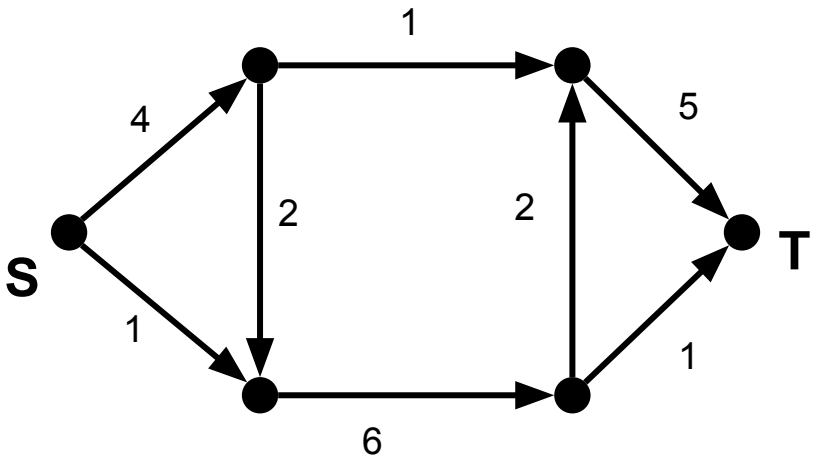
Ford-Fulkerson

1. Find any path *p* from **S** to **T**, using edges with capacity > 0 (e.g. with a DFS)
2. Push one unit of flow along that path:
    a. Reduce the capacity of each edge on *p* by 1
    b. Increase the capacity of each reverse (back) edge on *p* by 1
3. Repeat, until you can't find anymore paths

The algorithm definitely finds a valid flow, but is it the maximum flow?
Assuming it is, it has **O(Ef)** runtime, where f is the maximum flow.

What is minimum cut?

Given a weighted directed graph with a source vertex **S** and a sink vertex **T**, where edges represent roads and weights represent costs to destroy, what is the minimum total cost to separate **S** from **T**?



Formally, find a subset of edges where:

- Every path from **S** to **T** uses at least one of those edges,
- The sum of weights is minimized.

Max flow = min cut?!

TODO: proof goes here

Finding the min-cut

Incidentally, this proves Ford-Fulkerson's correctness
Also, that max flow is always an integer (remember that assumption?)

How do you actually find the minimum cut?

TODO: Put that here

Edmonds-Karp

Can we do better than Ford-Fulkerson's **O(Ef)**?

Two simplifications:

- Instead of increasing flow by 1 each iteration, increase by the minimum capacity along the path.
- Instead of finding any augmenting path, find the *shortest* path (i.e. BFS).

This can be shown to run in $\mathbf{O(VE^2)}$ (but still bounded by **O(Ef)**).

Proof sketch:

- The length of the shortest augmenting path is non-decreasing throughout the algorithm (since the augmentation changes to the residual graph never decreases the shortest path)
- After **E** iterations, the length must increase (since every iteration removes at least one edge from the shortest path DAG)
- Therefore, at most **O(VE)** iterations

Variations on flow/cut

- What if vertices have capacities, not edges?
- What about on an undirected graph?
- What if there are multiple sources your flow can start from?
- What if you had to find, of all the minimum cuts, which one uses the fewest edges?

Transmutation engine

There are **N** types of ingredients. You have a_1 units of type 1, a_2 units of type 2…

To make the philosopher's stone, you need b_1 units of type 1, b_2 units of type 2…

You have **E** machines. The i-th machine can convert type x_i to type y_i at a 1:1 ratio (e.g. 5 units of type x_i into 5 units of type y_i). You can convert a unit multiple times (e.g. Turn a type 3 ingredient into a type 4, then turn that type 4 into a type 2). Using these machines is free.

You also have a super-machine that can turn anything into anything else, but it costs 1 dollar per unit converted.

What is the minimum total cost?

N, **E** <= 300

0 <= a_i, b_i <= 100000

Type	Have	Need
1	6	2
2	1	3
3	0	2
4	1	2

Machine 1: Convert 1 -> 3

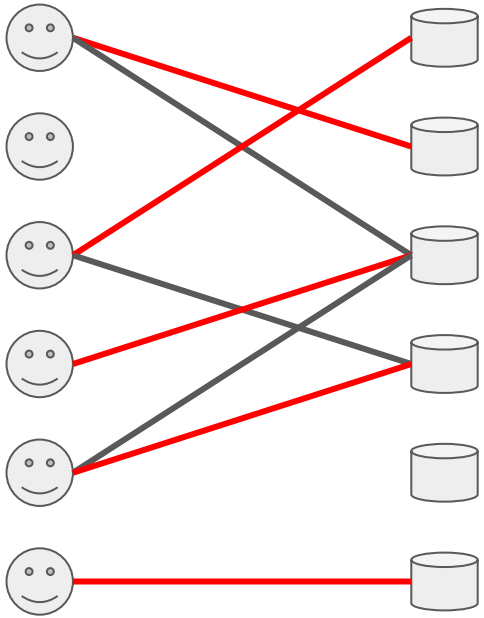
Machine 2: Convert 3 -> 2

Transmutation engine

Maximum Bipartite matching

There are **N** people and **M** desserts. There are **E** edges of the form “person i is willing to eat meal j.” How many people can you give a dessert? Each dessert can go to at most one person.

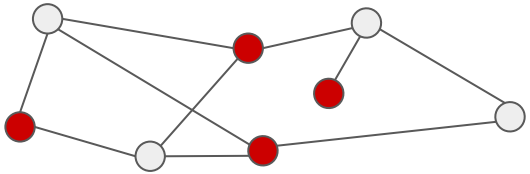
How can you turn this into a flow/cut?



Vertex Cover & Independent Set

Min Vertex Cover: Given a graph, pick the smallest subset of vertices so that for **every** edge, at least one of its endpoints.

Max Independent Set: Given a graph, pick the largest possible subset of vertices so that **no two** vertices have an edge between them.



Minimum vertex cover & Maximum independent set on general graphs is NP-complete.

But for bipartite graphs…

Relation to bipartite matching

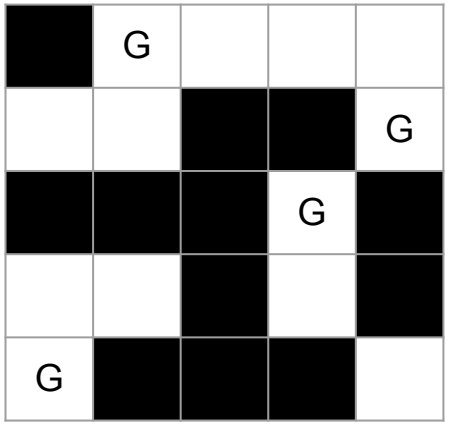
A corollary of the solution we just showed is that for any bipartite graph, the maximum matching equals the minimum vertex cover, known as **Kőnig's theorem** (proof left as an exercise to the reader).

Thus we have the relationships:

- Max matching = Min vertex cover
- Min vertex cover inverted gives max independent set

Royal Guards (no walls)

You have an NxM dungeon. Each square of the dungeon is either empty, or has a hole. What's the maximum number of guards you can place on empty squares, so that no two guards "see" each other? Two guards see each other if they are in the same row, or in the same column.



Other notes

Max flow/min cut is *not* in the IOI syllabus, but Bipartite Matching in **O(VE)** is (i.e. that's just Ford-Fulkerson). In practice, Ford-fulkerson runs pretty fast, especially if you randomize your DFS order. Try that if you have the "right" complexity but get TLE.

There are lots of flow algorithms with a variety of complexities. Usually Edmonds-Karp is good enough for competitive programming. Dinic's is also a popular choice in ICPC notebooks. Hopcroft-Karp is also interesting because it solves bipartite matching in O(fast) (VsqrtE I think). These are not too hard to learn if you want to study them yourself (again, totally unnecessary for IOI).

Vertex cover/independent set is NP-complete for general graphs, but matching on general graphs is in P, see blossoming. Warning: may god have mercy on your soul.

Lots of generalisations of flow type problems, e.g. MCMF, circulation problems.

You can also formulate max flow as an LP.

Method	Complexity	
Linear programming		Co
Ford–Fulkerson algorithm	$O(E\|f_{max}\|)$	As Th gu
Edmonds–Karp algorithm	$O(VE^2)$	A
Dinic's algorithm	$O(V^2E)$	In O
MKM (Malhotra, Kumar, Maheshwari) algorithm[10]	$O(V^3)$	A
Dinic's algorithm with dynamic trees	$O(VE\log V)$	Th
General push–relabel algorithm[11]	$O(V^2E)$	Th co
Push–relabel algorithm with *FIFO* vertex selection rule[11]	$O(V^3)$	Pu
Push–relabel algorithm with *maximum distance* vertex selection rule[12]	$O(V^2\sqrt{E})$	Pu
Push-relabel algorithm with dynamic trees[11]	$O\left(VE\log\frac{V^2}{E}\right)$	Th
KRT (King, Rao, Tarjan)'s algorithm[13]	$O\left(VE\log_{\frac{E}{V\log V}}V\right)$	
Binary blocking flow algorithm[14]	$O\left(E\cdot\min\{V^{2/3},E^{1/2}\}\cdot\log\frac{V^2}{E}\cdot\log U\right)$	Th
James B Orlin's + KRT (King, Rao, Tarjan)'s algorithm[9]	$O(VE)$	Or
Kathuria-Liu-Sidford algorithm [15]	$E^{4/3+o(1)}U^{1/3}$	Int
BLNPSSSW / BLLSSSW algorithm [17] [18]	$\tilde{O}((E+V^{3/2})\log U)$	Int
Gao-Liu-Peng algorithm [19]	$\tilde{O}(E^{\frac{3}{2}-\frac{1}{328}}\log U)$	Ga un