Kattis

Graphs Part 3

Atli Fannar Franklín

October 20, 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.

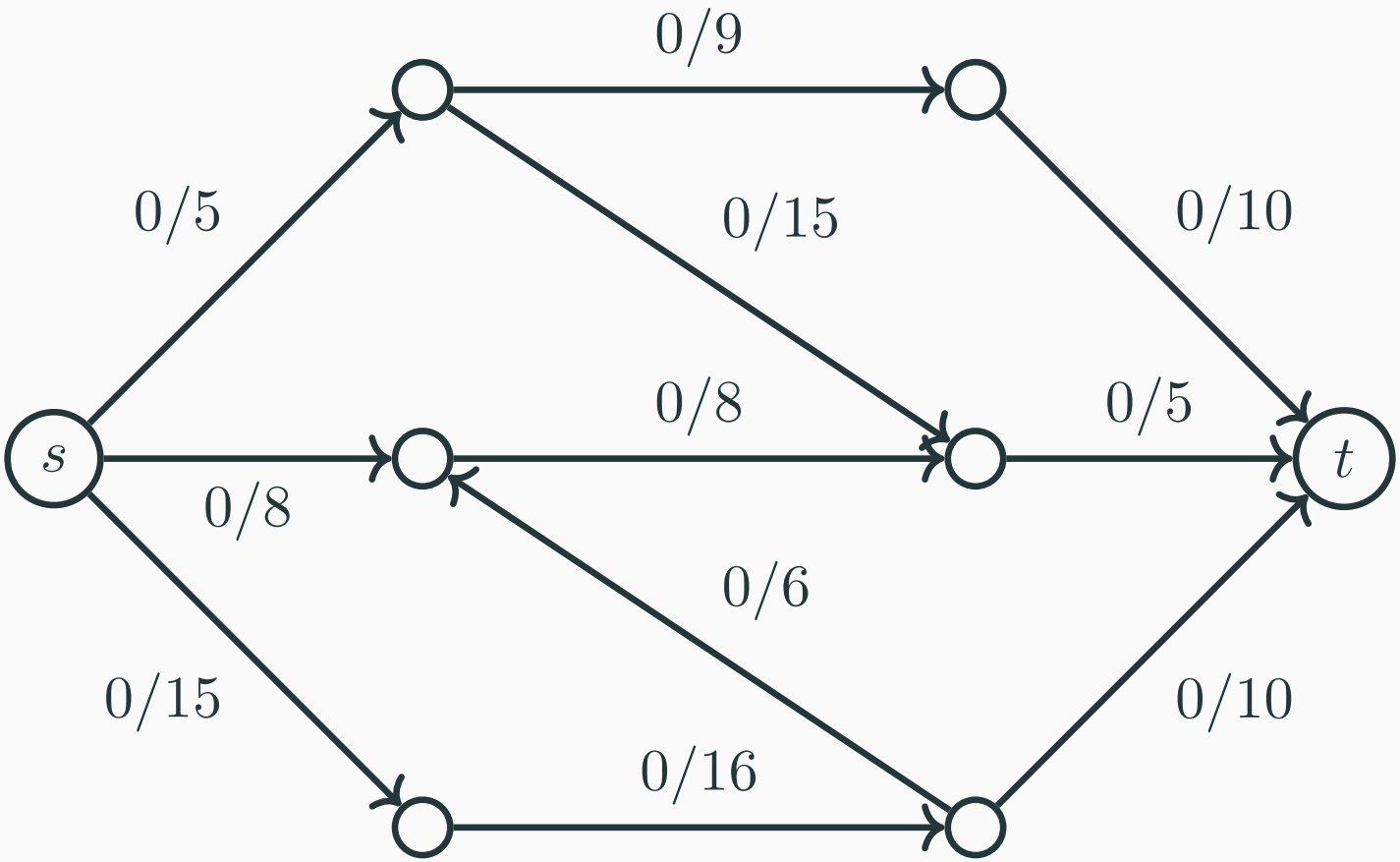
First solution ideas

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

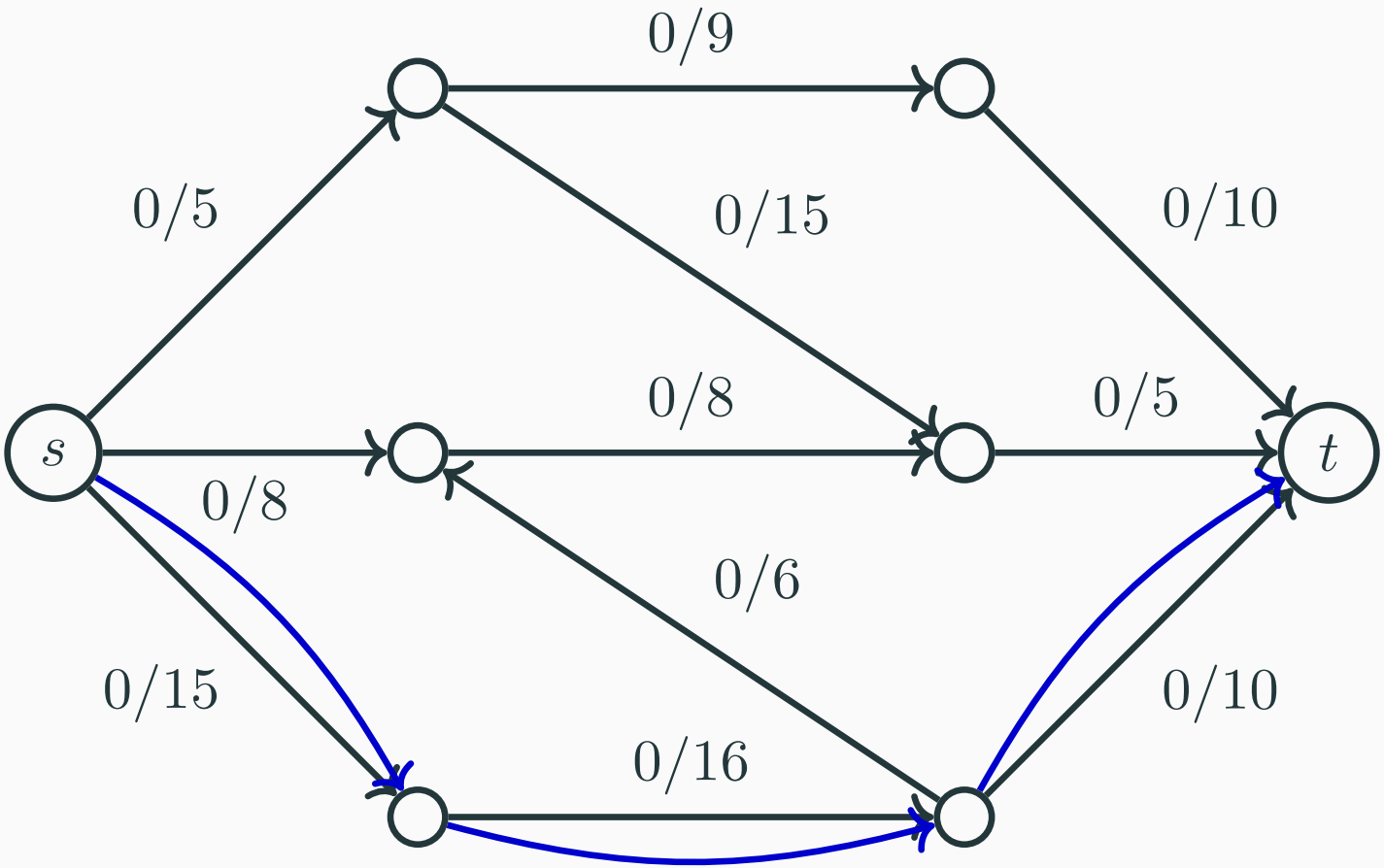
First solution ideas

- 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.

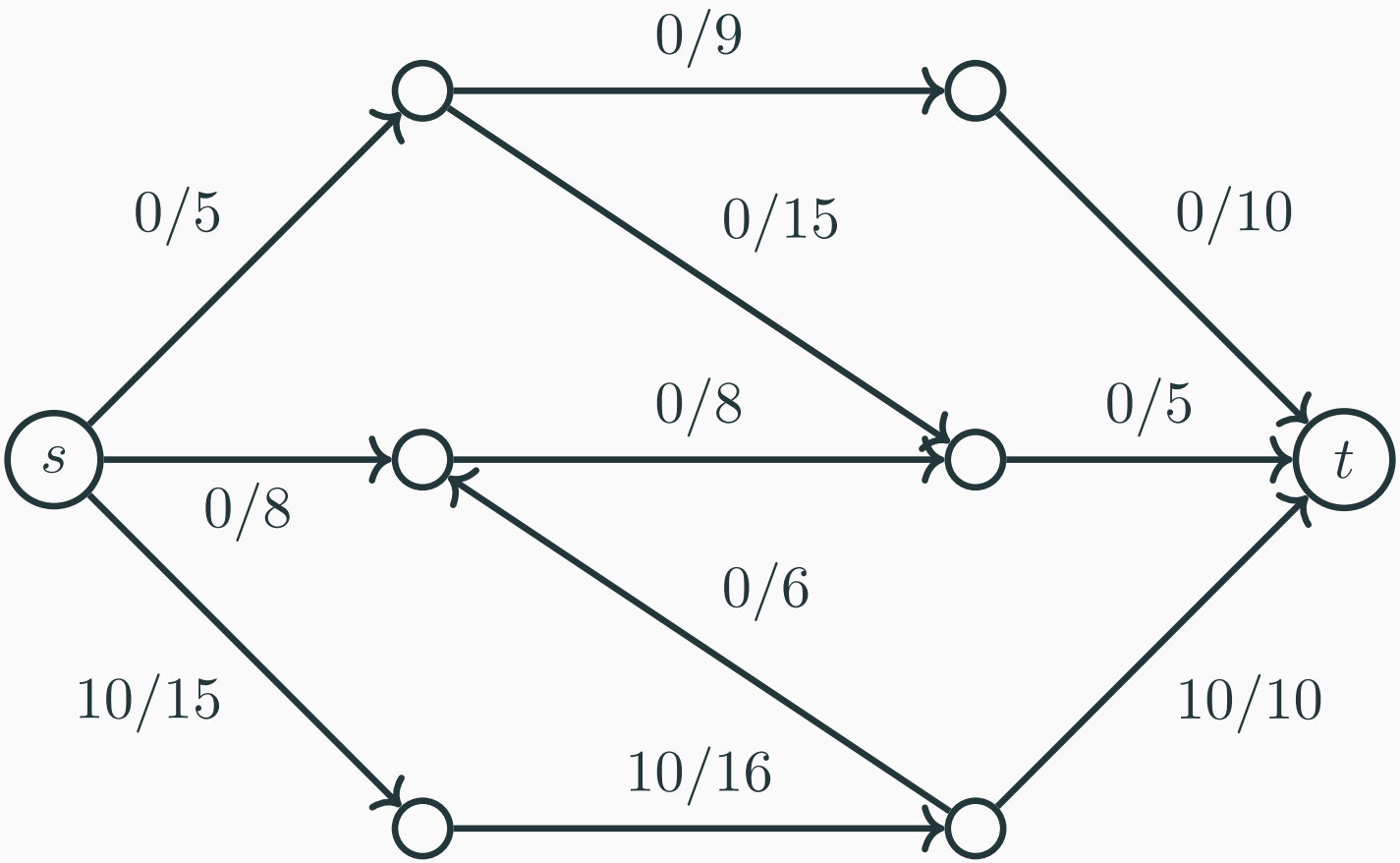
Example



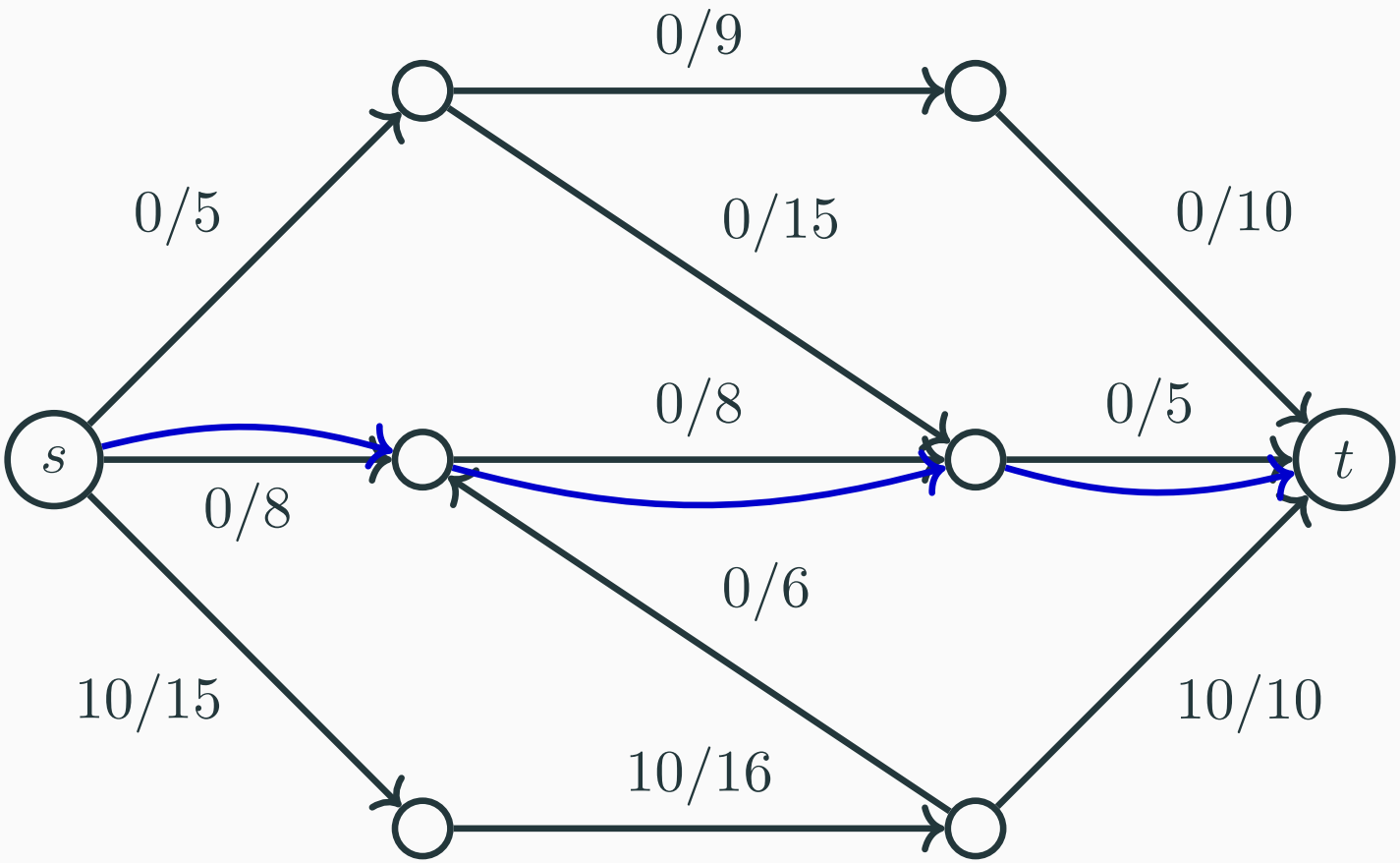
Example



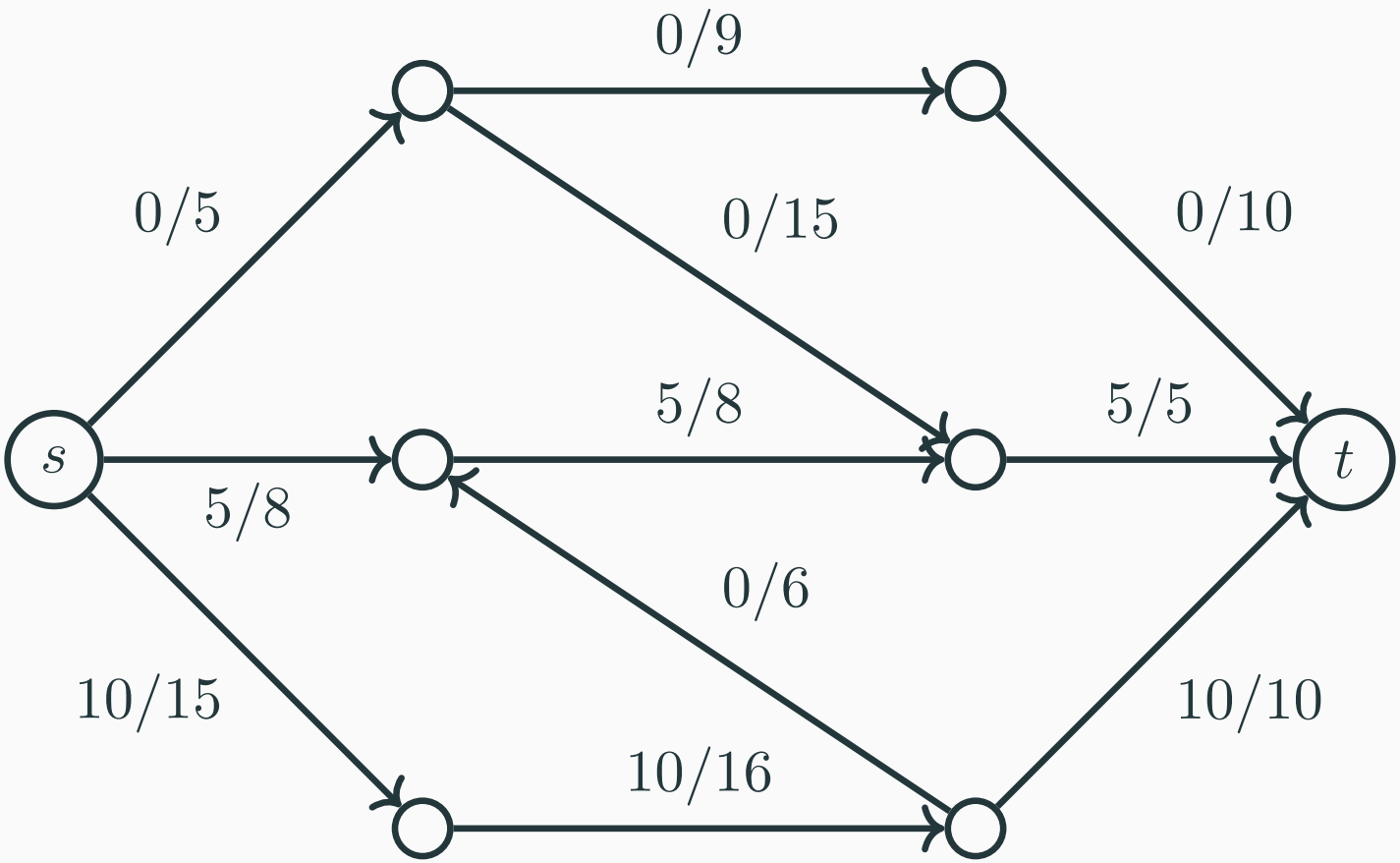
Example



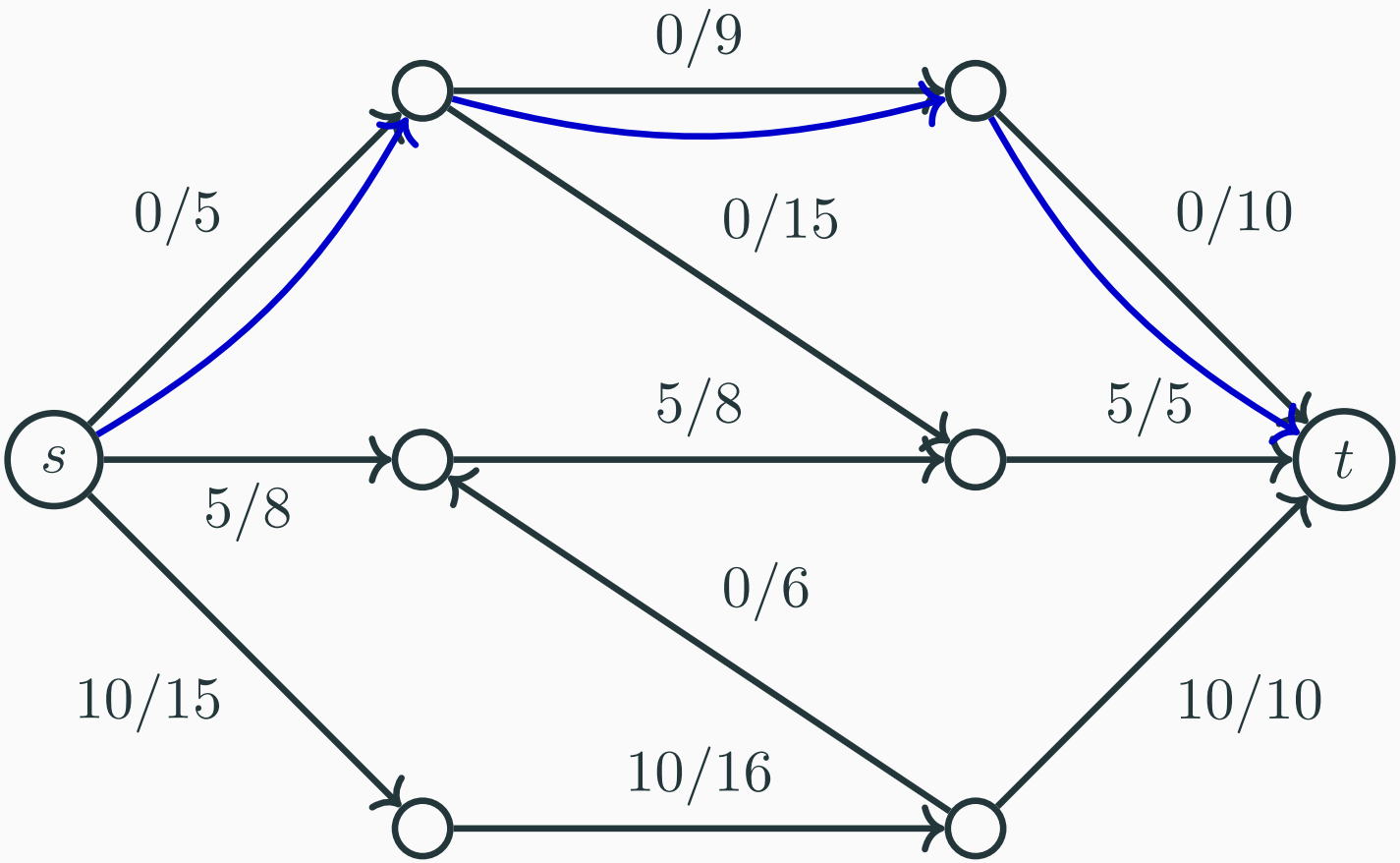
Example



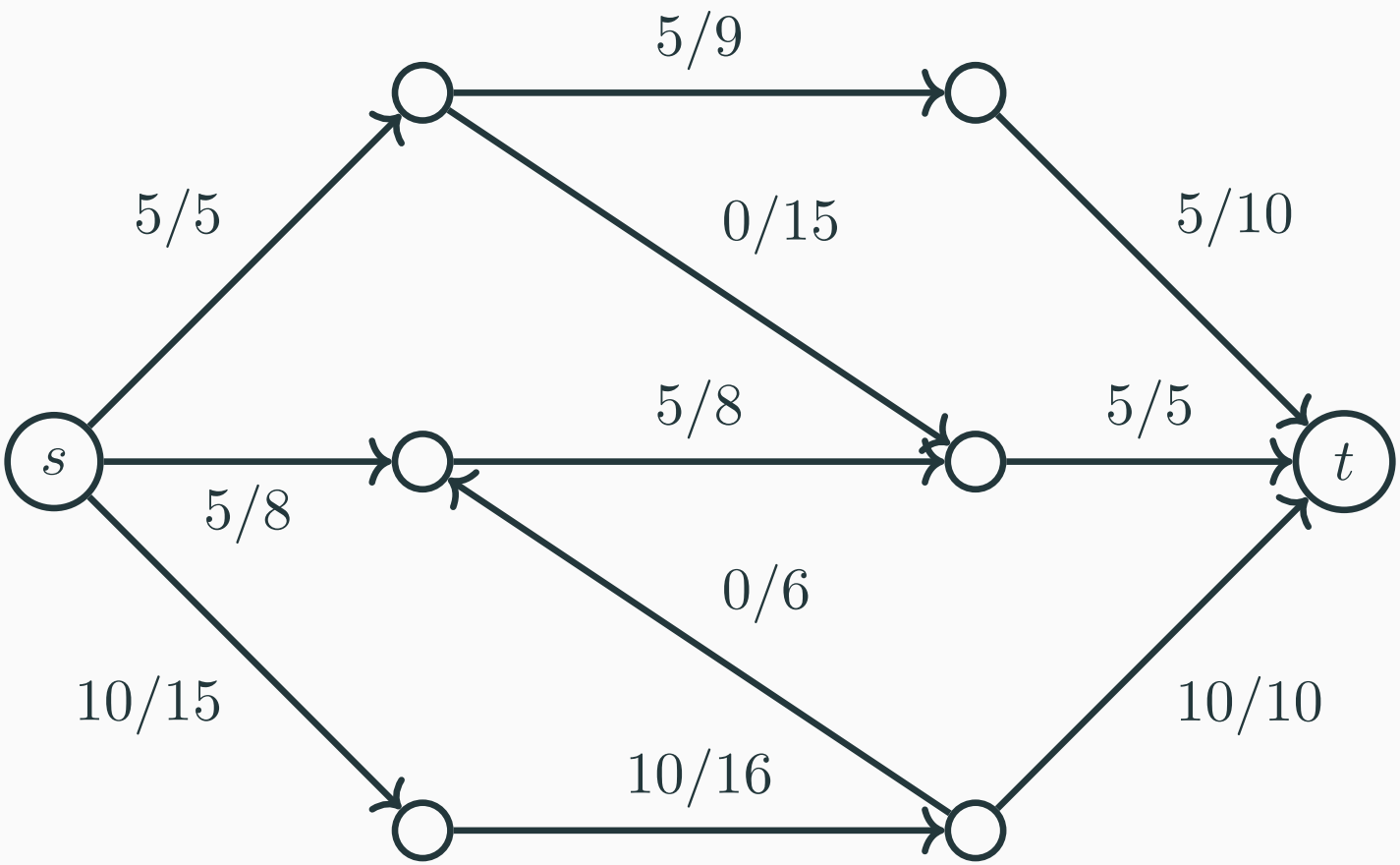
Example



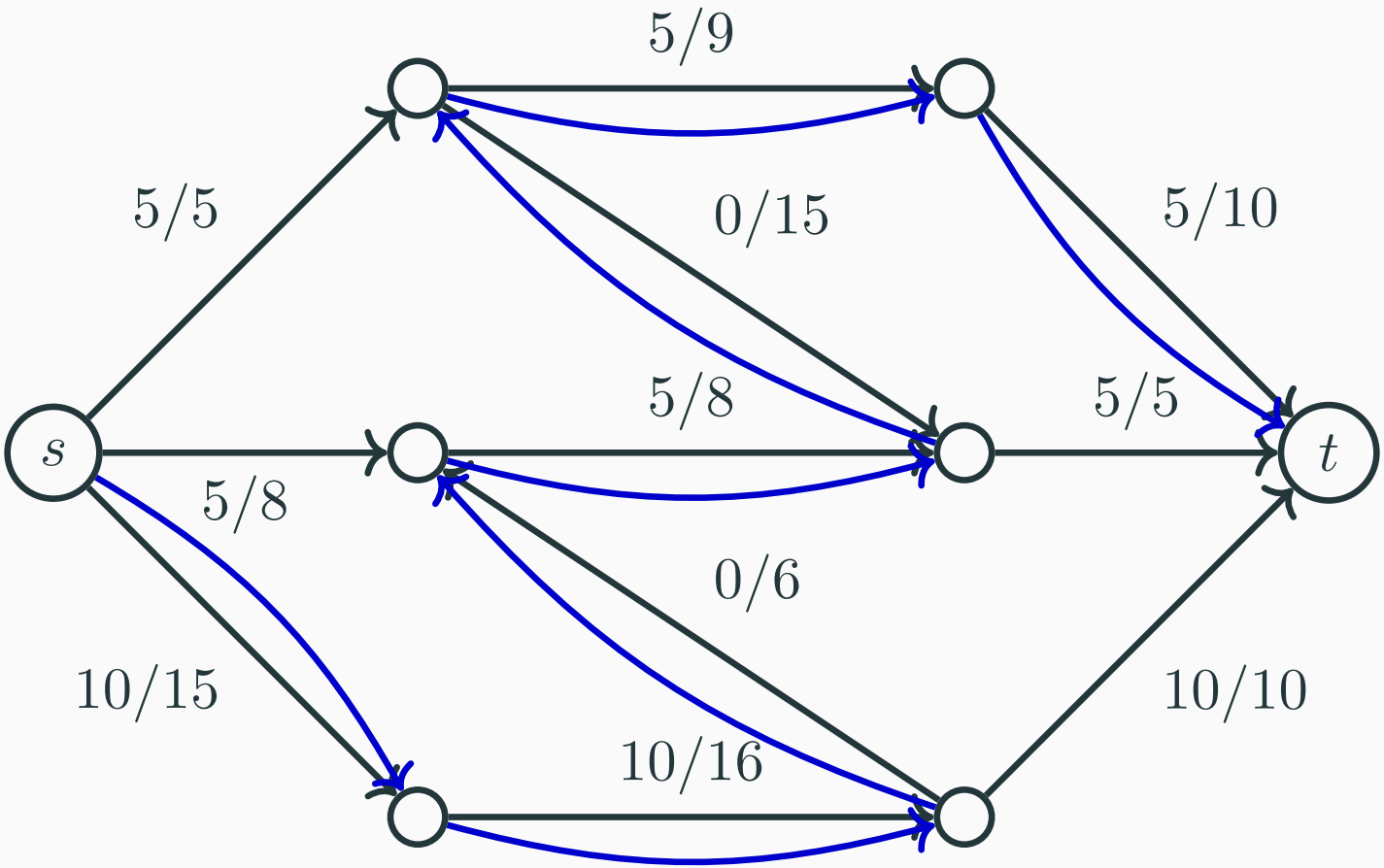
Example



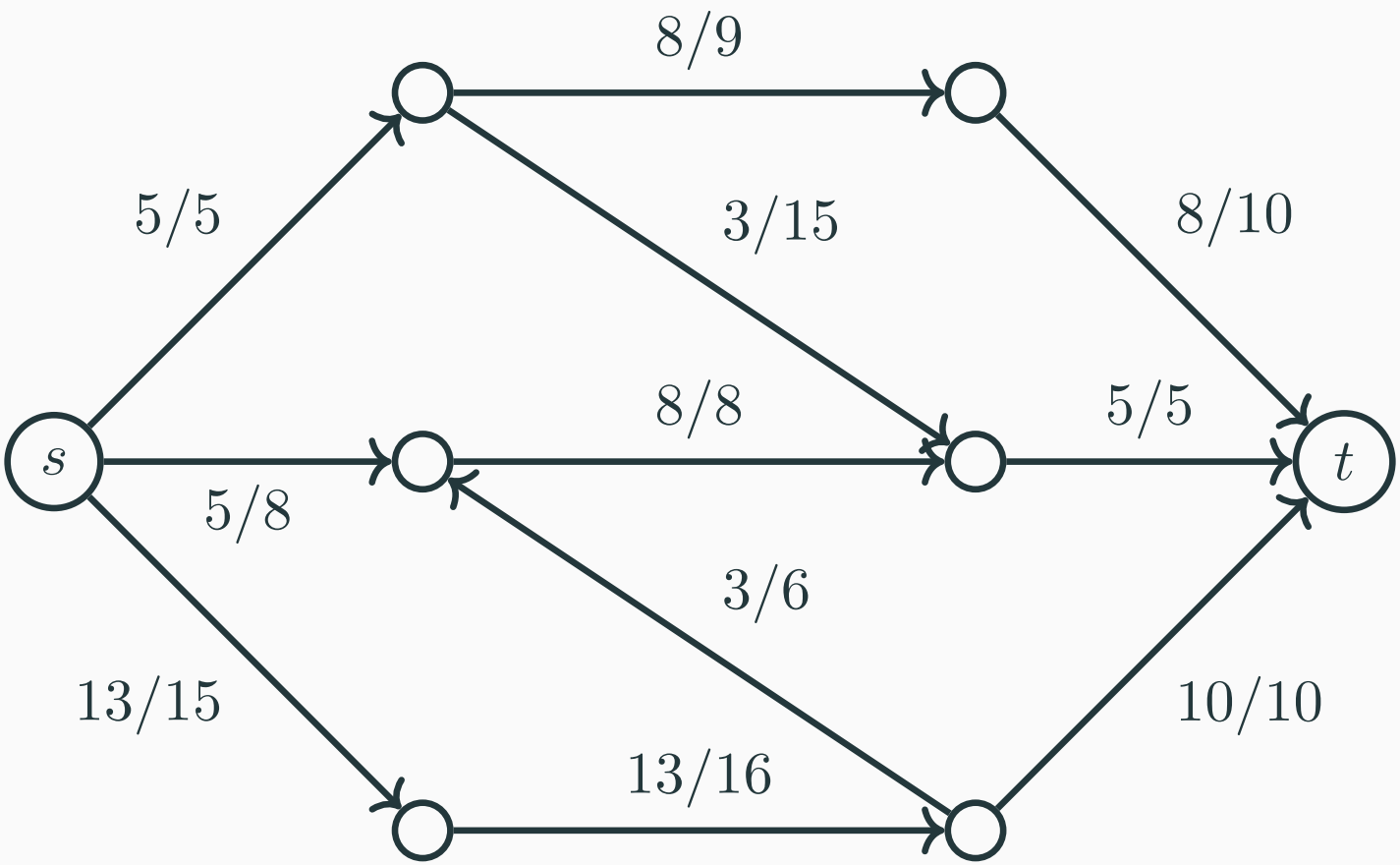
Example



Example



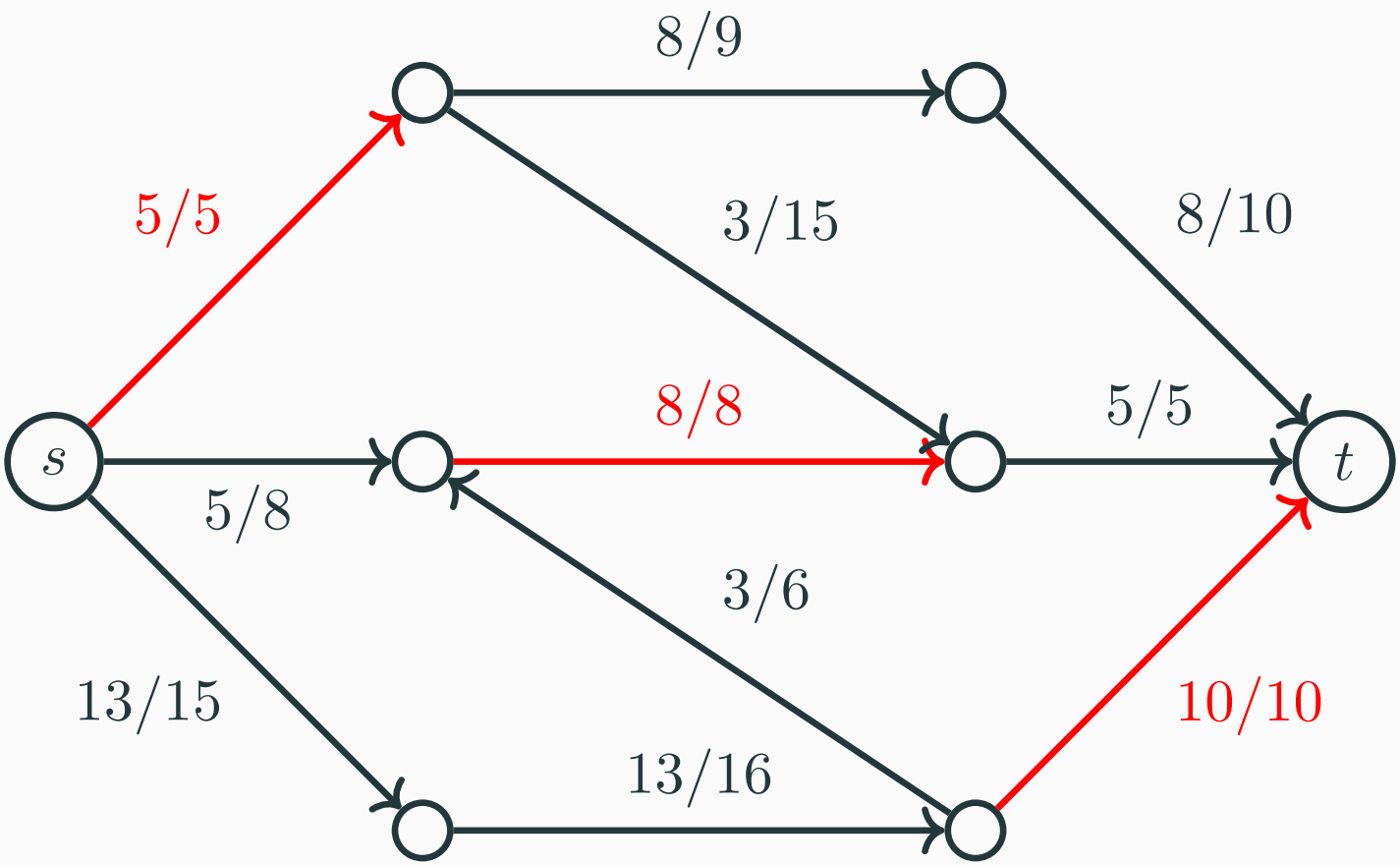
Example



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



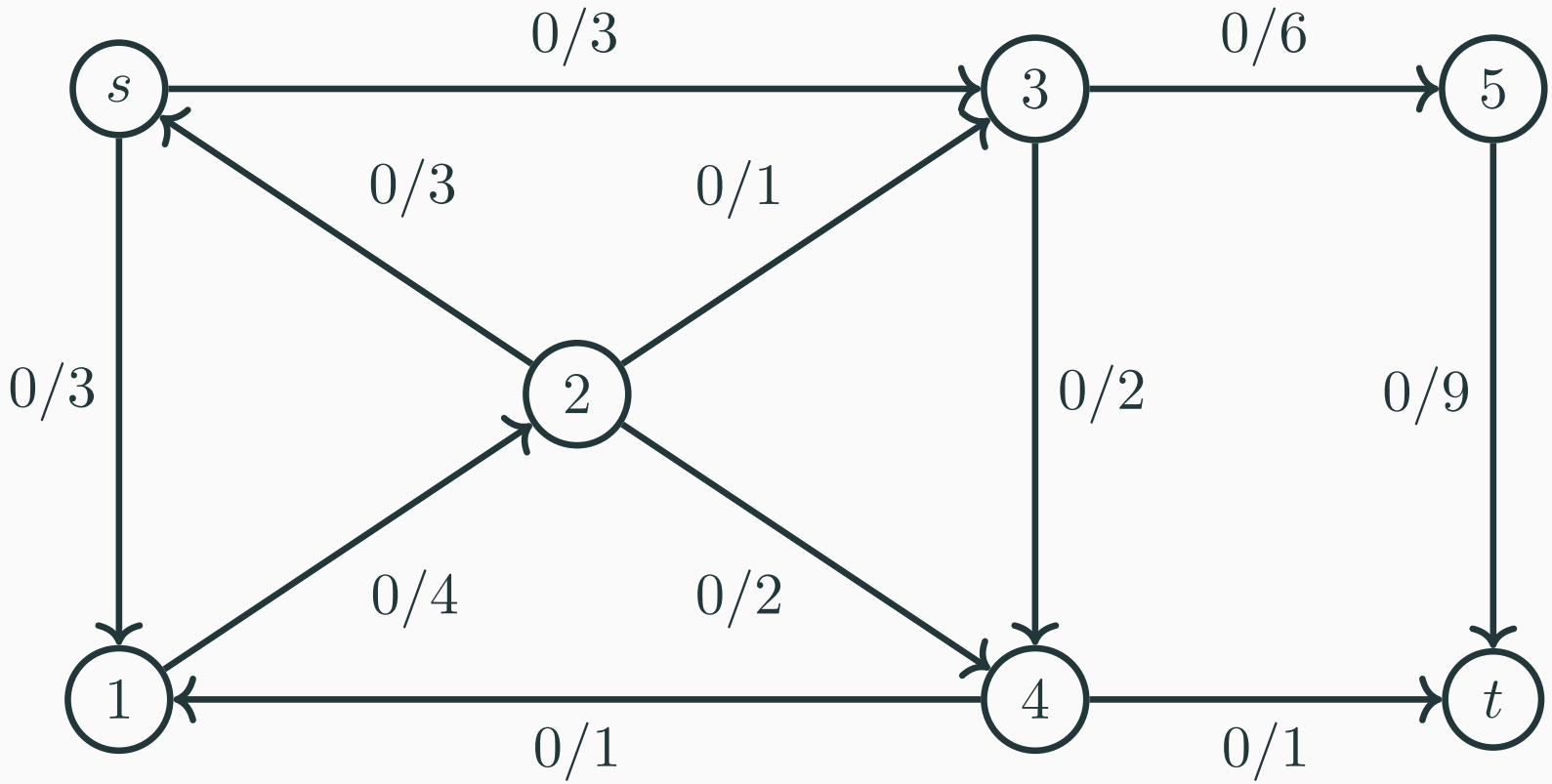
Max-flow Min-cut

- 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.

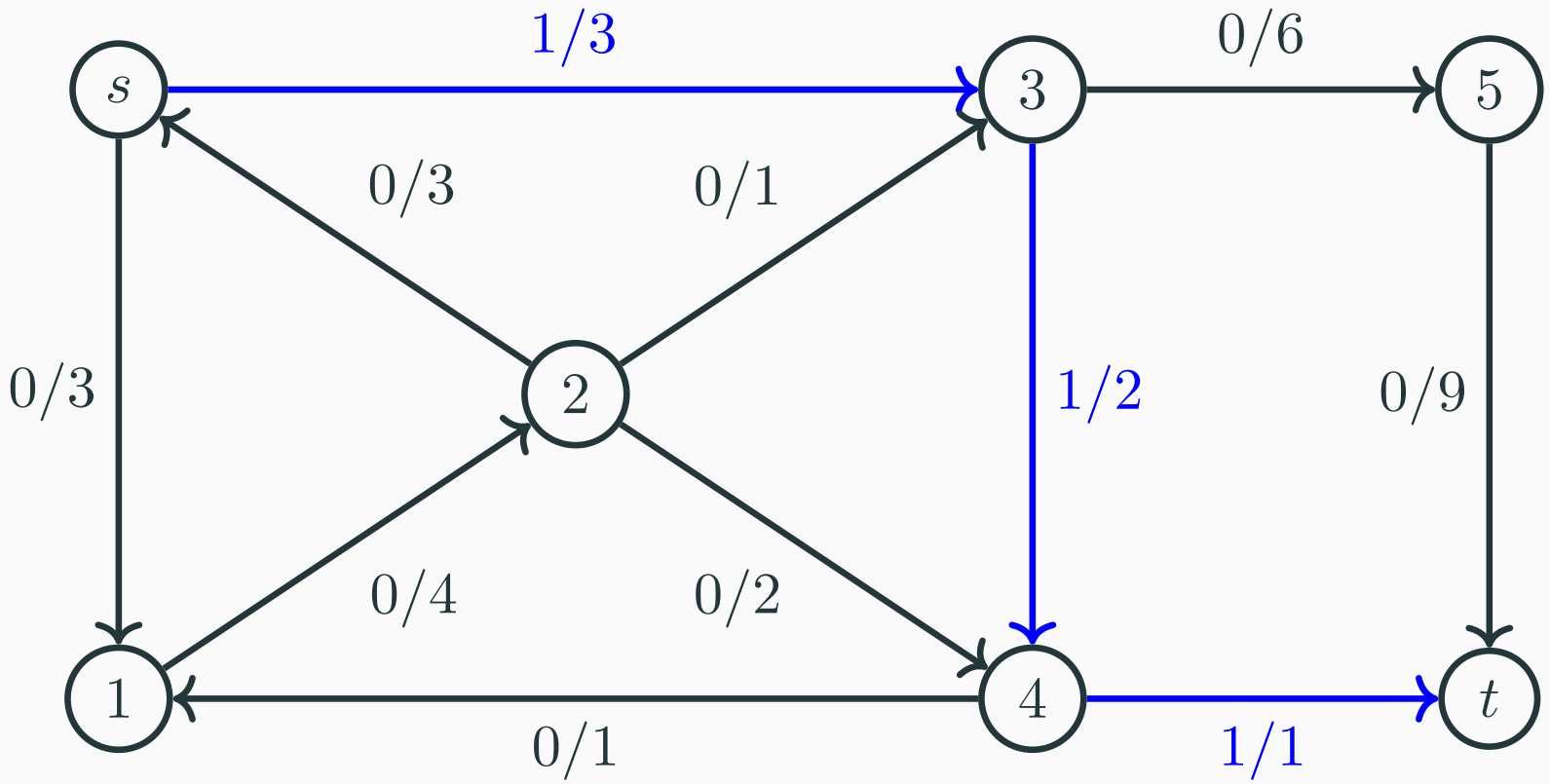
Max-flow Min-cut

- 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.
- But our algorithm is missing something to get all the way there!

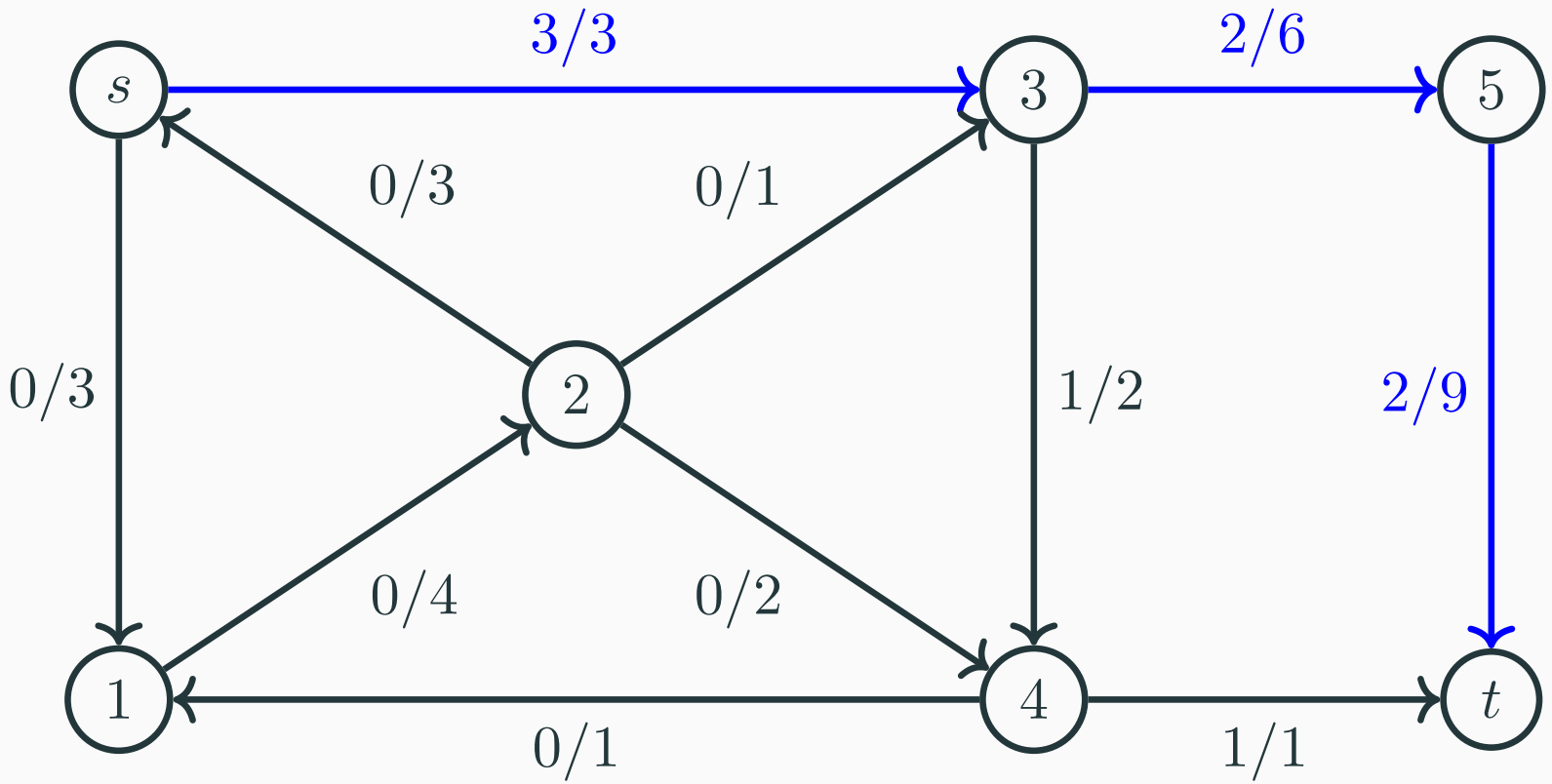
Example



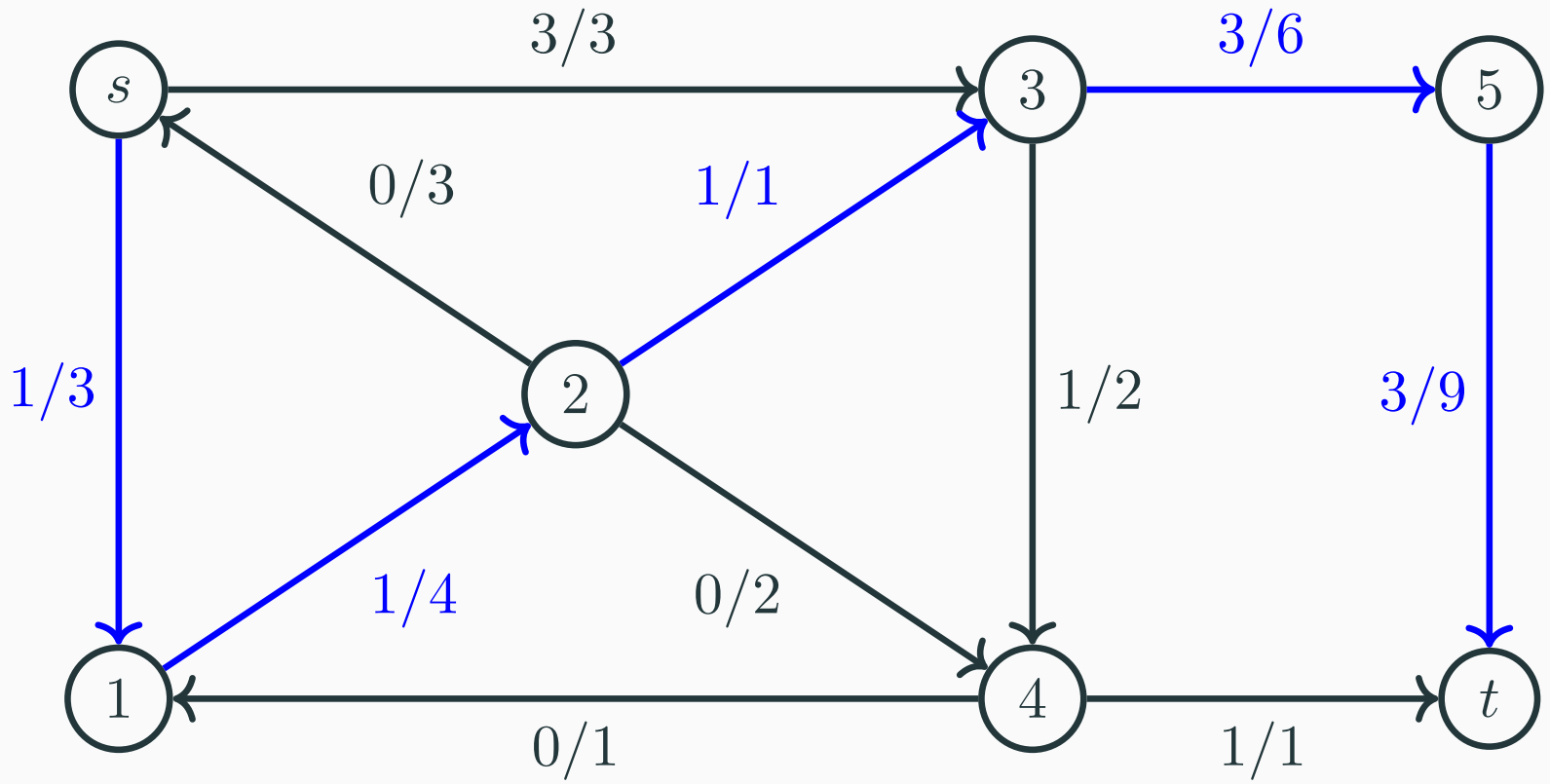
Example



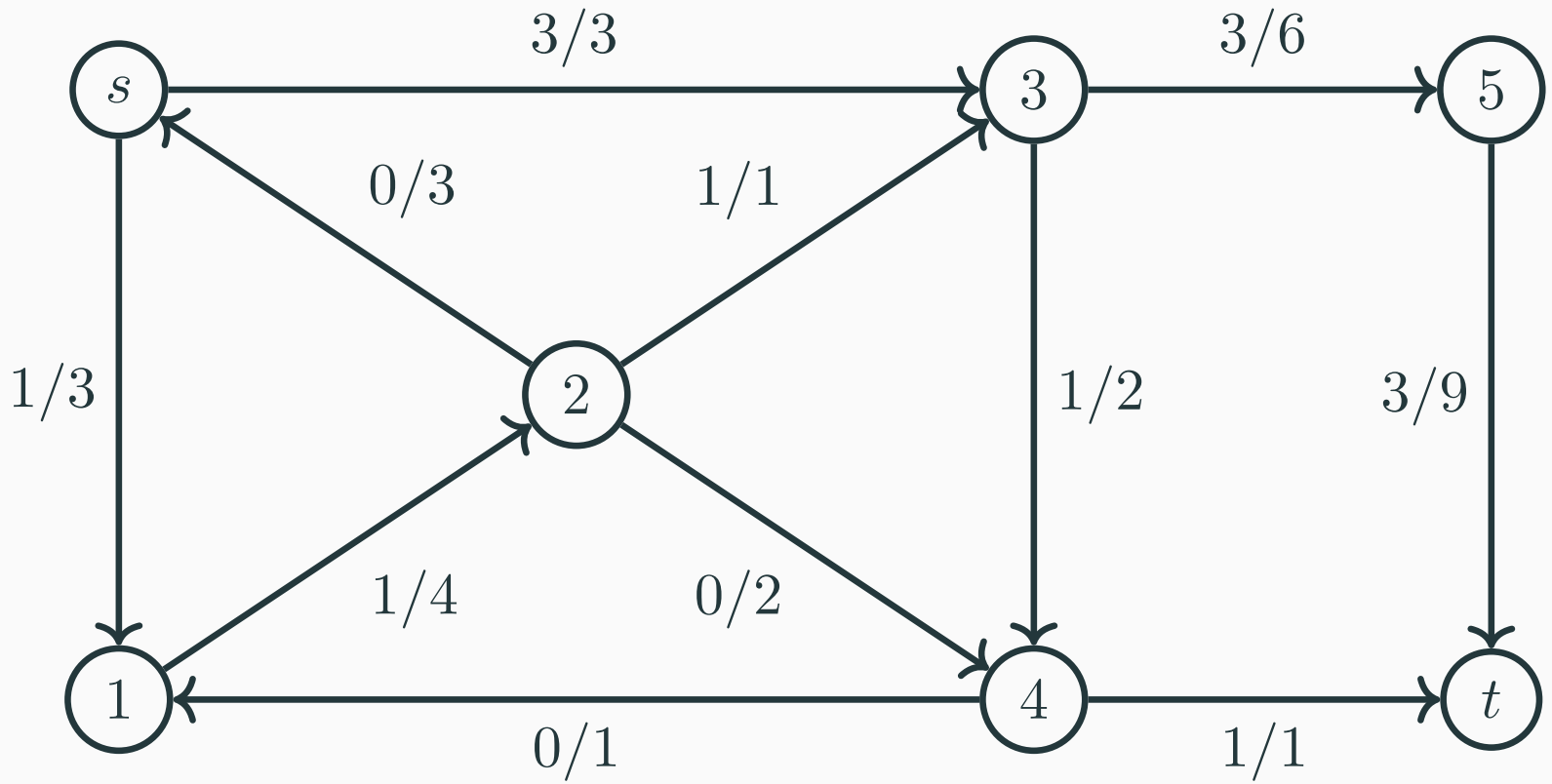
Example



Example



Example



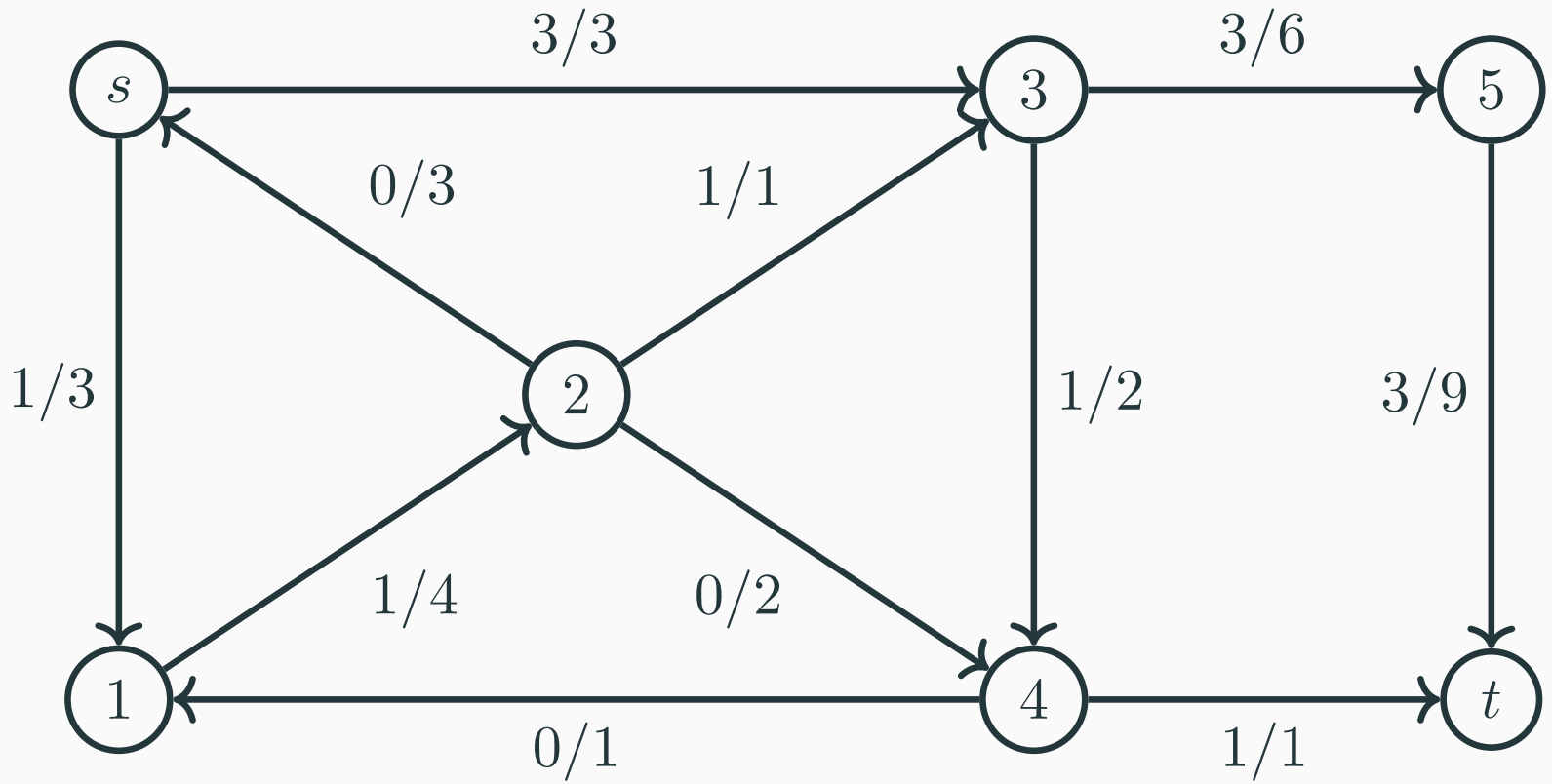
Now what?

- We're stuck! The max flow is not 4, and there is no min cut.
- But there is no way from s to t without using a full edge!
- What to do?

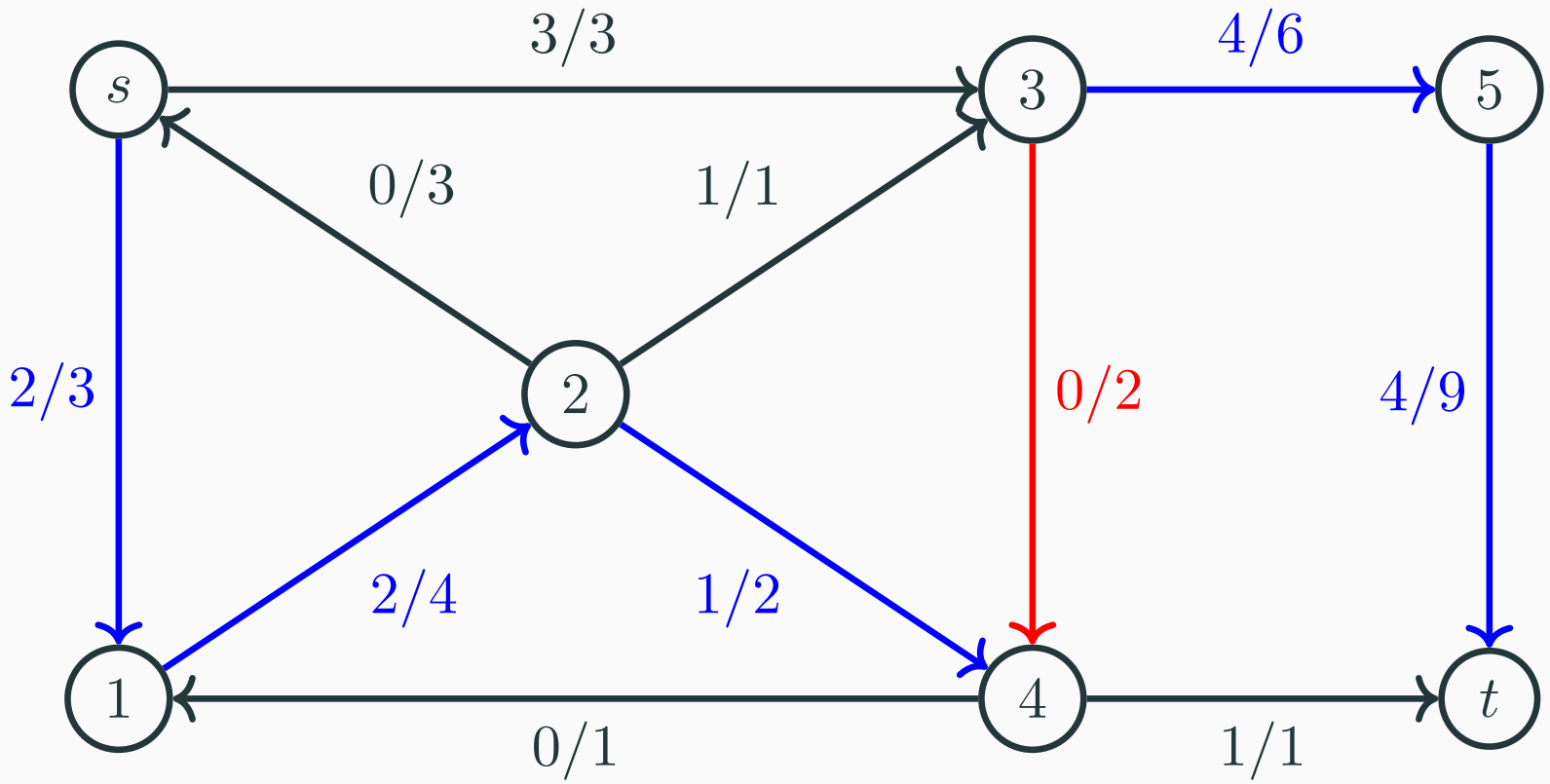
Now what?

- We're stuck! The max flow is not 4, and there is no min cut.
- But there is no way from s to t without using a full edge!
- What to do?
- What if we can "borrow" flow that's already been assigned?
- Then we can assign some more.

Example



Example



Reverse edges

- "Stealing" flow from an edge and following it backwards maintains all conditions we need.
- To program this the easiest way is just to add a backwards edge everywhere.
- If we add flow to a forward edge we subtract it from the backward edge, and vice versa.
- It can be proven that with this addition the algorithm always finds a max flow (try to think why).

Integers

- 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]
```

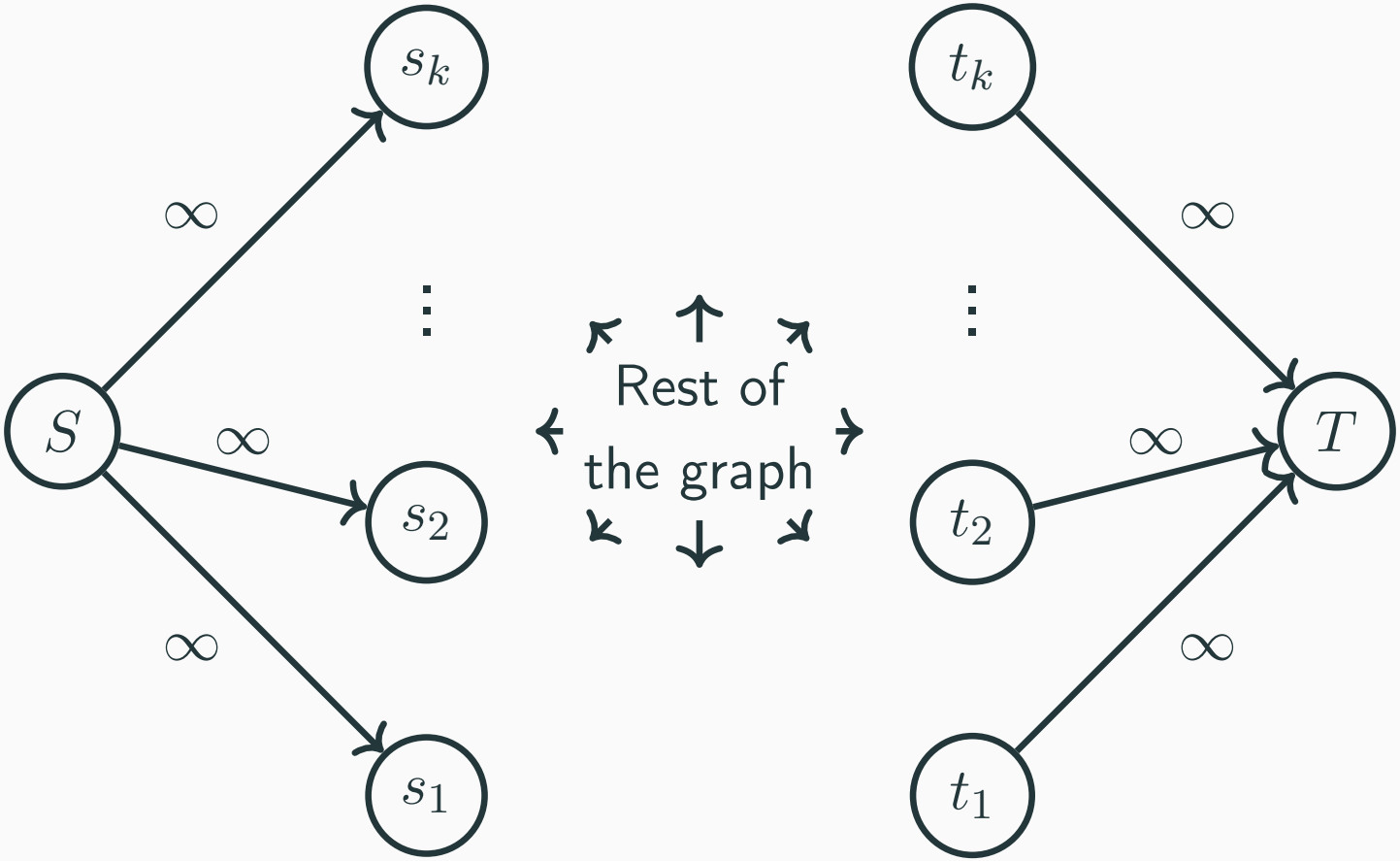
Implementation, part 4

```
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?

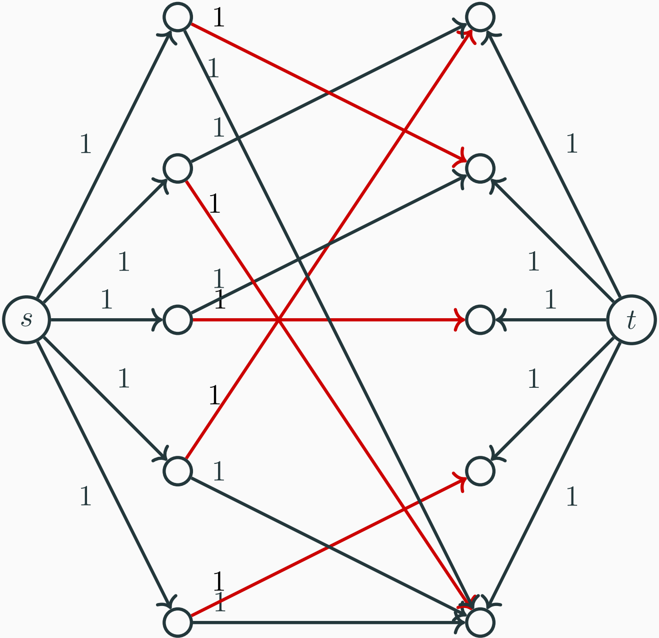
Multi source/sink



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.

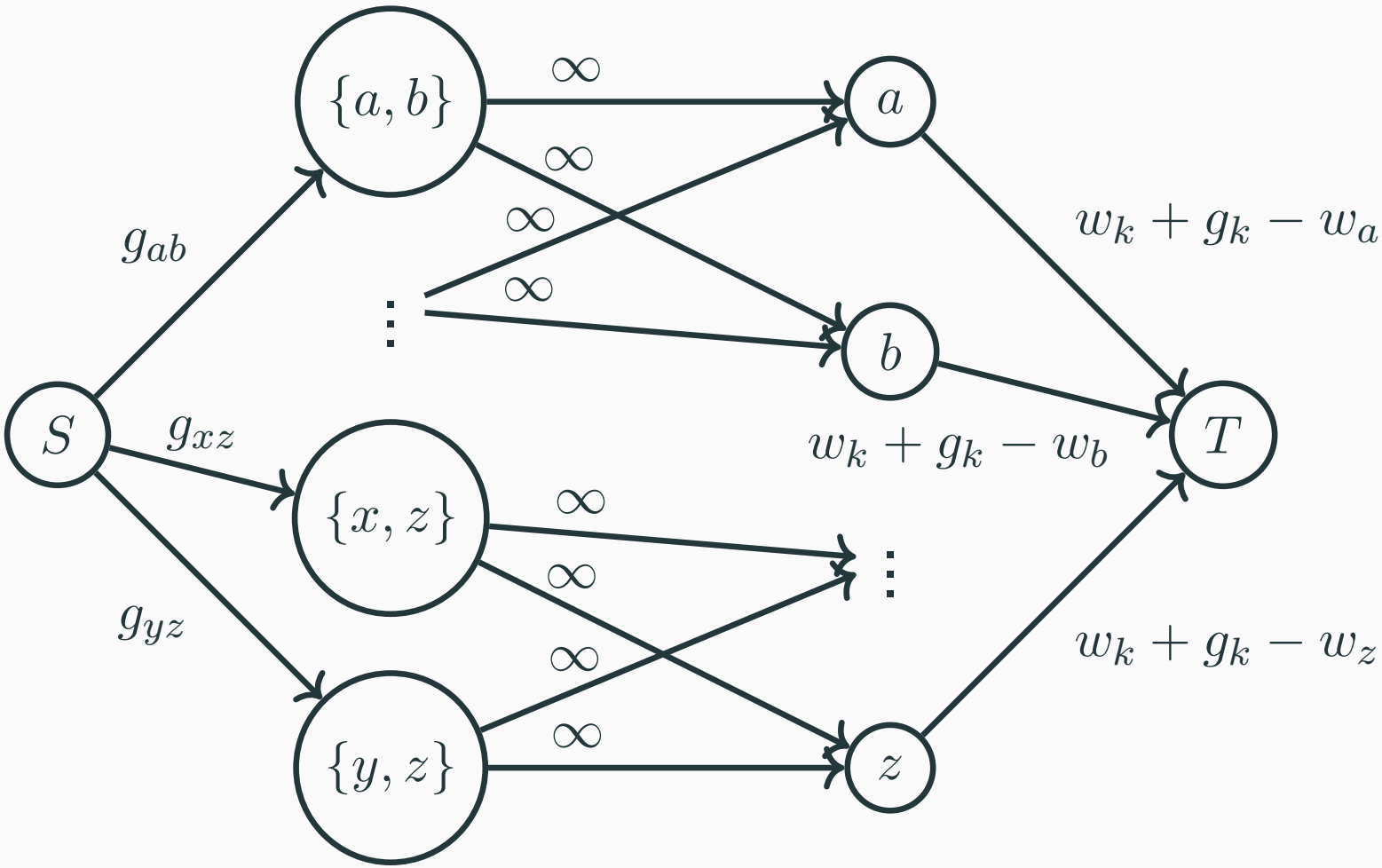
Baseball league

- Let's consider an example from baseball.
- During a league a fixed schedule of games will be played, whichever teams has strictly the most wins at the end is the winner (let's just say for ties there are no winners).
- At some point in the league you wonder, can your favourite team x win the league?

Baseball league

- Clearly it's best for team x to win all remaining games, so let's assume that's the case.
- Then we need to choose a winner for all other games such that no team gets as many wins as team x has in this hypothetical full win scenario.
- This is an assignment problem of games to winners, so we can model it with maximum flow.
- Each game (or rather pairing of teams) gets a node which can flow to winners, and the winners have a limit of flow to the sink based on how many wins would put them ahead of x.
- Then we just check if this graph has a saturating flow!

Baseball league



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 algorithm

- Edmond-Karp is the original polynomial algorithm for maximum-flow, but there are (many) others.

Method	Year	Complexity	Description
Edmonds–Karp algorithm[16]	1969	$O(VE^2)$	A specialization of Ford–Fulkerson, finding augmenting paths with breadth-first search.
Dinic's algorithm[17]	1970	$O(V^2E)$ (arbitrary weights) $O(\min\{V^{2/3}, E^{1/2}\}E)$ (unit weights)	Repeated phases that build a "layered" subgraph of residual graph edges belonging to shortest paths, using breadth-first search, and then find a blocking flow (a maximum flow in this layered graph) in time $O(VE)$ per phase. The shortest path length increases in each phase so there are at most $V-1$ phases.
Karzanov's algorithm[18]	1974	$O(V^3)$	A predecessor to the push-relabel algorithm using preflows (flow functions allowing excess at vertices) to find a blocking flow in each phase of Dinic's algorithm in time $O(V^2)$ per phase. The first cubic-time flow algorithm.
Cherkassky's algorithm[19]	1977	$O(V^2\sqrt{E})$	Combines the blocking flow methods of Dinic (within blocks of consecutive BFS layers) and Karzanov (to combine blocks). The first subcubic strongly polynomial time bound for sparse graphs. Remained best for some values of E until KRT 1988.
Malhotra, Kumar, and Maheshwari[20]	1978	$O(V^3)$	Not an improvement in complexity over Karzanov, but a simplification. Finds blocking flows by repeatedly finding a "reference node" of the layered graph and a flow that saturates all its incoming or outgoing edges, in time proportional to the number of nodes plus the number of saturated edges.
Galil's algorithm[21]	1978	$O(V^{5/3}E^{2/3})$	Modifies Cherkasky's algorithm by replacing the method for finding flows within blocks of consecutive layers.

Dinic's algorithm

- Edmond-Karp is the original polynomial algorithm for maximum-flow, but there are (many) others.

Galil, Naamad, and Shiloach[22] [23]	1978	$O(VE(\log V)^2)$	Uses tree contraction on a breadth-first search forest of the layered graph to speed up blocking flows. The first of many $O(VE \operatorname{polylog} V)$ algorithms, still the best polynomial exponents for a strongly polynomial algorithm.
Blocking flow with link/cut trees.[24]	1981	$O(VE \log V)$	Introduces the link/cut tree data structure and uses it to find augmenting paths in layered networks in logarithmic amortized time per path.
Push–relabel algorithm with link/cut trees[25]	1986	$O\left(VE \log \frac{V^2}{E}\right)$	The push-relabel algorithm maintains a preflow, and a height function estimating residual distance to the sink. It modifies the preflow by pushing excess to lower-height vertices and increases the height function at vertices without residual edges to lower heights, until all excess returns to the source. Link/cut trees allow pushes along paths rather than one edge at a time.
Cheriyan and Hagerup[26]	1989	randomized, $O\left(VE + V^2(\log V)^2\right)$ with high probability	Push-relabel on a subgraph to which one edge is added at a time, prioritizing pushes of high excess amounts, with randomly permuted adjacency lists
Alon[27]	1989	$O(VE + V^{8/3} \log V)$	Derandomization of Cheriyan and Hagerup
Cheriyan, Hagerup, and Mehlhorn[28]	1990	$O\left(\frac{V^3}{\log V}\right)$	Uses Alon's derandomization of Cheriyan and Hagerup with ideas related to the Method of Four Russians to speed up the search for height-reducing edges on which to push excess.
King, Rao, and Tarjan[29]	1992	$O(VE + V^{2+\varepsilon})$ for any $\varepsilon > 0$	Another derandomization of Cheriyan and Hagerup. Preliminary version of King, Rao, and Tarjan 1994 with weaker bounds.
Phillips and Westbrook[30]	1993	$O(VE \log_{E/V} V + V(\log V)^{2+\varepsilon})$ for any $\varepsilon > 0$	Improved from King, Rao, and Tarjan 1992 using similar ideas.
King, Rao, and Tarjan[31]	1994	$O\left(VE \log_{\frac{E}{V \log V}} V\right)$	Improved from Phillips and Westbrook using similar ideas.

Dinic's algorithm

- Edmond-Karp is the original polynomial algorithm for maximum-flow, but there are (many) others.

Orlin[9]	2013	$O(VE)$	Applies a pseudopolynomial algorithm of Goldberg and Rao to a compressed network, maintained using data structures for dynamic transitive closure. Takes time $O\left(VE + E^{31/16}(\log V)^2\right)$, which simplifies to $O(VE)$ for $E = O(V^{16/15-\varepsilon})$, while previous bounds simplify to $O(VE)$ for $E = \Omega(V^{1+\epsilon})$.
Orlin and Gong[32]	2021	$O\left(\frac{VE\log V}{\log\log V + \log\frac{E}{V}}\right)$	Based on a pseudopolynomial algorithm of Ahuja, Orlin, and Tarjan. Faster than King, Rao, and Tarjan and does not use link/cut trees, but not faster than Orlin + KRT.

Dinic's algorithm

- Edmond-Karp is the original polynomial algorithm for maximum-flow, but there are (many) others.

Method	Year	Complexity	Description
Ford–Fulkerson algorithm[33]	1956	$O(EU)$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on that path.
Binary blocking flow algorithm[34]	1998	$O\left(E \cdot \min\{V^{2/3}, E^{1/2}\} \cdot \log \frac{V^2}{E} \cdot \log U\right)$	
Kathuria-Liu-Sidford algorithm[35]	2020	$E^{4/3+o(1)}U^{1/3}$	Interior point methods and edge boosting using ℓ_p-norm flows. Builds on earlier algorithm of Madry, which achieved runtime $\tilde{O}(E^{10/7}U^{1/7})$.[36]
BLNPSSSW / BLLSSSW algorithm[37] [38]	2020	$\tilde{O}((E+V^{3/2})\log U)$	Interior point methods and dynamic maintenance of electric flows with expander decompositions.
Gao-Liu-Peng algorithm[39]	2021	$\tilde{O}(E^{\frac{3}{2}-\frac{1}{328}}\log U)$	Gao, Liu, and Peng's algorithm revolves around dynamically maintaining the augmenting electrical flows at the core of the interior point method based algorithm from [Mądry JACM '16]. This entails designing data structures that, in limited settings, return edges with large electric energy in a graph undergoing resistance updates.
Chen, Kyng, Liu, Peng, Gutenberg and Sachdeva's algorithm[10]	2022	$O(E^{1+o(1)}\log U)$ The exact complexity is not stated clearly in the paper, but appears to be $E\exp O(\log^{7/8} E \log\log E)\log U$	Chen, Kyng, Liu, Peng, Gutenberg and Sachdeva's algorithm solves maximum flow and minimum-cost flow in almost linear time by building the flow through a sequence of $E^{1+o(1)}$ approximate undirected minimum-ratio cycles, each of which is computed and processed in amortized $E^{o(1)}$ time using a dynamic data structure.

Dinic's algorithm

- Edmond-Karp is the original polynomial algorithm for maximum-flow, but there are (many) others.

Bernstein, Blikstad, Saranurak, Tu[40]	2024	$O(n^{2+o(1)} \log U)$ randomized algorithm when the edge capacities come from the set $\{1, \ldots, U\}$	The algorithm is a variant of the push-relabel algorithm by introducing the *weighted* variant. The paper establishes a weight function on directed and acyclic graphs (DAG), and attempts to imitate it on general graphs using directed expander hierarchies, which induce a natural vertex ordering that produces the weight function similar to that of the DAG special case. The randomization aspect (and subsequently, the $n^{o(1)}$ factor) comes from the difficulty in applying directed expander hierarchies to the computation of *sparse cuts*, which do not allow for natural dynamic updating.

Dinic's algorithm

- Many of these are not that performant in practice as they don't start being faster until for unfeasibly large graphs.
- But there are two that are commonly implemented as performance improvements over Edmond-Karp.
- Those are Dinic's algorithm and the Push-Relabel algorithm.
- We will not look at Push-Relabel as it is quite technical, but I recommend it for interested students.
- Instead we look at Dinic's algorithm.

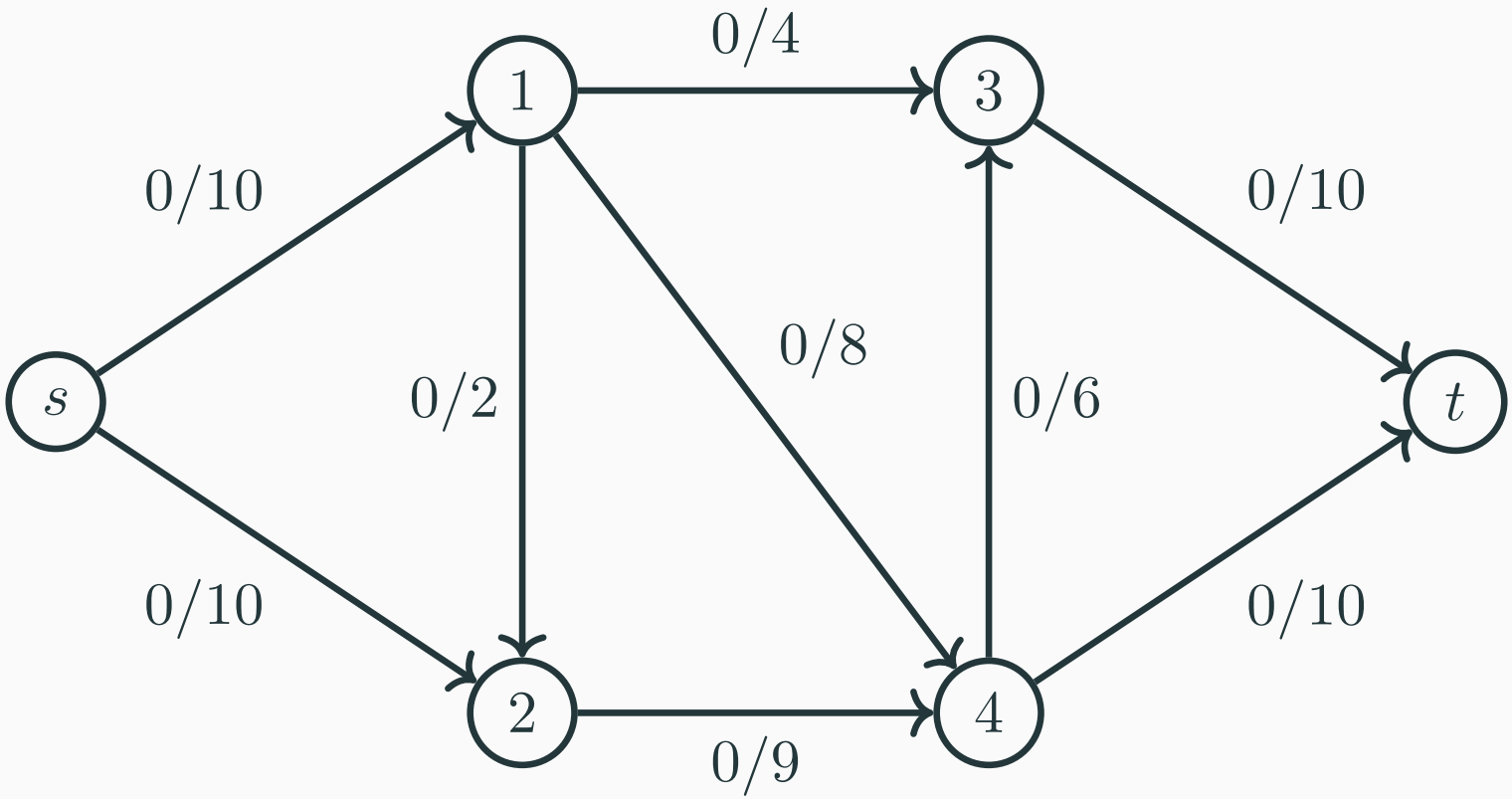
Dinic's algorithm

- We need two concepts.
- A blocking flow is a flow such that every path from the sink to the source has at least one saturated edge (note this is not necessarily a maximal flow).
- A layered network of a graph G is a graph constructed in the following manner:
  - Let $L(v)$ be the length of the shortest (unweighted) path from the source to v using only edges with capacity left.
  - Construct a subgraph of G keeping only the edges $u \to v$ such that $L(v) = L(u) + 1$.

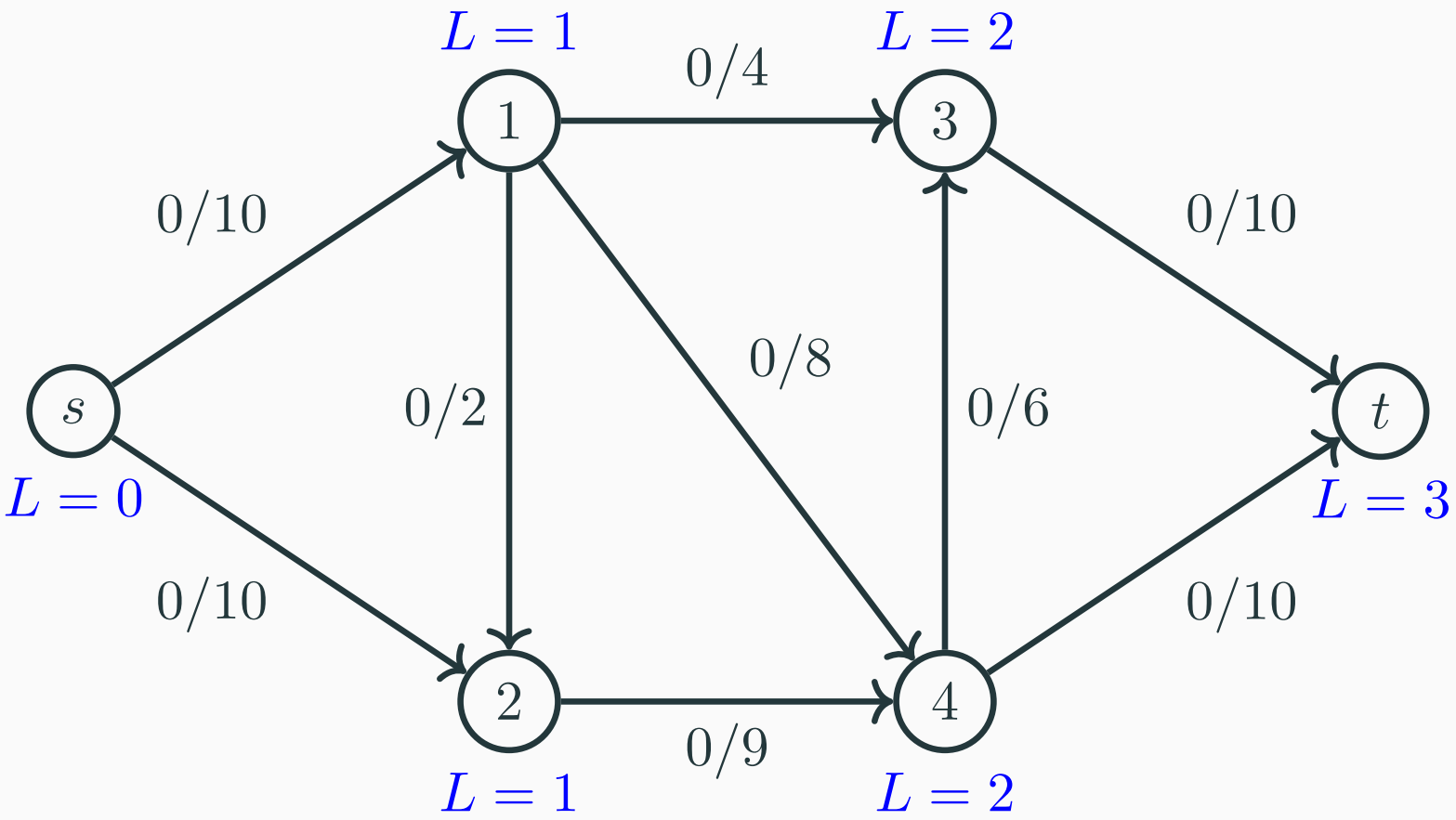
Dinic's algorithm

- The algorithm proceeds by repeatedly constructing the layered network of our graph.
- Each time we find a blocking flow in that layered network and add it to our total flow.
- If we get no flow in the layered network we terminate.
- If we found no flow, then the layered network has no unsaturated path from s to t and then G doesn't either, so the flow is already maximum.

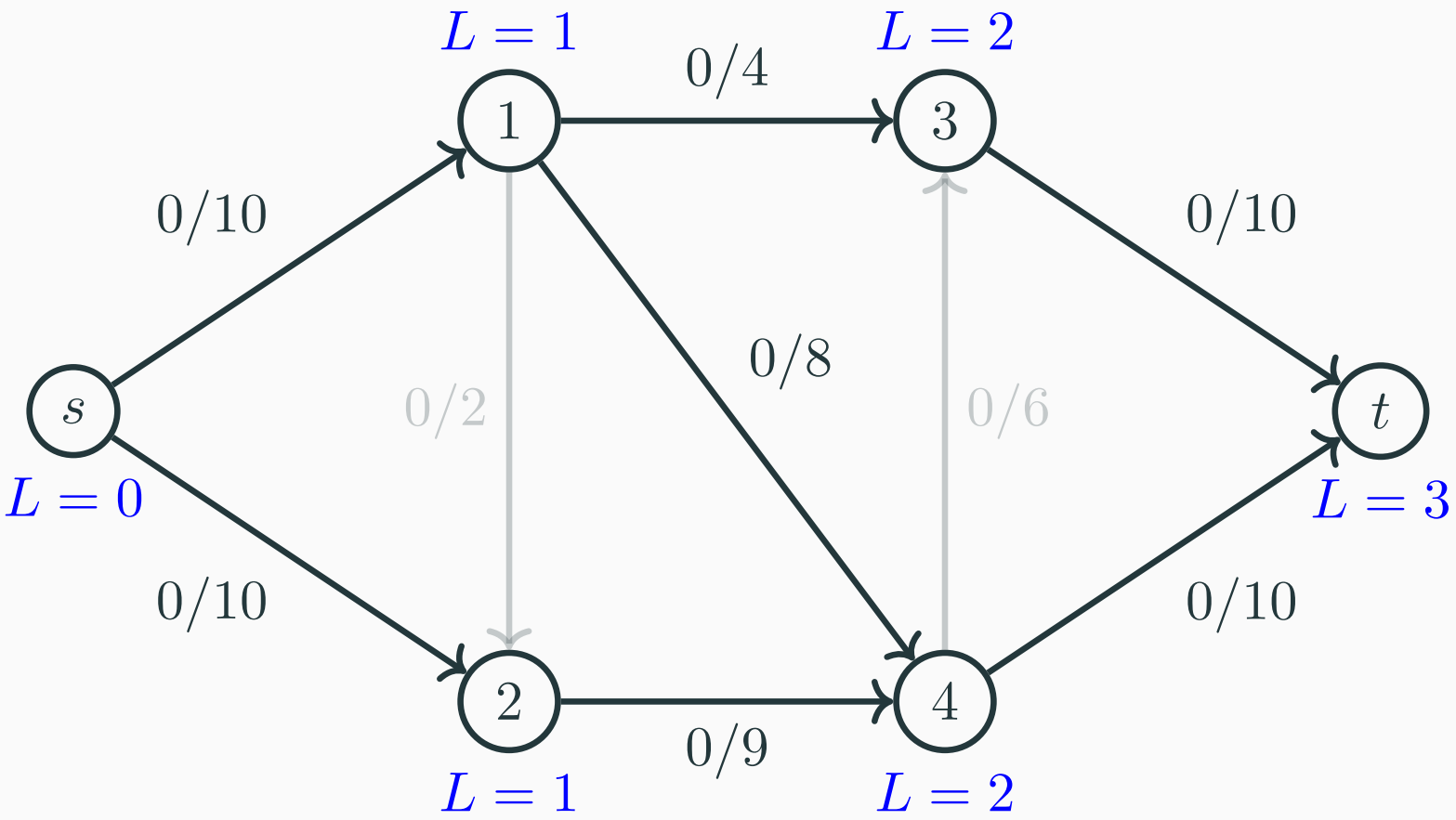
Example



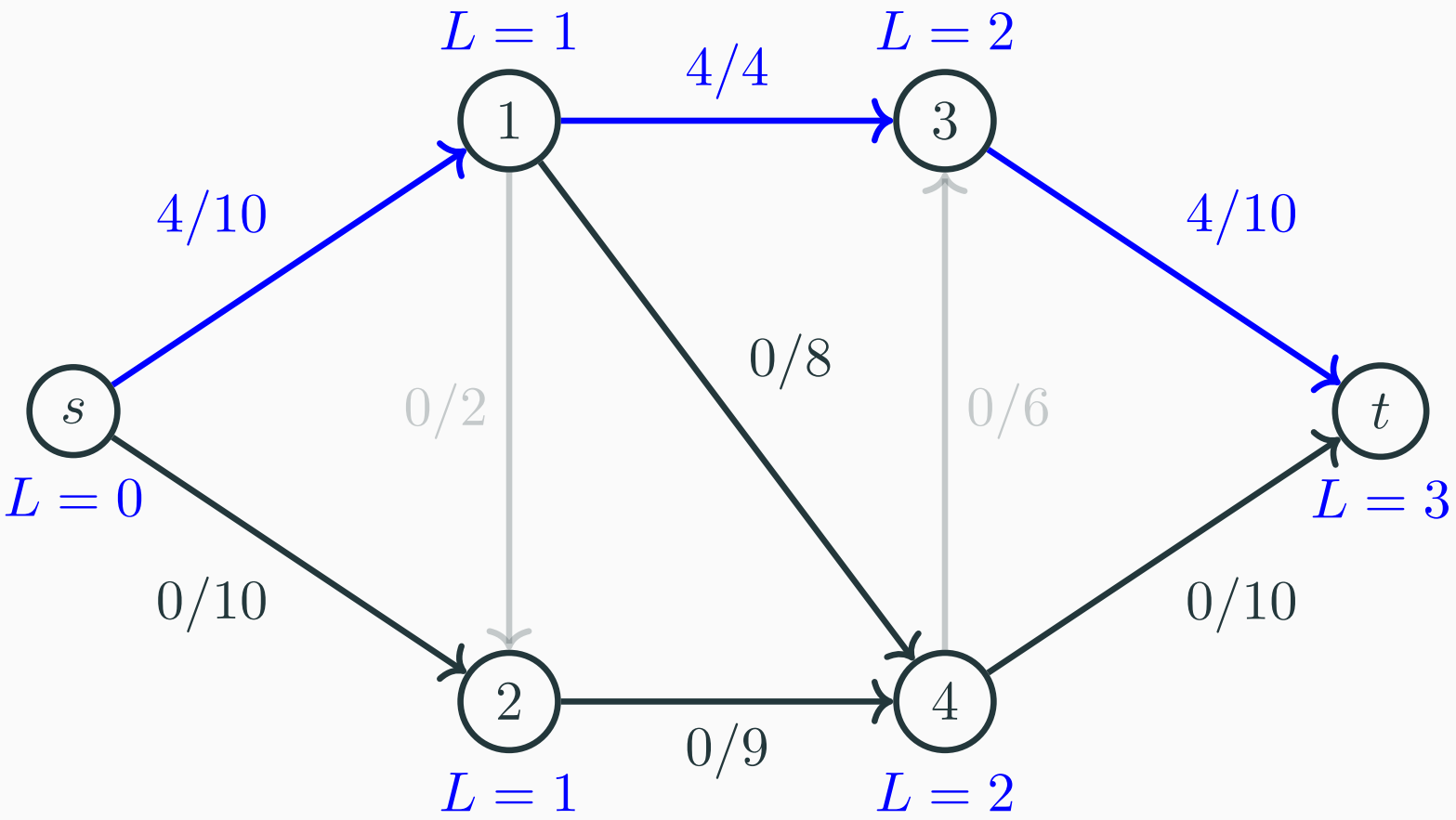
Example



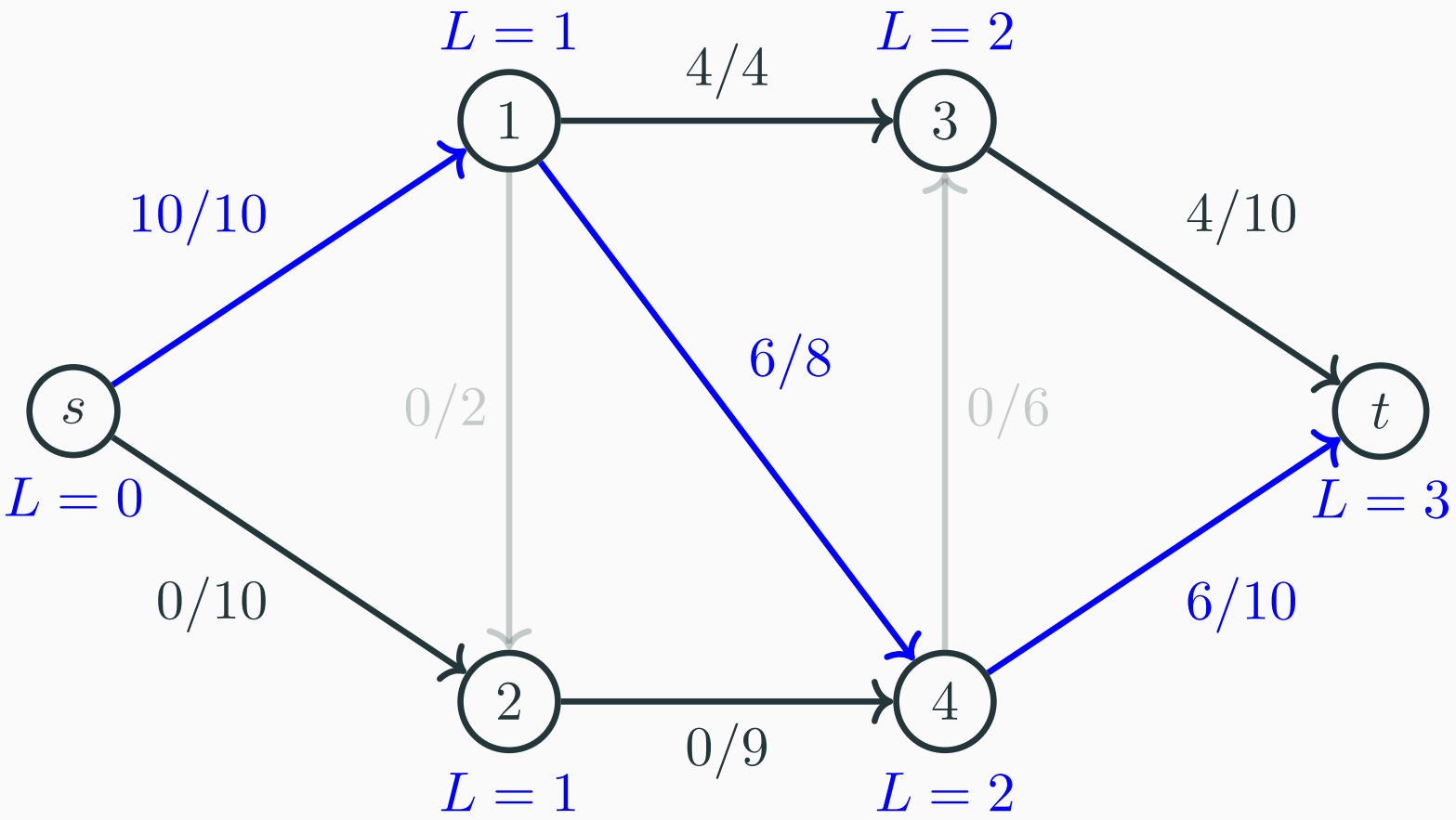
Example



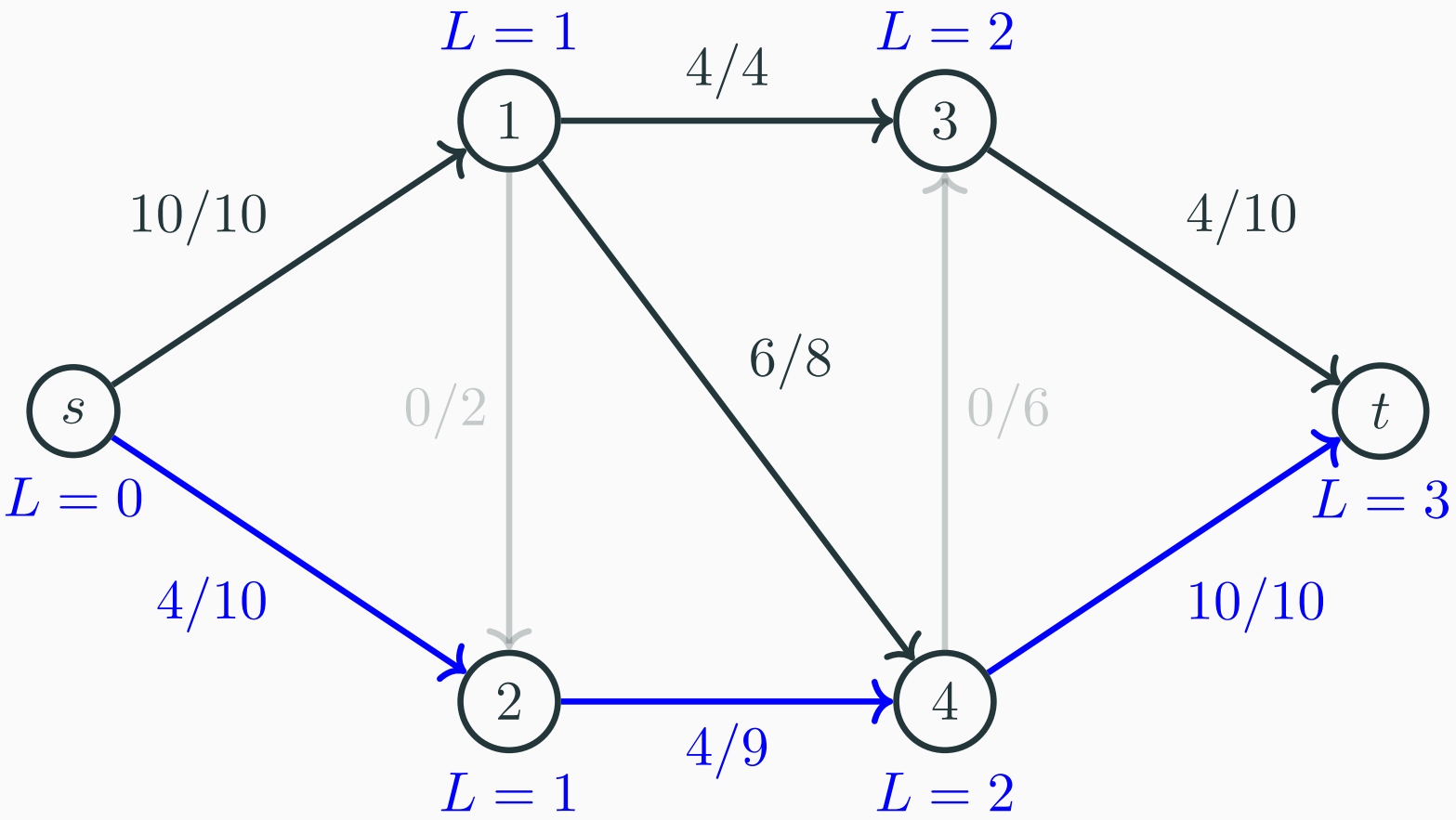
Example



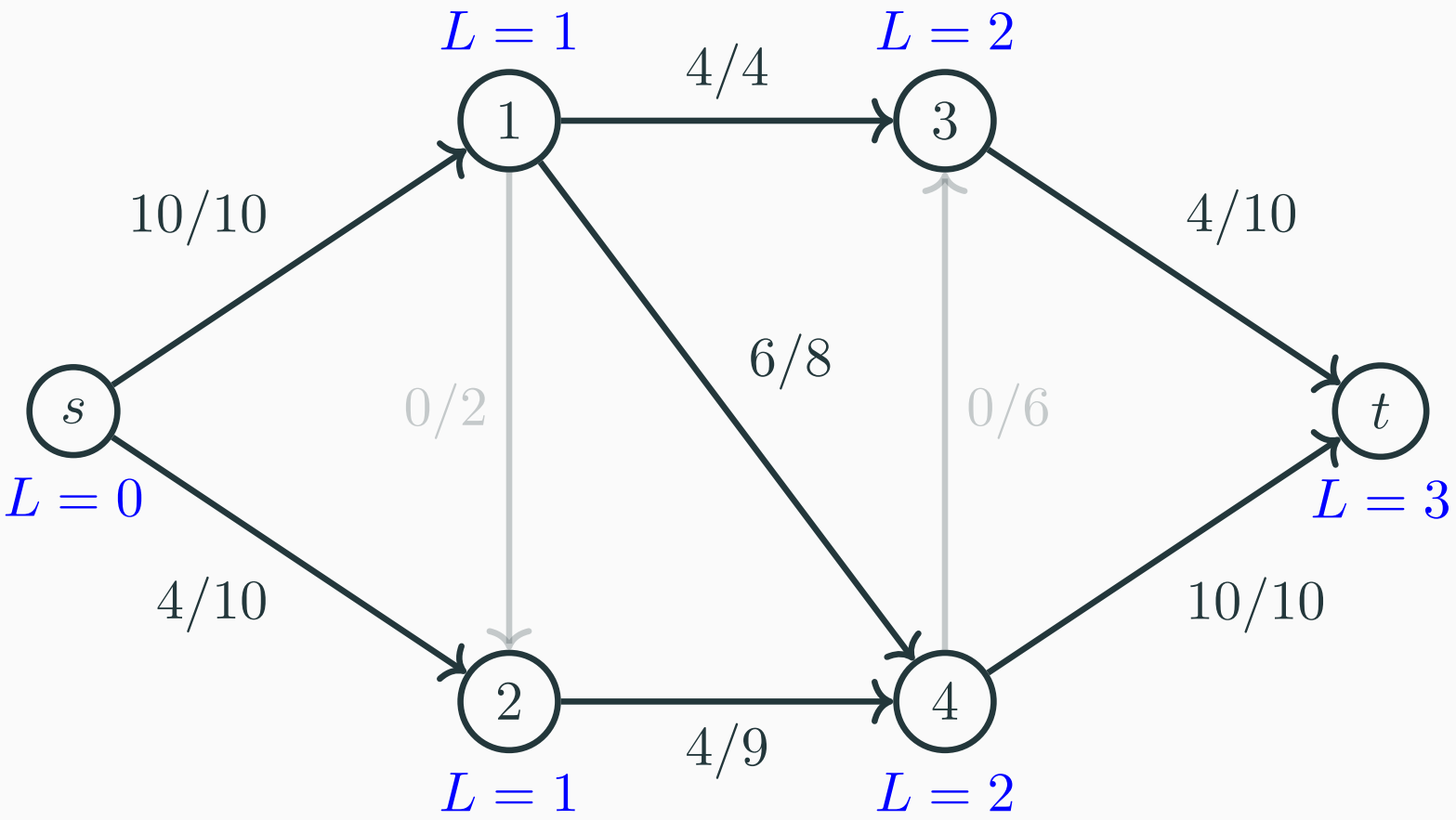
Example



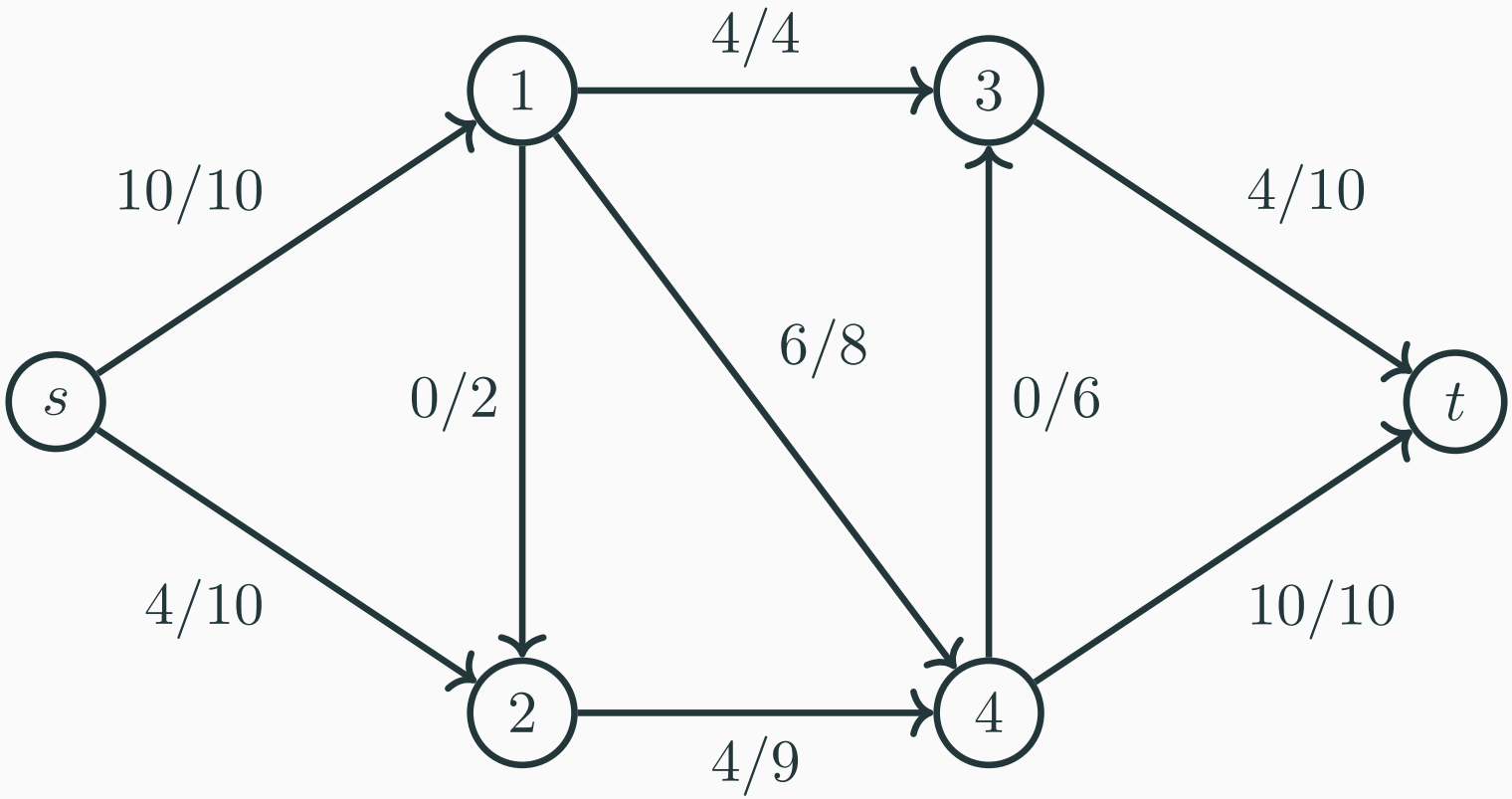
Example



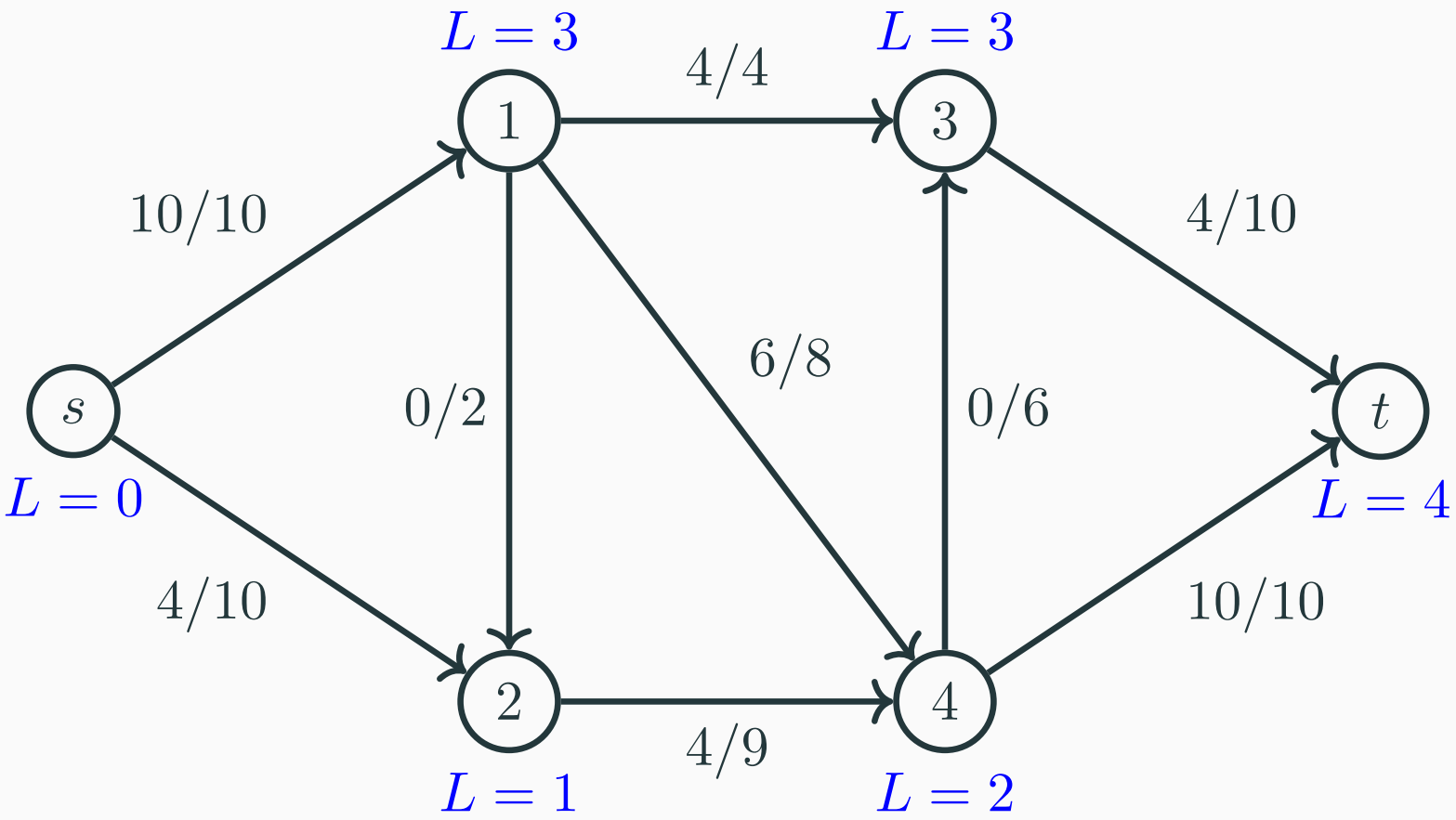
Example



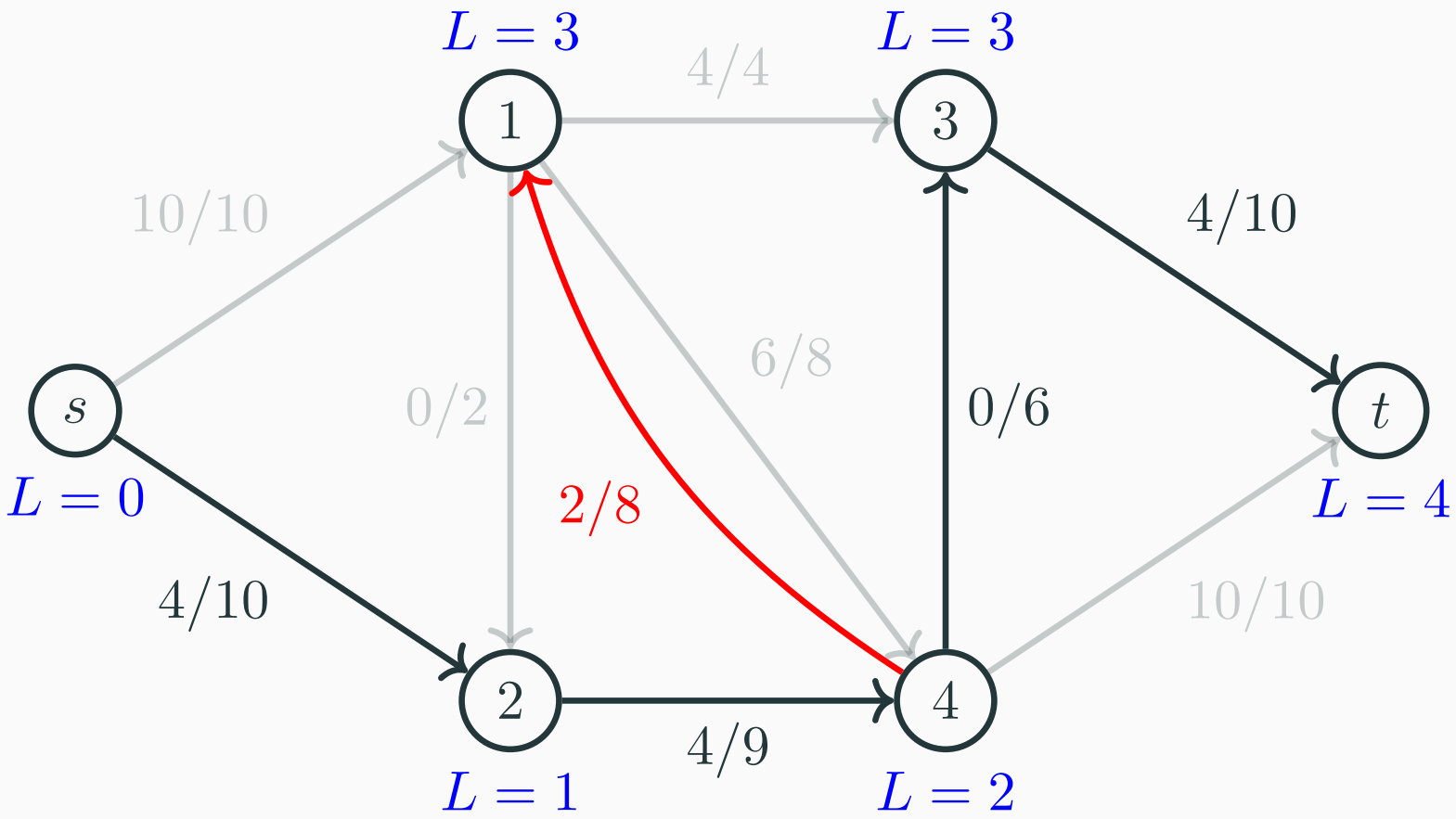
Example



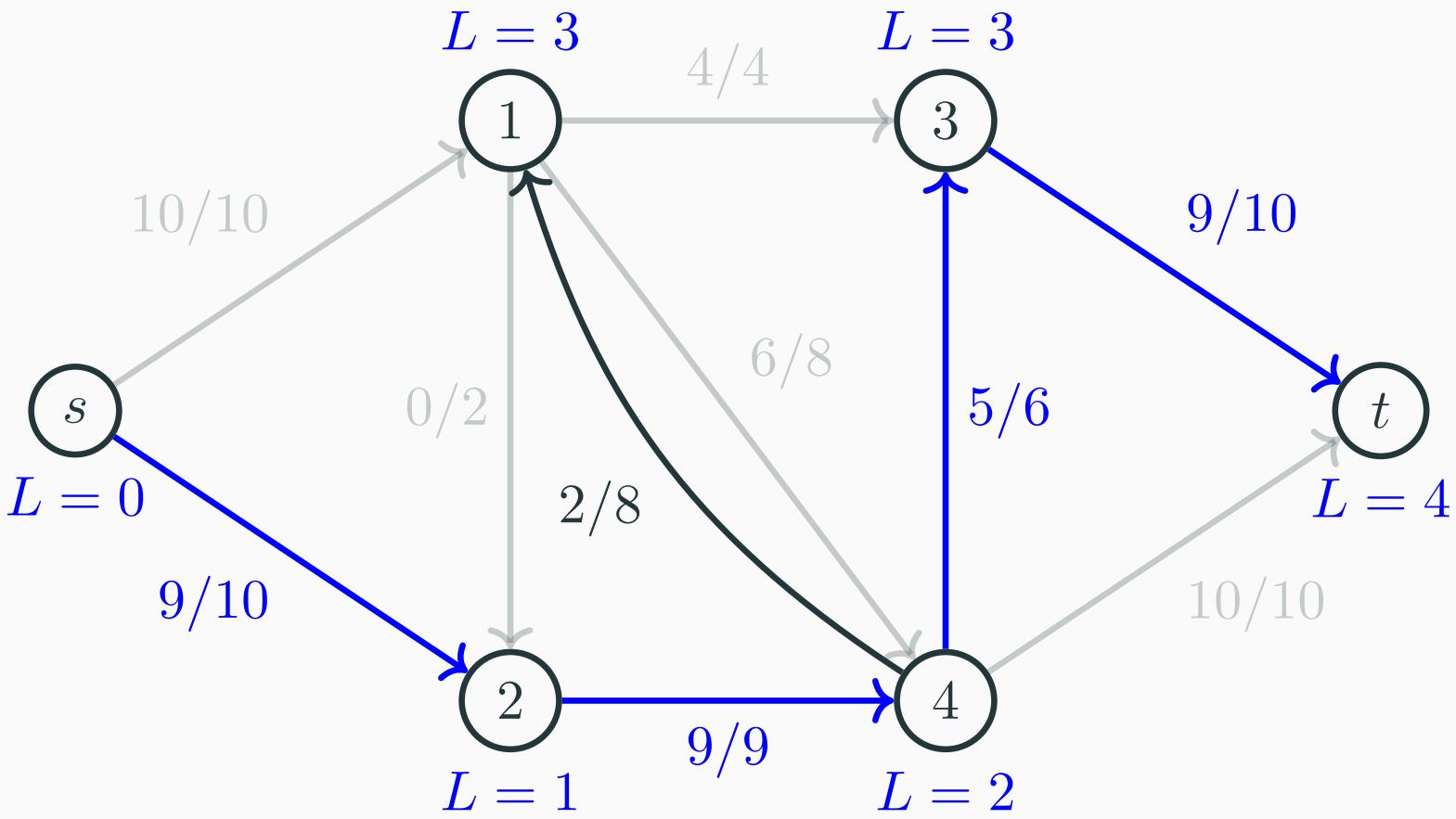
Example



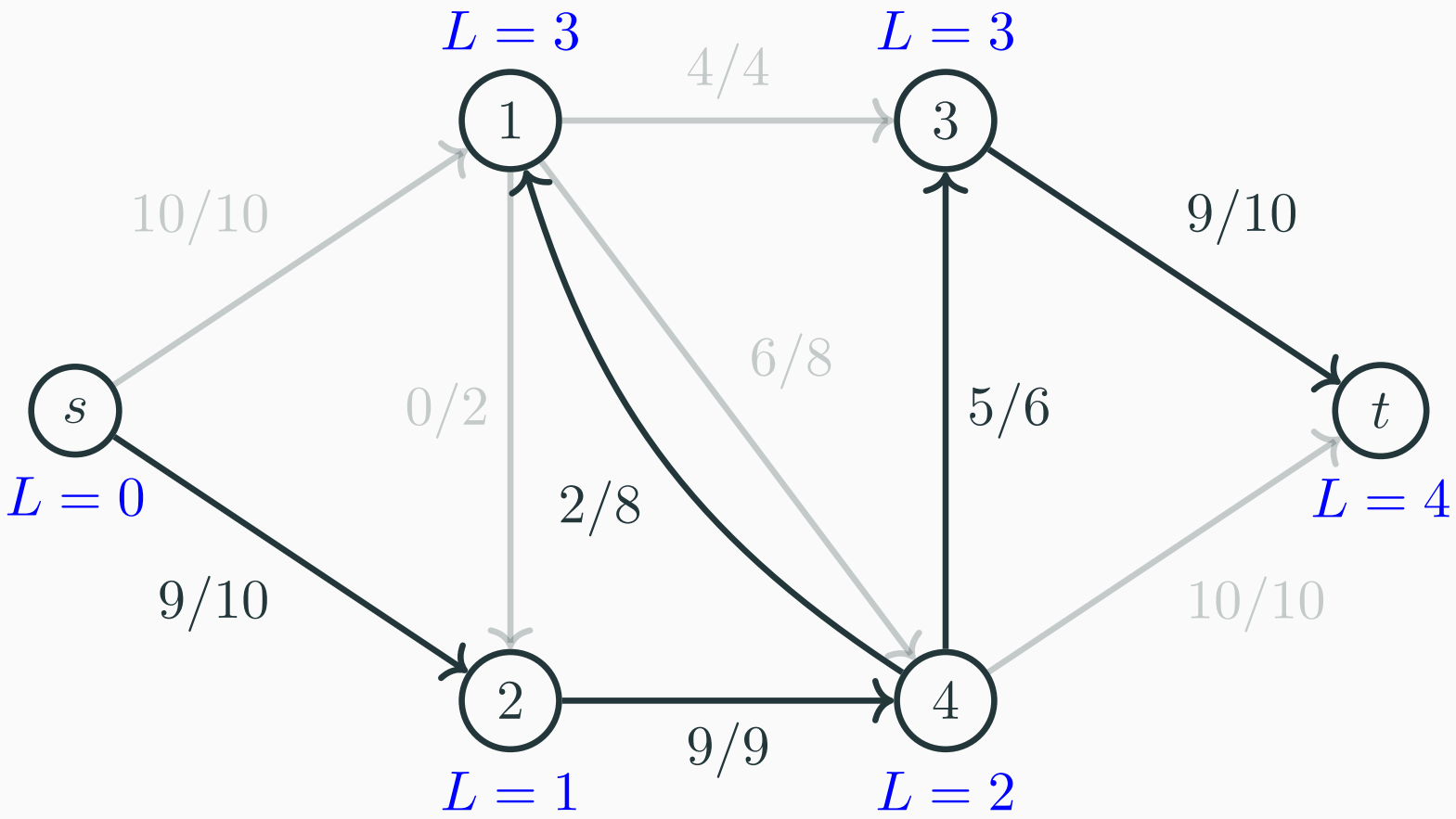
Example



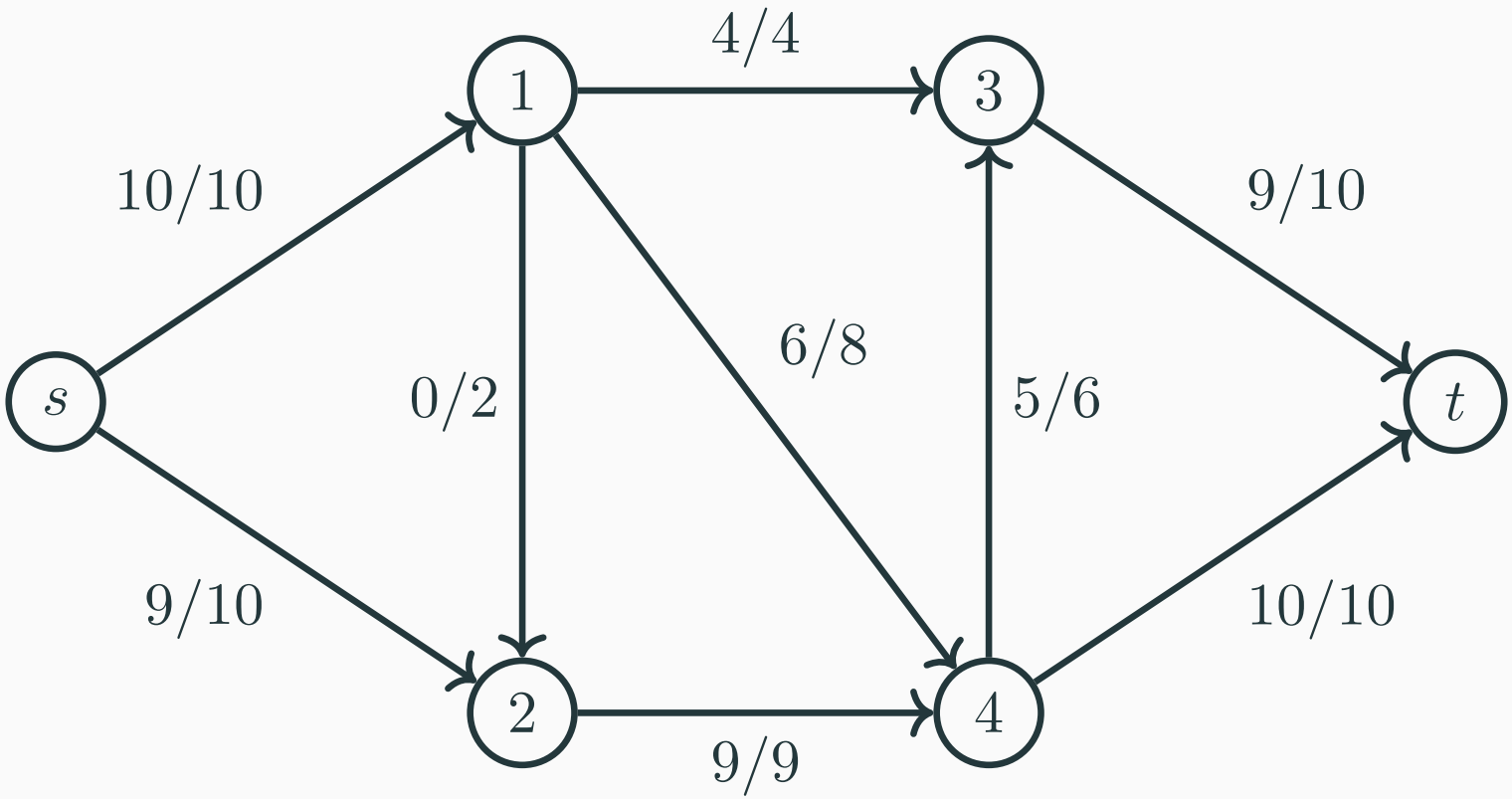
Example



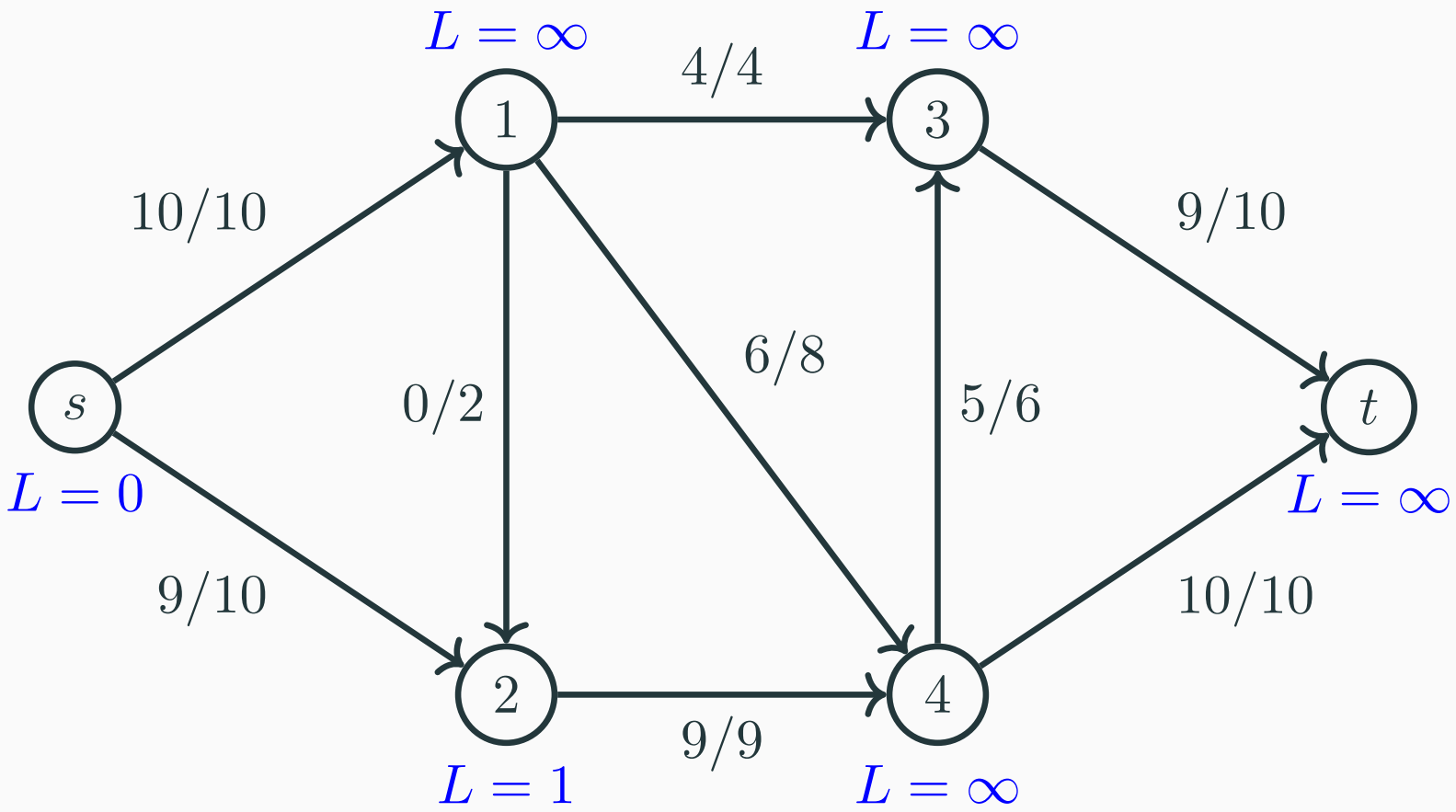
Example



Example



Example


$L = \infty$
$L = \infty$
4/4
1
3
10/10
9/10
6/8
0/2
5/6
s
t
$L = 0$
$L = \infty$
9/10
10/10
2
4
9/9
$L = 1$
$L = \infty$

Dinic's algorithm

- Why do this?
- Well, now in the layered network every edge out of a vertex v is "toward the sink", so pushing along any one of them is good.
- So when pushing flow out of v we just push to the first non-saturated edge.
- Each vertex is given a counter which points to the first non-saturated edge, so we just push to that edge until it's full, then increment if it's full.

Dinic's algorithm

- How long does making a blocking flow take then?
- Each DFS is $\mathcal{O}(n + V))$ where n is the number of increments.
- Each DFS saturates one edge, so we do at most E of them. And we can only increment each vertex E times so the sum of all the n is at most VE.
- Thus the total time is $\mathcal{O}(VE)$.
- Then one can prove this has to be done at most V times as $L(t)$ increases each time (slightly technical, we omit it here)

Dinic's algorithm

- Thus the total time complexity is $\mathcal{O}(V^2E)$, an improvement over $\mathcal{O}(VE^2)$.
- But I promised $\mathcal{O}(E\sqrt{V})$?
- Well, if all capacities are 1 then each layered network will take $\mathcal{O}(E)$ since each edge is only considered once.
- One can also prove that if all capacities are 1 there are at most $\sqrt{E}$ phases.
- If each vertex has either only 1 in-edge or only 1-out edge, then this can be improved to at most $\sqrt{V}$ phases.

Implementation - part 1

```
struct FlowEdge {
        int v, u;
        long long cap, flow = 0;
        FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
        const long long flow_inf = 1e18;
        vector<FlowEdge> edges;
        vector<vector<int>> adj;
        int n, m = 0;
        int s, t;
        vector<int> level, ptr;
        queue<int> q;

        Dinic(int n, int s, int t) : n(n), s(s), t(t) {
                adj.resize(n);
                level.resize(n);
                ptr.resize(n);
        }
```

Implementation - part 2

```
void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
}
bool bfs() {
        while (!q.empty()) {
                int v = q.front();
                q.pop();
                for (int id : adj[v]) {
                        if (edges[id].cap == edges[id].flow)
                            continue;
                        if (level[edges[id].u] != -1)
                            continue;
                        level[edges[id].u] = level[v] + 1;
                        q.push(edges[id].u);
                }
        }
        return level[t] != -1;
}
```

Implementation - part 3

```
long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
                int id = adj[v][cid];
                int u = edges[id].u;
                if (level[v] + 1 != level[u])
                    continue;
                long long tr = dfs(u,
                    min(pushed, edges[id].cap - edges[id].flow));
                if (tr == 0)
                    continue;
                edges[id].flow += tr;
                edges[id ^ 1].flow -= tr;
                return tr;
        }
        return 0;
}
```

Implementation - part 4

```
        long long flow() {
                long long f = 0;
                while (true) {
                        fill(level.begin(), level.end(), -1);
                        level[s] = 0;
                        q.push(s);
                        if (!bfs())
                            break;
                        fill(ptr.begin(), ptr.end(), 0);
                        while (long long pushed = dfs(s, flow_inf)) {
                                f += pushed;
                        }
                }
                return f;
        }
};
```

MCMF

- Last algorithm of the day.
- What if not all flow is created equal?
- It costs much more to send something from A to B than from A to C perhaps
- How could we modify Ford-Fulkerson to do this?

MCMF

- Well we could simply take the shortest (cost) path each time instead of the shortest (edges) path each time.
- This simply works! (Try to reason why)
- If all the costs are ≥ 0 we can use Dijkstra, otherwise Bellman-Ford is needed (or Johnson's, since negative cycles make the problem undefined anyway)
- Only thing to be careful about is to make the reverse edge have cost $-c$ if the original edge has cost c, since we want things to cancel

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