Kattis

Graphs Part 3

Atli Fannar Franklín

October 12, 2025

School of Computer Science
Reykjavík University

Today we're going to cover

- Maximum flow
  - Ford-Fulkerson
  - Edmond-Karp
  - Dinic's
  - MCMF

Flow problems

- What is maximum flow?
- We imagine each edge in a (directed) graph is a pipe, and the weight says how many units of liquid can pass through it per second.
- Then vertices are joints, and the amount of liquid going in and out must be the same.
- Except for one vertex where we let water flow in from the outside, the source, and one where we let it flow out, the sink.
- Then the maximum flow problem is the problem of determining the maximum amount of liquid that can flow per second from the source to the sink.

Formal definition

- For a graph $G = (V, E)$ we say that $f : E \to \mathbb{R}$ is a flow from $s \in V$ to $t \in V$ if $0 \leq f(e) \leq c_e$ where c_e is the capacity for e and for all $v \in V \setminus s, t$ we have

$$\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{in}(v)} f(e)$$

- The value of a flow f is the amount of liquid flowing out at the source (or in at the sink, equivalently), more formally given as

$$\sum_{e \in \text{out}(s)} f(e) = \sum_{e \in \text{in}(t)} f(e)$$

- The maximum flow problem is then to find the flow with the maximum value for a given graph.

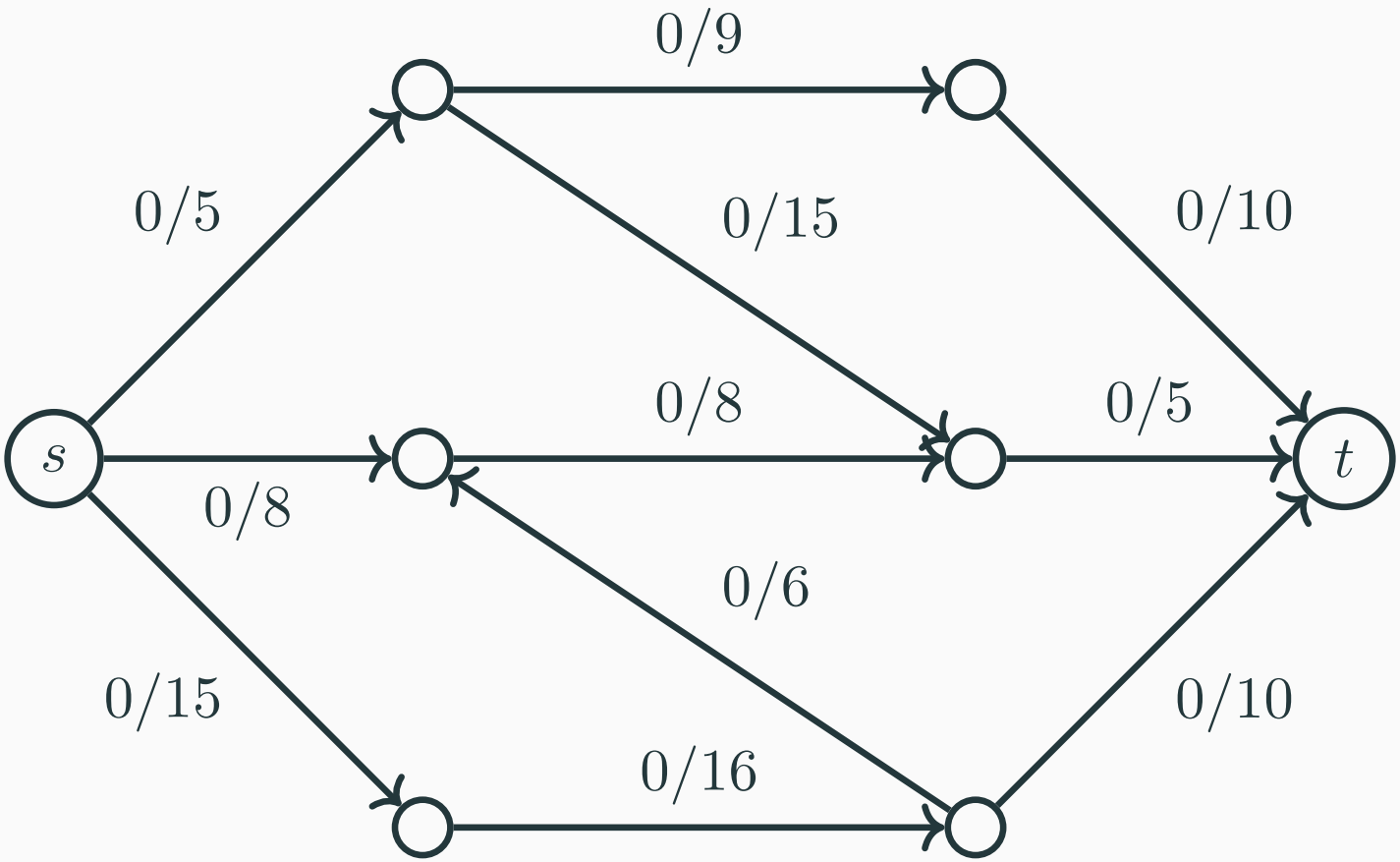
First solution ideas

- How do we solve this?
- Brute force trying every combination will be massively slow.
- So we just try something greedy.

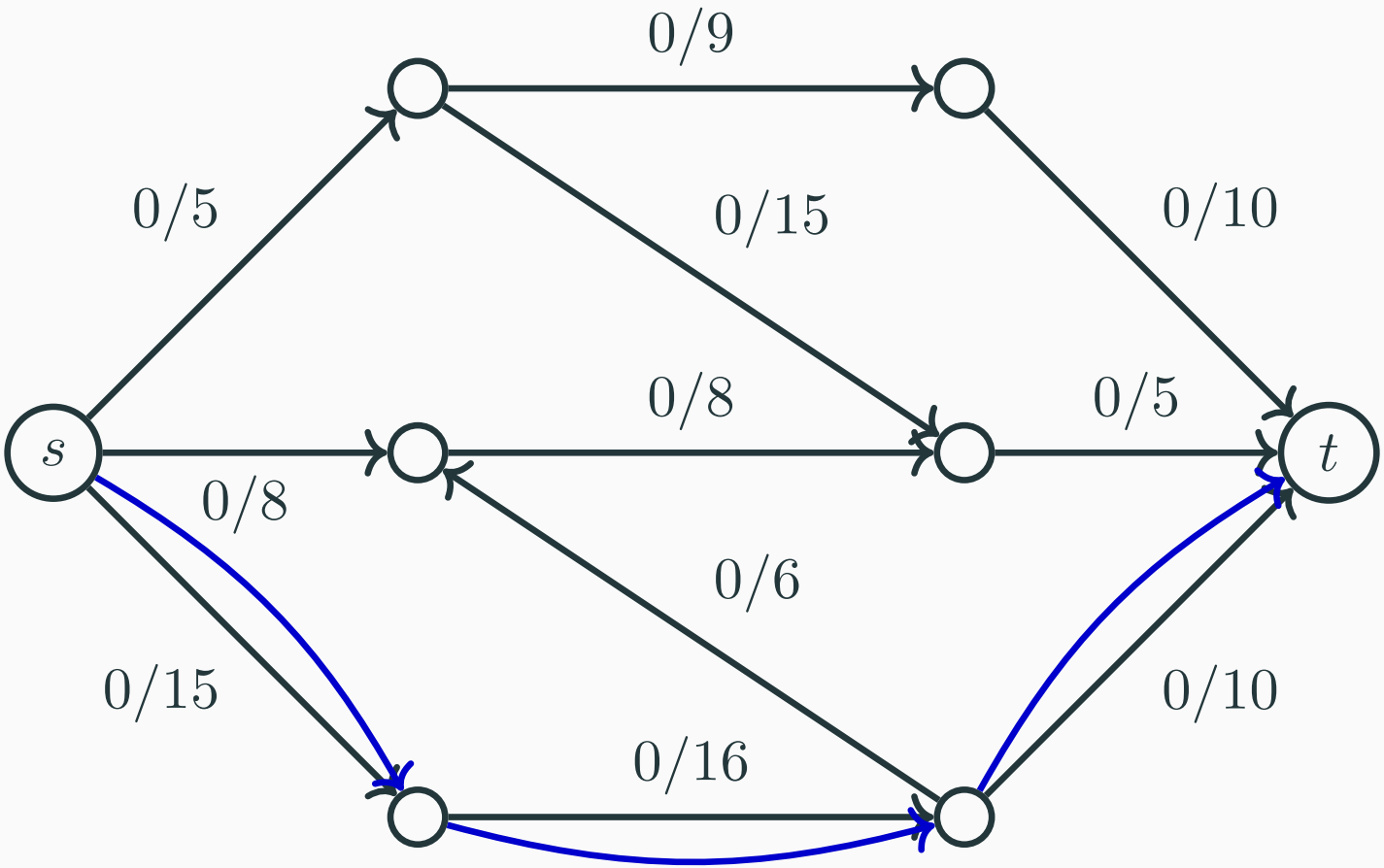
First solution ideas

- How do we solve this?
- Brute force trying every combination will be massively slow.
- So we just try something greedy.
- How about we just find some path from s to t where we can fit more flow, and fit more flow there?
- Rinse and repeat, until we get stuck.

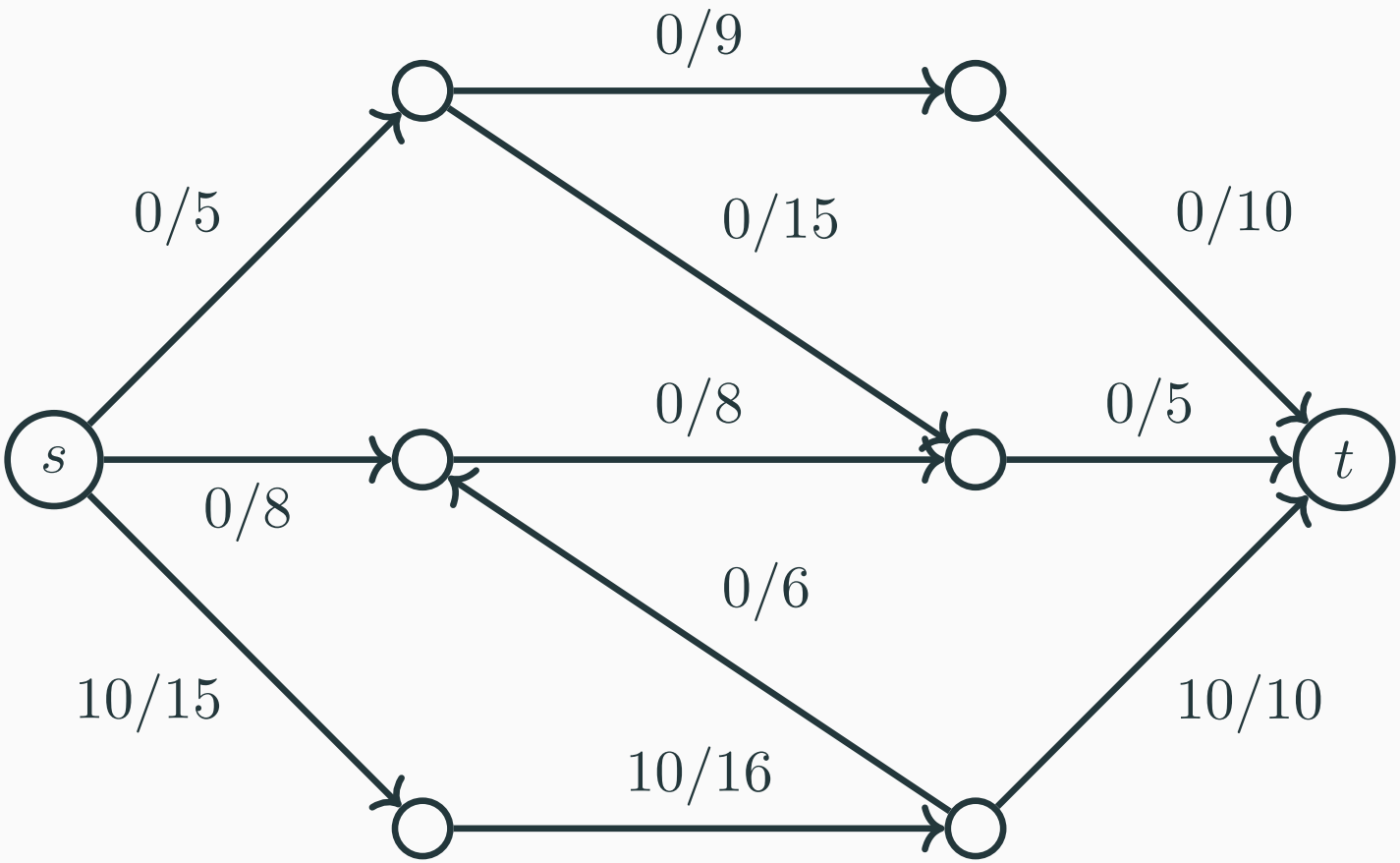
Example



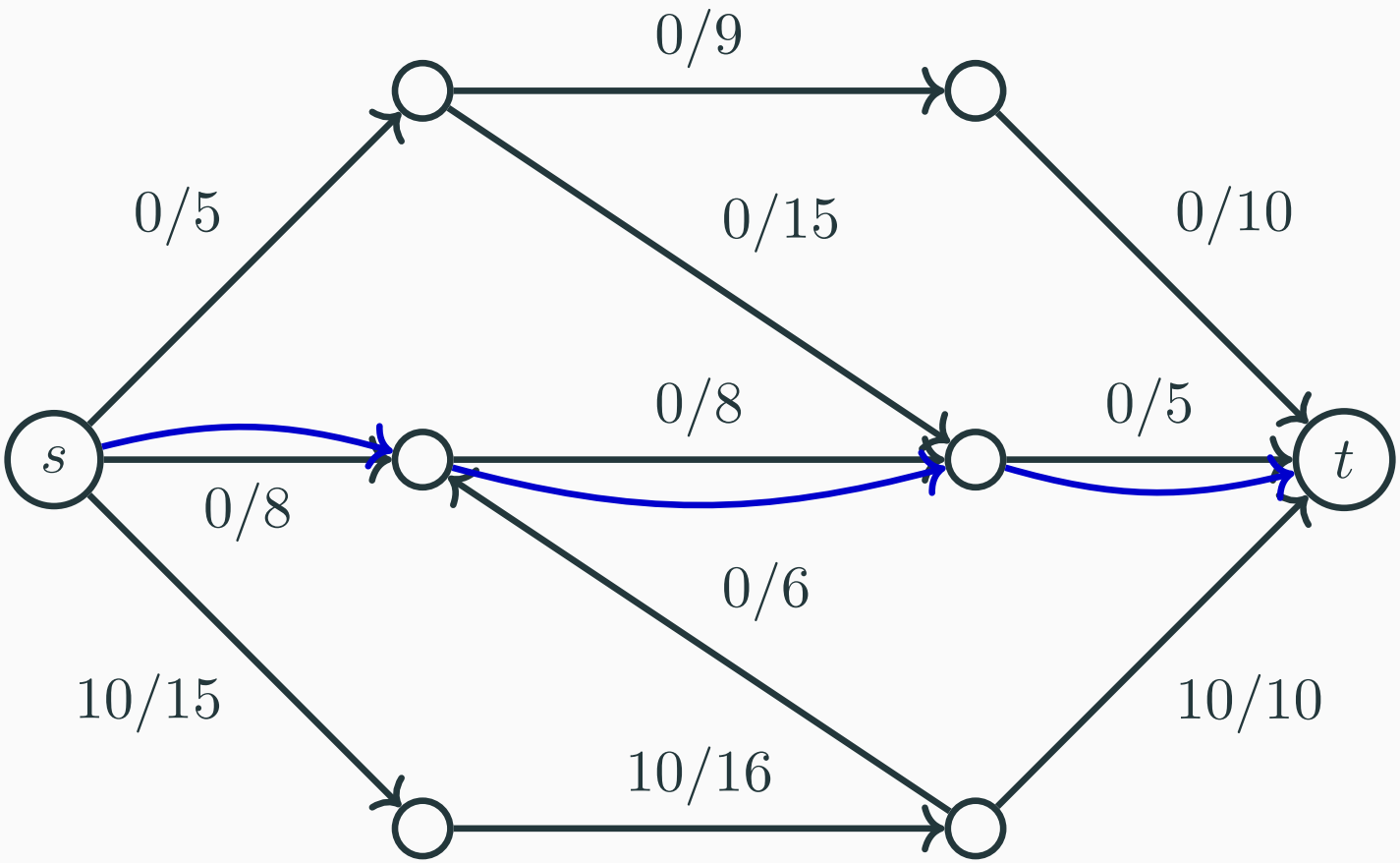
Example



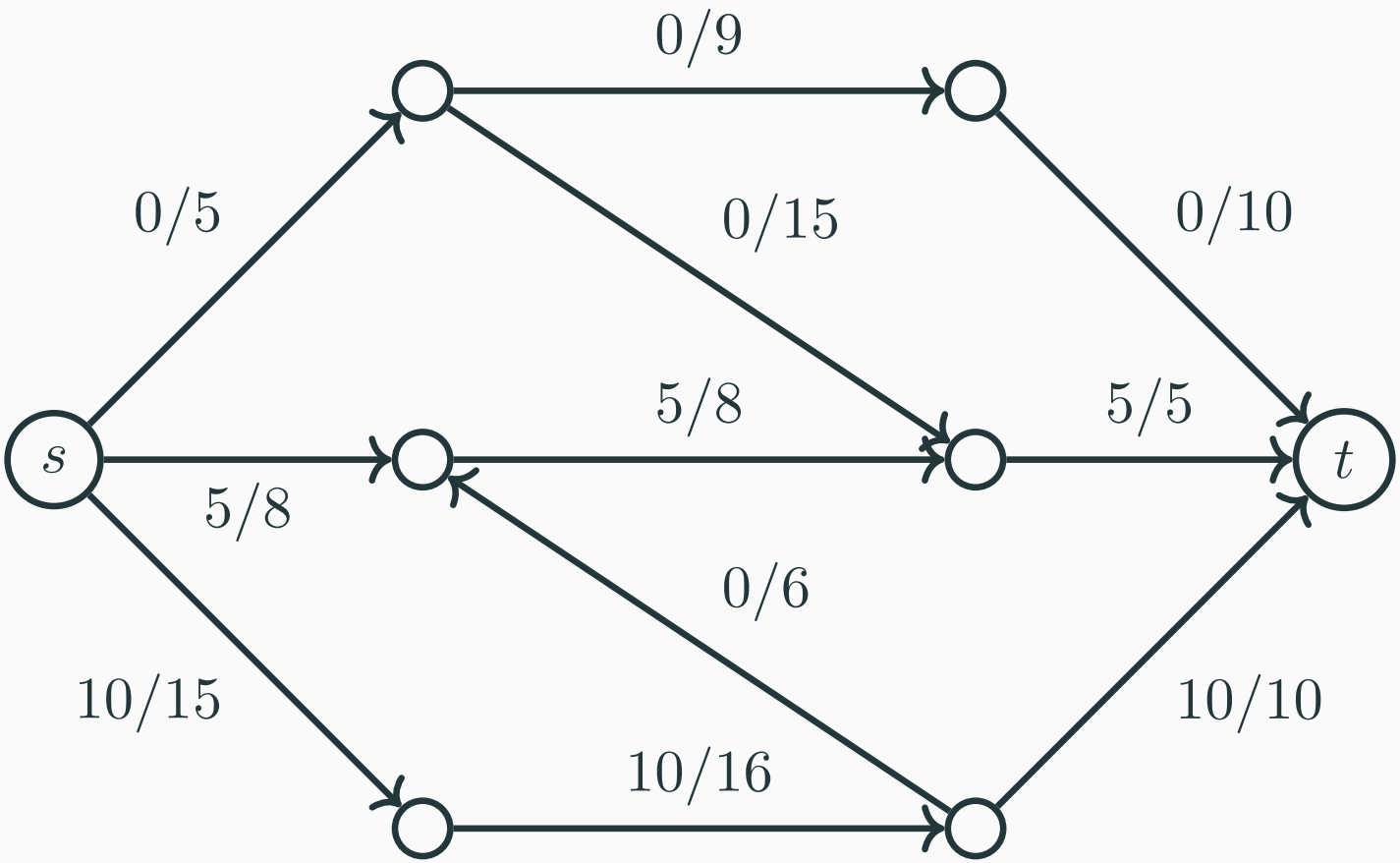
Example



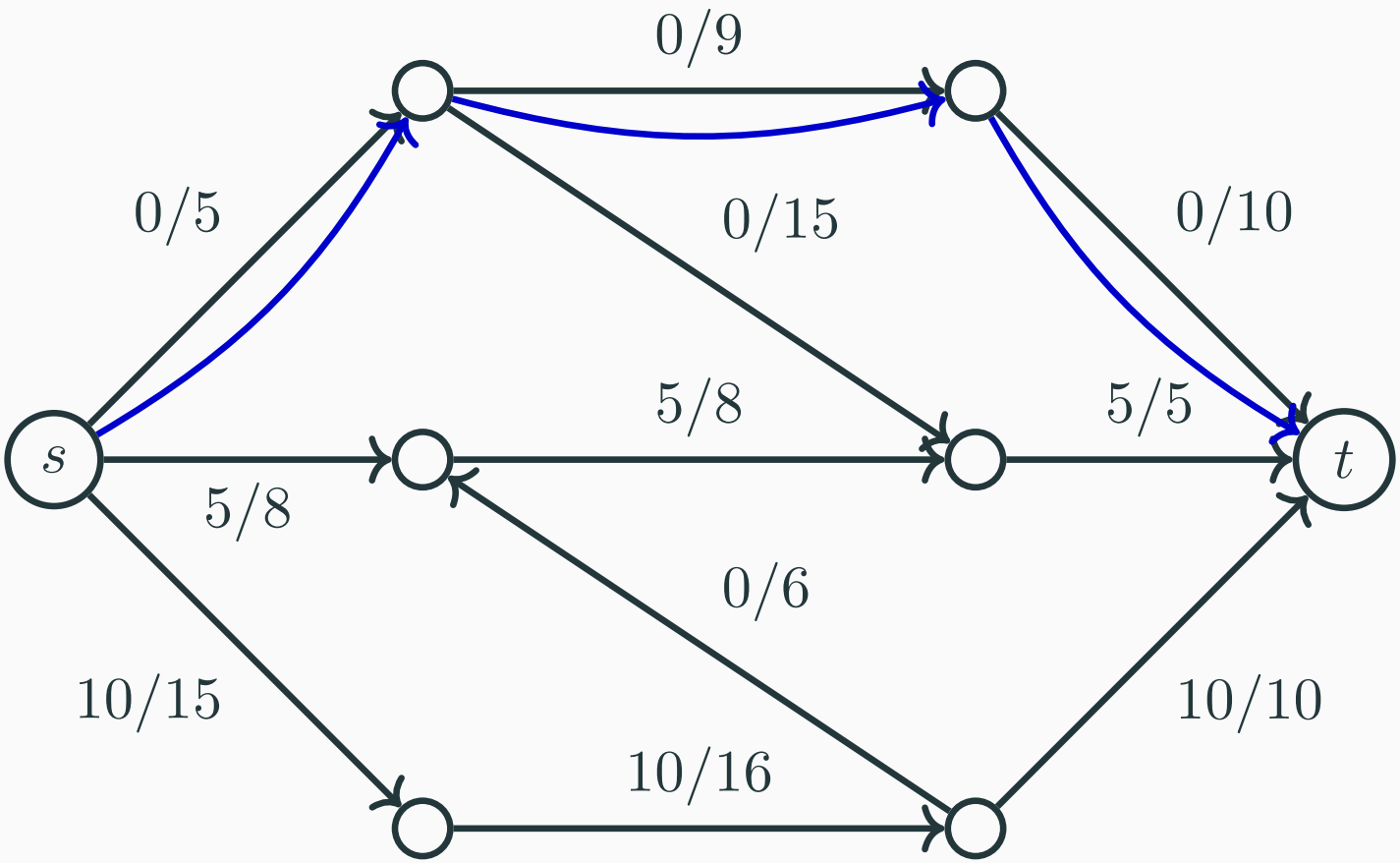
Example



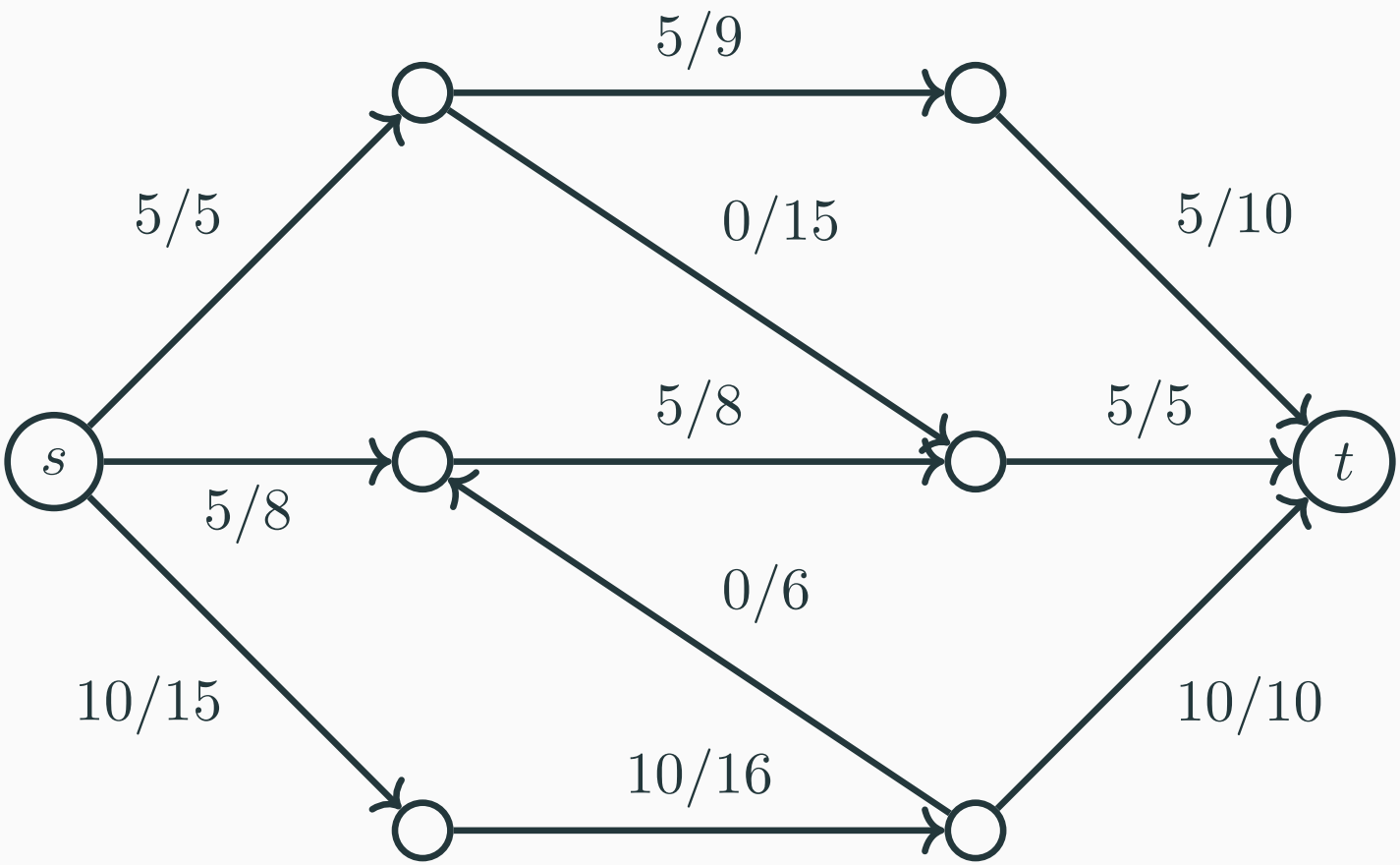
Example



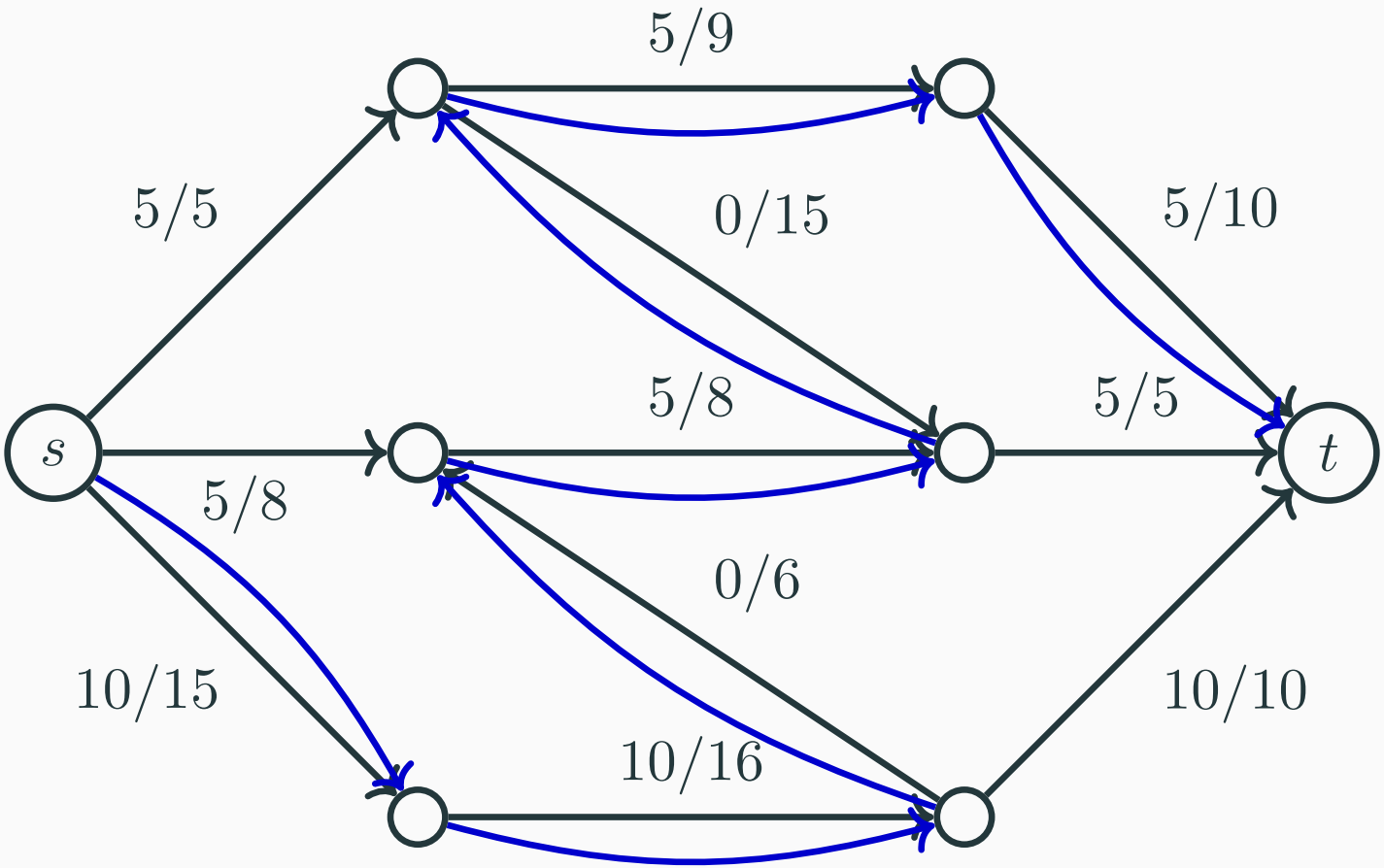
Example



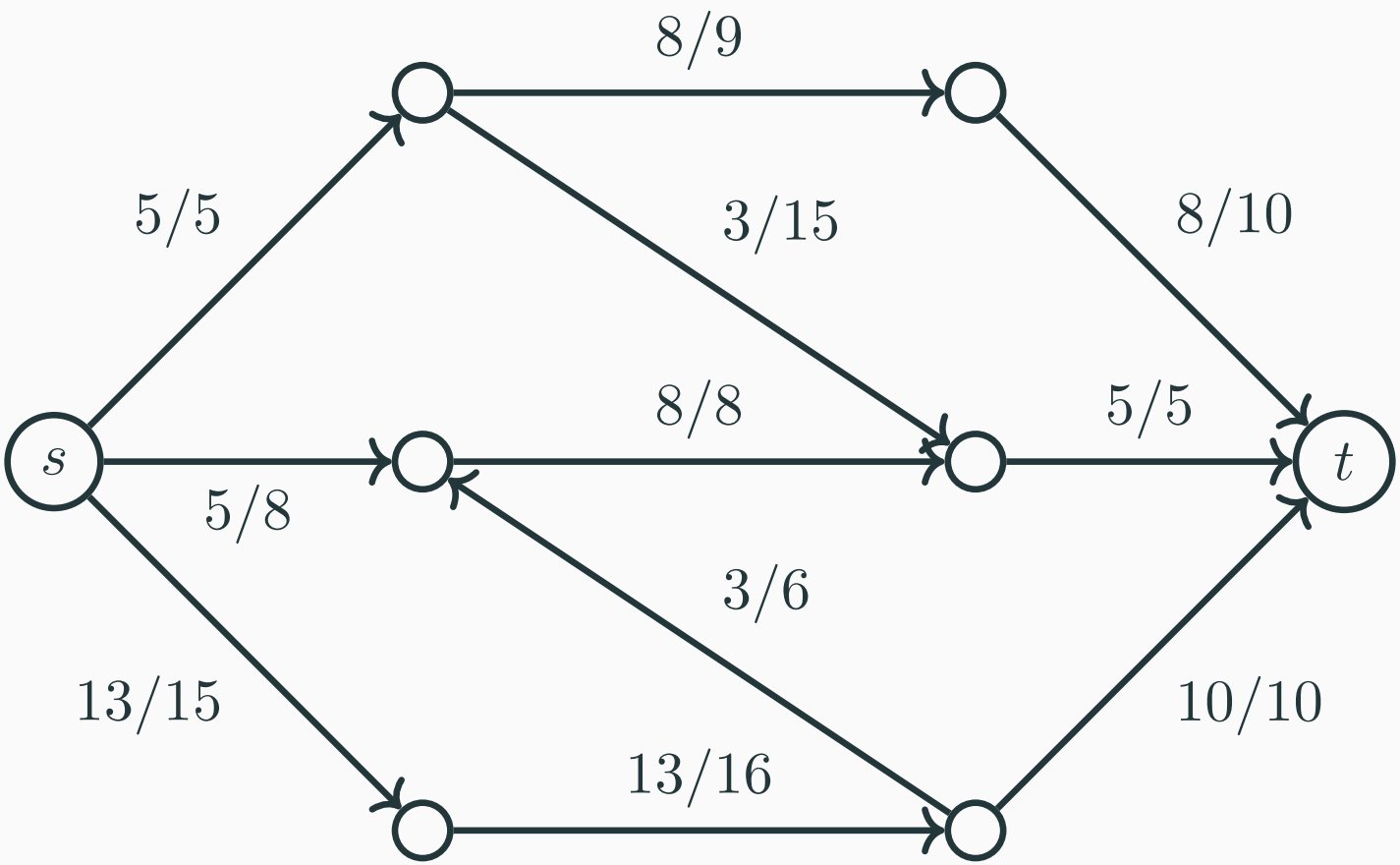
Example



Example



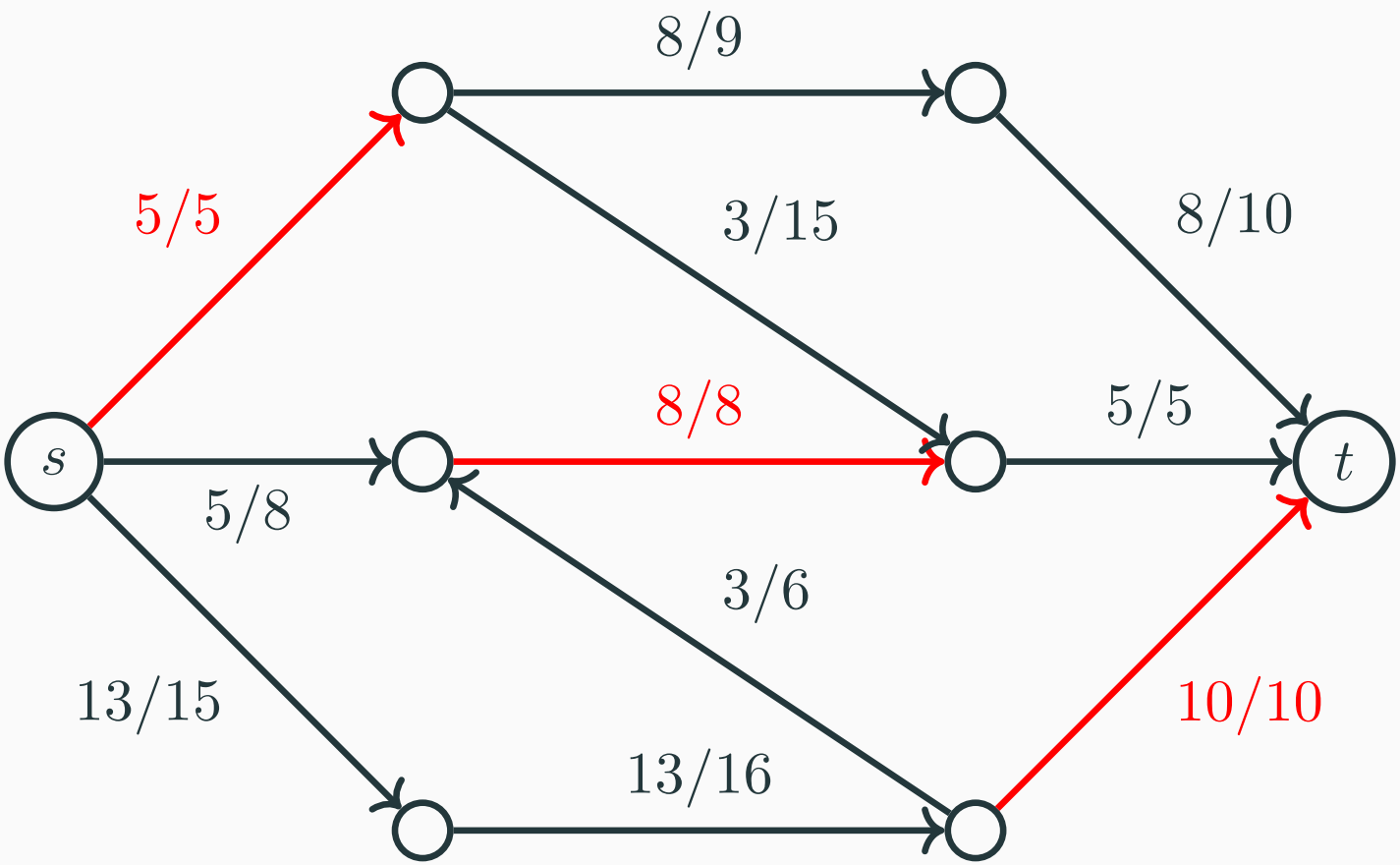
Example



Min-cut

- We see that the value of the flow above is 23, but how do we know that's the maximum?
- The crux of showing that you can't do better is to look at the min-cut.
- This is a dual problem, asking what's the cheapest (minimum total capacity) set of vertices you need to cut to disconnect s from t?

Example



Max-flow Min-cut

- The fact that the min cut has value 23 as well is no coincidence.
- It is not too hard to see that a max flow can never exceed a cut, as the flow has to pass from one side to the other of the cut using only the edges in the cut.
- So all cuts have costs greater or equal to the values of all flows.
- So if we find a flow that has value equal to the cost of a cut, that flow must be maximal and that cut minimal.
- The fact is, this is always the case. The maximum flow value is always equal to the cost of the minimum cut, proved by Ford and Fulkerson in 1962.

Max-flow Min-cut

- The proof is not so hard, it is just a proof by looking at the greedy algorithm above and showing that if the flow is not yet equal to the cost of a minimum cut there must be some path that gets us closer.
- The proof also gives us that our greedy method always gets us closer to the answer in each step.
- Note that since our algorithm always increases the flow as much as it can along the path we choose, this means our flow will always have integer flow values if our capacities have integer values.
- If we do not have integer values, the greedy algorithm might not terminate!

Ford Fulkerson

- This is known as the Ford-Fulkerson algorithm. So how fast is it if we have integer capacities?
- We have to find an augmenting path, we can do this using some standard graph search.

Ford Fulkerson

- This is known as the Ford-Fulkerson algorithm. So how fast is it if we have integer capacities?
- We have to find an augmenting path, we can do this using some standard graph search.
- So we have $\mathcal{O}(V + E)$ but the flow might just increase by one each time.
- Therefore the complexity is $\mathcal{O}(F(V + E))$ where F is the maximum flow.

Edmond Karp

- The problem is that if the capacities are very large, this is quite slow (in the worst case).
- There are ways around this, we just have to be more specific about how we find our augmenting paths.
- We essentially have two options, DFS and BFS:
- If you try both, you will see that BFS gives better worst-case behaviour.

Edmond Karp

- Ford-Fulkerson with BFS is known as Edmond-Karp.
- The key feature one can prove is that if you saturate an edge several in the process, it must always be further away from the source along the augmenting path than last time. (Proof by contradiction, bit tricky, but not terrible)
- Then each edge can be saturated at most V times, so we do at most VE augmentations.
- Thus the time complexity is $\mathcal{O}(VE^2)$ at most (still also at most $\mathcal{O}(FE)$)

Implementation, part 1

```
# We use this as the queue module is not what we want
# It is meant for asynchronous tasks, not this
from collections import deque
from math import inf
# We define a class that takes in a graph
# and provides a max flow method
class FlowNetwork:
```

Implementation, part 2

```
def __init__(self, graph):
    # We construct the residual graph
    self.residual = [[] for _ in range(len(graph))]
    self.edges = []
    for vertex in range(len(graph)):
        for cap, neighbour in graph[vertex]:
            # Each edge is doubled and always adjacent
            # This means we can get the reverse edge with
            # ^1, since 0 and 1 map to each other, 2 and 3
            # and so on (just flips last bit in number)
            # Before adding the edge the length of the list
            # gives the index the new edge will be at
            self.residual[vertex].append(len(self.edges))
            # Initially we have a budget of up to 'cap'
            e1 = [vertex, neighbour, cap]
            self.edges.append(e1)
            self.residual[neighbour].append(len(self.edges))
            # But the reverse edge has no initial budget
            e2 = [neighbour, vertex, 0]
            self.edges.append(e2)
```

Implementation, part 3

```
def bfs(self, s, t, parent): # s is the source, t the sink
    # Keep an array of the flow found, augmenting path pushes to vertex
    flow = [0 for i in range(len(self.residual))]
    # Initialize BFS
    queue = deque()
    queue.append(s)
    flow[s] = inf
    while queue:
        cur = queue.popleft()
        for index in self.residual[cur]:
            edge = self.edges[index]
            # edge[1] is endpoint of edge, edge[2] is cap
            # So if we've already looked at endpoint or
            # the edge is saturated we skip
            if flow[edge[1]] > 0 or edge[2] == 0:
                continue
            # Otherwise update and continue
            queue.append(edge[1])
            flow[edge[1]] = min(flow[cur], edge[2])
            parent[edge[1]] = index
    # flow[t] contains how much we can push to sink
    return flow[t]
```

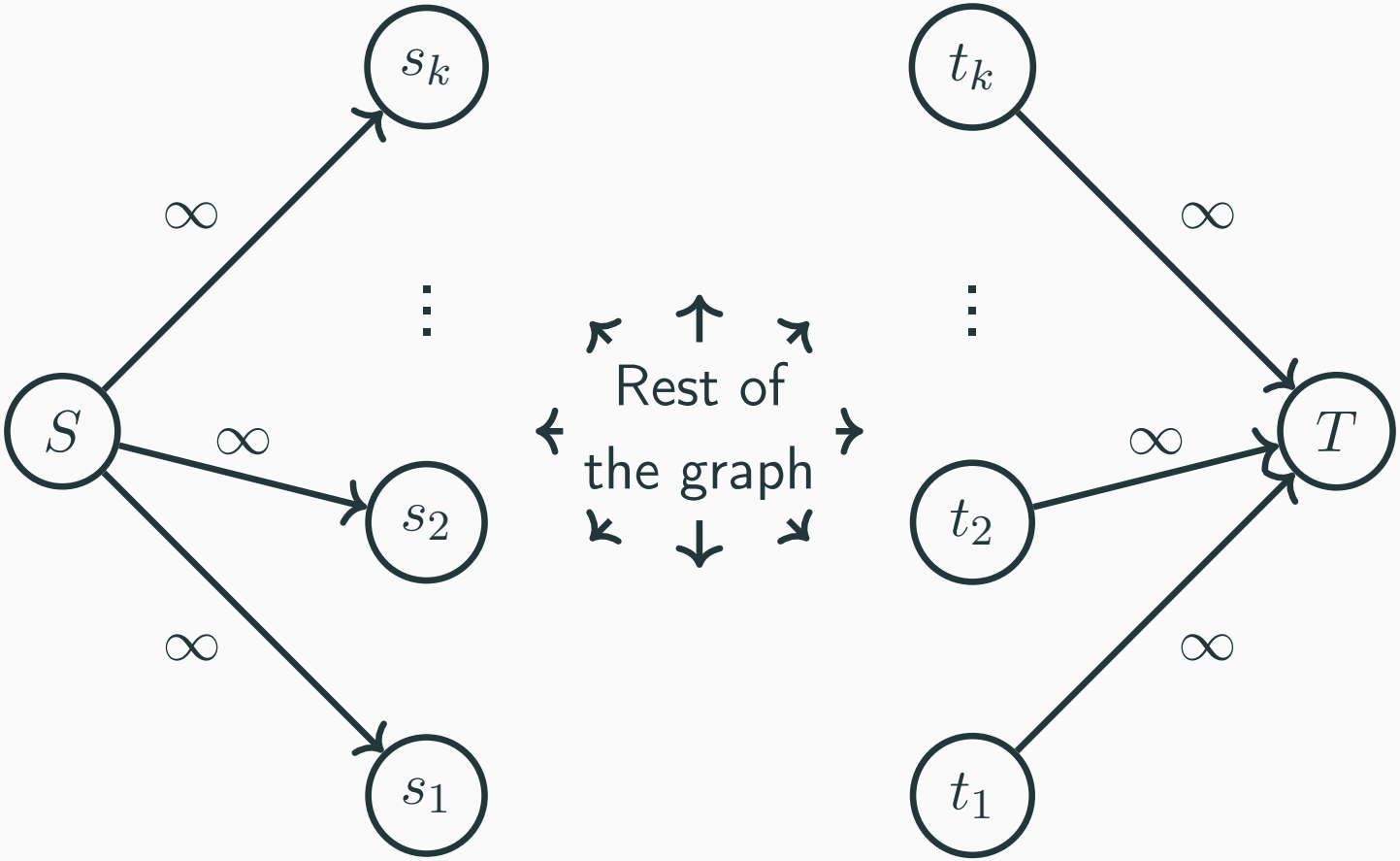
Implementation, part 4

```
def max_flow(self, s, t):
    parent = [-1 for _ in range(len(self.residual))]
    max_flow = 0
    # This will set path_flow to the result of bfs
    # and only loop if that is > 0
    while path_flow := self.bfs(s, t, parent):
        # Add it to max flow
        max_flow += path_flow
        # Now we need to walk along the path
        # and update the residual capacities
        position = t
        while position != s:
            index = parent[position]
            # Decrease residual capacity
            self.edges[index][2] -= path_flow
            # The reverse edge is adjacent, ^1 trick
            self.edges[index ^ 1][2] += path_flow
            position = self.edges[index][0]
     # And we're done!
    return max_flow
```

But then what?

- What do we do with this then?
- In fact very many things can be modeled as maximum flow.
- Probably one of the most practical algorithms in this course. But to show some examples, first we have to look at a general construction that allows us to have multiple sources and multiple sinks.
- Any suggestions how?

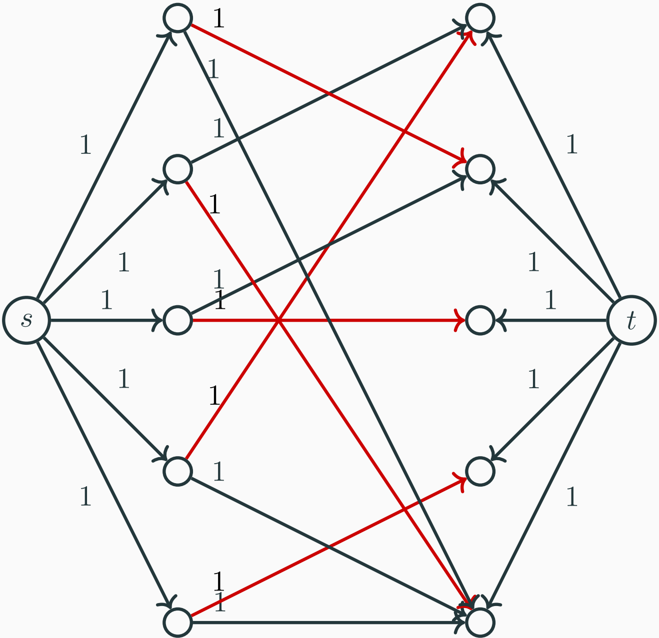
Multi source/sink



Bipartite matching

- There are many practical situations where we want to pair up some set of objects with another, but not every pair is acceptable, and each objects can only be in one pair.
- As an example my team solved a problem for a contest of practical problems hosted by Google (Hash Code) about pairing employees to tasks they can handle using a variant of this method.
- We can use maximum flow just by making the graph on the last slide (except with 1 instead of ∞ so each vertex can only be in ine pair) and putting an edge with capacity 1 between any two vertices we are allowed to pair together.

Bipartite matching



Bipartite matching

- Here actually the "bad" bound of $\mathcal{O}(FE)$ is better than $\mathcal{O}(VE^2)$ as FE simplifies to VE in this graph.
- Though we will see later a way to do it in $\mathcal{O}(E\sqrt{V})$.
- We will also later consider the case where each pairing has different weights, like the cost of hiring an employee to do a particular task, where we want to minimise total cost as well.
- Let's first take two other examples of problems solvable with max flow.

Baseball league

- TODO

Project selection

- TODO

Theorems

- Some theorems also give us use cases for maximum flow. König's theorem for example gives us the solution to the minimum vertex cover (fewest vertices needed to touch every edge) problem in bipartite graphs in terms of the maximum matching.
- Menger's theorem tells us that the number of disjoint paths between s and t in a graph is equal to the maximum flow from s to t if every edge has unit capacity. This way we can find whether a graph is k-edge-connected. (There is also a version of Menger's theorem for vertex connectivity)

Tricks

- Aside from there being many theorems that show that max flow or problems solved solved by max flow are equivalent to something else, there are also many constructions that allow us to solve more general problems.
- We already saw how to have multiple sources and sinks, but let's look at some other examples of this.
- We'll look at vertex capacities and flows with demands as two examples.

Vertex split

- Say we have a flow problem, but we want to limit the amount of flow that goes through each vertex as well
- We can split the vertex, and make it have an internal edge with a capacity
- So each edge pointing to v goes to a new vertex v_{in}, each pointing out goes from a new vertex v_{out}, and then we put an edge from v_{in} to v_{out} with our desired capacity

Vertex split

- Say we have a flow problem, but we want to limit the amount of flow that goes through each vertex as well
- We can split the vertex, and make it have an internal edge with a capacity
- So each edge pointing to v goes to a new vertex v_{in}, each pointing out goes from a new vertex v_{out}, and then we put an edge from v_{in} to v_{out} with our desired capacity

With demands

- What if we want each edge e to have a minimum flow d_e? We can ask for any flow or even the minimum flow
- We can actually augment the graph to solve this with maximum flow as well
- This won't be in the homework, but is a good exercise for those wanting to try it

Capacity scaling

- Another trick one can use to do max flow is to start with only capacities 0/1 and solve that in $\mathcal{O}(FE) = \mathcal{O}(E)$
- Then we double all the capacities and flows that should be ≥ 2 and solve again, each edge needs to be incremented at most once, so this takes $\mathcal{O}(E^2)$
- Then we double, and repeat
- This trick is known as capacity scaling, which solves max flow in $\mathcal{O}(E^2 \log(F))$

Dinic's

- TODO

MCMF

- TODO

Hungarian

- In the case of bipartite matchings where we want maximum/minimum weight, then MCMF will solve it, but there is a better solution
- We won't cover it in detail here, but it's known as the Hungarian algorithm or Kuhn-Munkres algorithm
- Interested students can also look into the push-relabel algorithm which is even faster than Dinic's