

# Minimum Cut, Bipartite Matching and Minimum Cost Maximum Flow with BGL

---

Tim Taubner<sup>1</sup>

November 13, 2019

ETH Zürich,

<sup>1</sup> based on material from Daniel Graf and Andreas Bärtschi

# Recap: Basic Network Flows – What did we see last time?

One problem to rule them all...

### Network Flow: Problem Statement

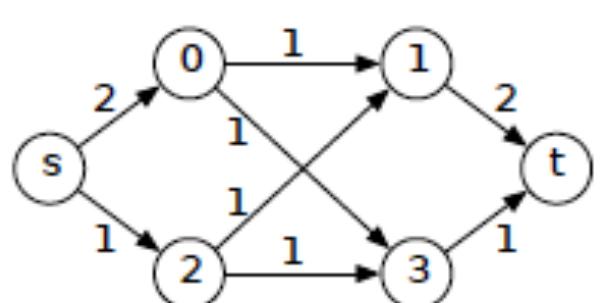
**Input:** A flow network consisting of

- directed graph  $G = (V, E)$
- source  $s \in V$  with only out edges
- sink  $t \in V$  with only in edges
- edge capacity  $c : E \rightarrow \mathbb{N}$ .

**Output:** Flow function  $f : E \rightarrow \mathbb{R}$  subject to

- Capacity constraints for every edge  $e$ .  
 $0 \leq f(e) \leq c(e)$
- Flow conservation at every vertex  $u$ .  
 $\sum_{(v,u) \in E} f((v,u)) \text{ in flow} = \sum_{(u,w) \in E} f((u,w)) \text{ out flow}$
- Maximization of total flow.  
 $|f| := \sum_{(s,v) \in E} f((s,v))$

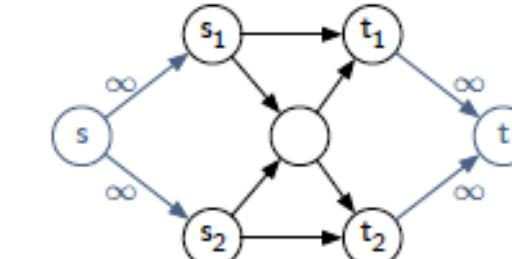
**Theorem**  
On a graph with integral capacities, there always exists an integral maximum flow.



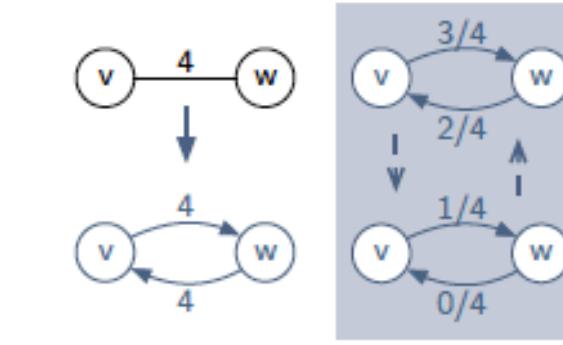
4 / 23

### Common techniques

**Multiple sources/sinks**  
with e.g.  $\infty \approx \sum_{e \in E} c(e)$



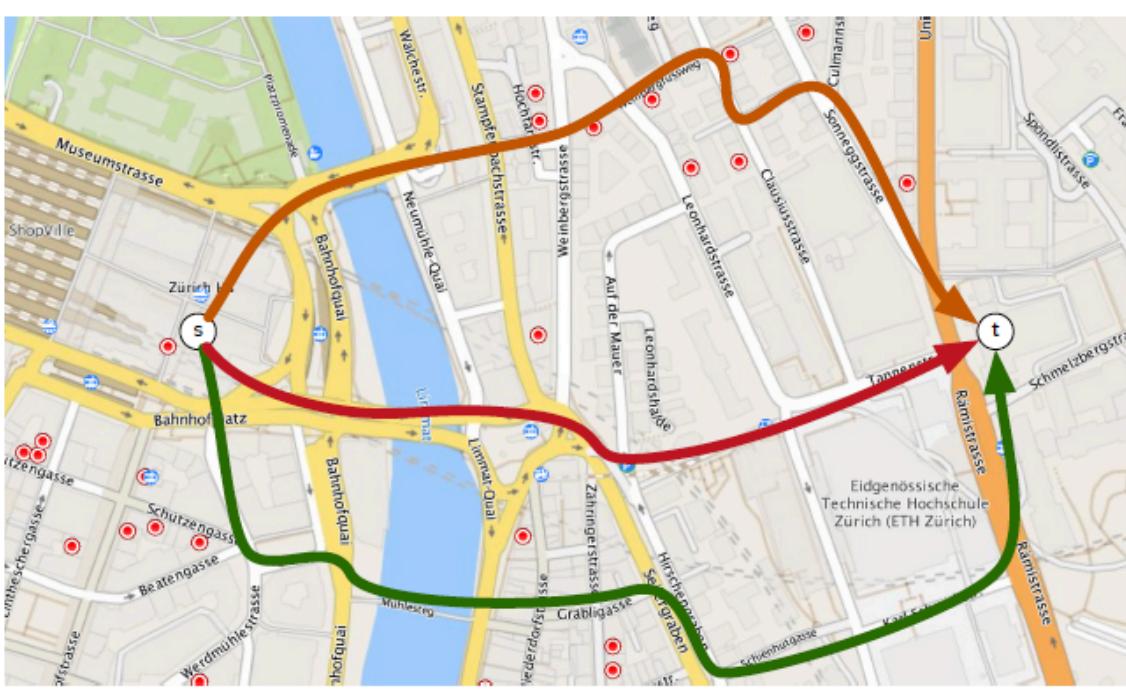
**Undirected graphs**  
antiparallel edges with flow reducible to one direction



12 / 23

### Flow Application: Edge Disjoint Paths

How many ways are there to get from HB to CAB without using the same street twice?



Map: search.ch, TomTom, swisstopo, OSM

13 / 23

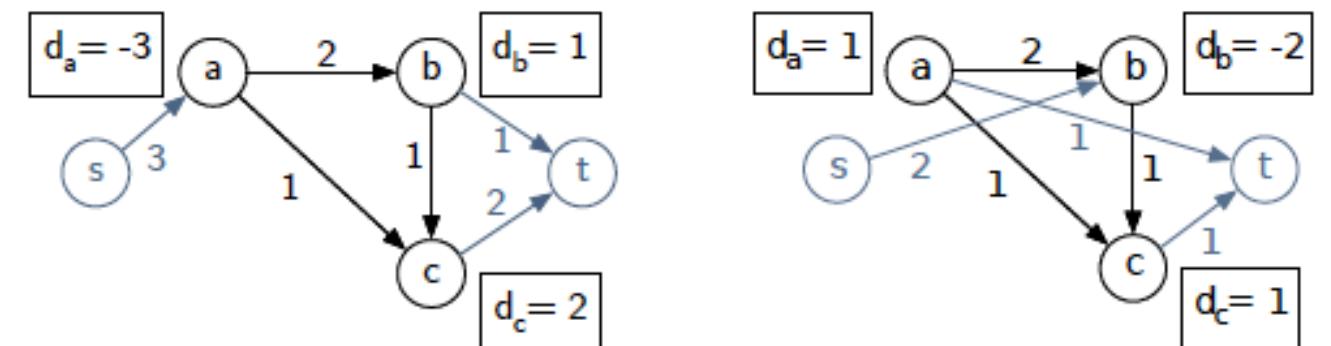
### Flow Application: Circulation Problem

► Multiple sources with a certain amount of flow to give (supply).

► Multiple sinks that want a certain amount of flow (demand).

► Model these as negative or positive demand per vertex  $d_v$ .

► Question: Is there a feasible flow? Surely not if  $\sum_{v \in V} d_v \neq 0$ . Otherwise? Add super-source and super-sink to get a maximum flow problem.



feasible flow exists

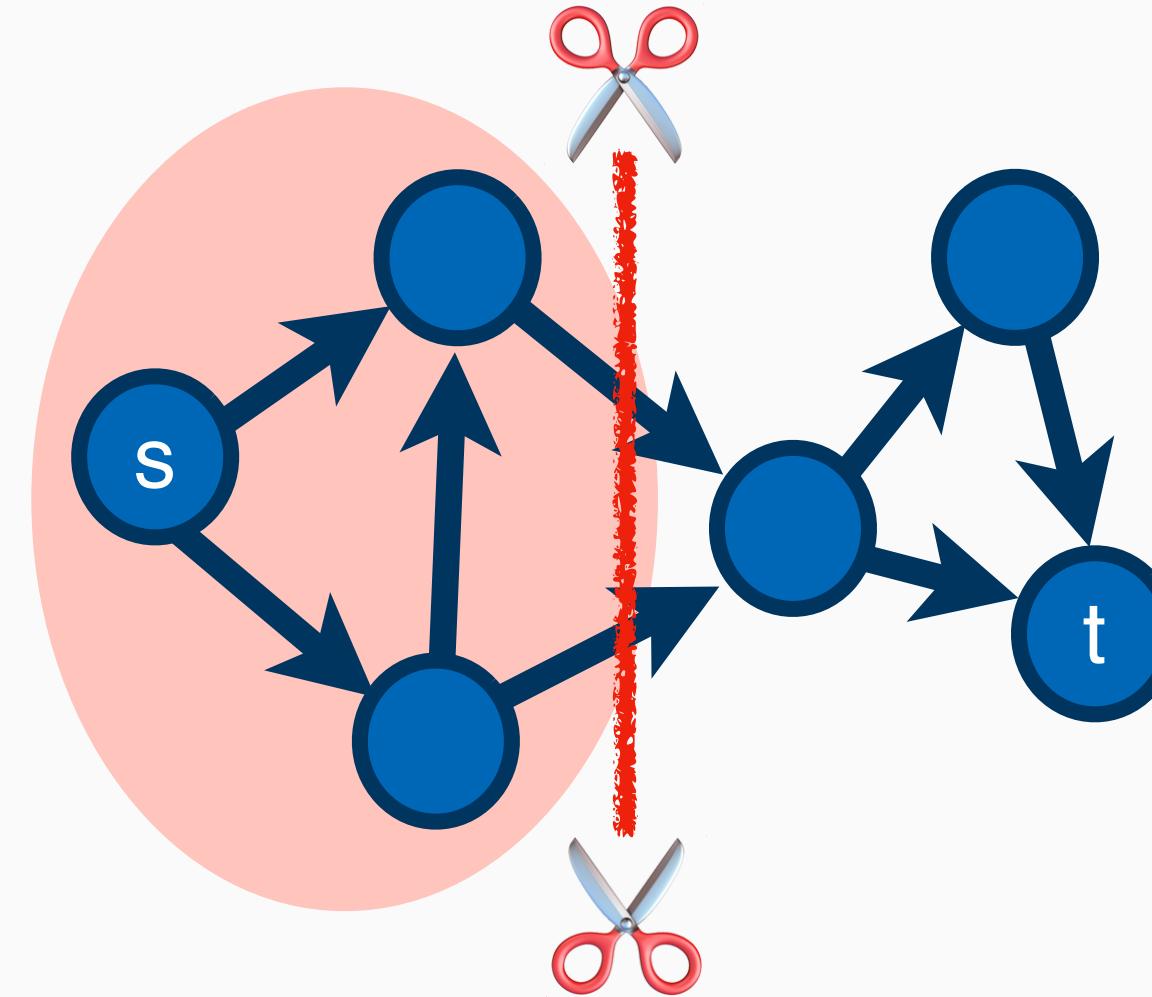
no feasible flow exists

18 / 23

# Today: Advanced Network Flow – What else are flows useful for?

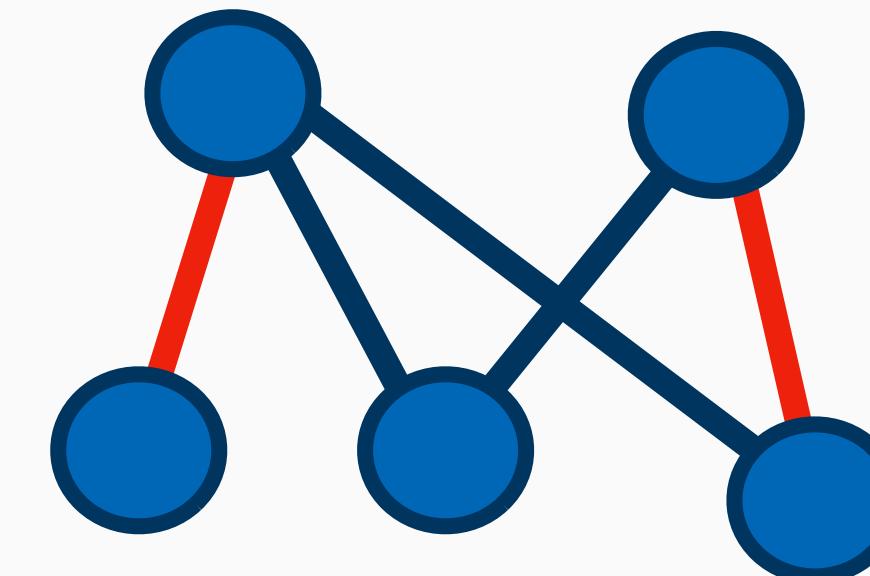
## Minimum Cuts

- ▶ How to disconnect  $t$  from  $s$  cheaply?



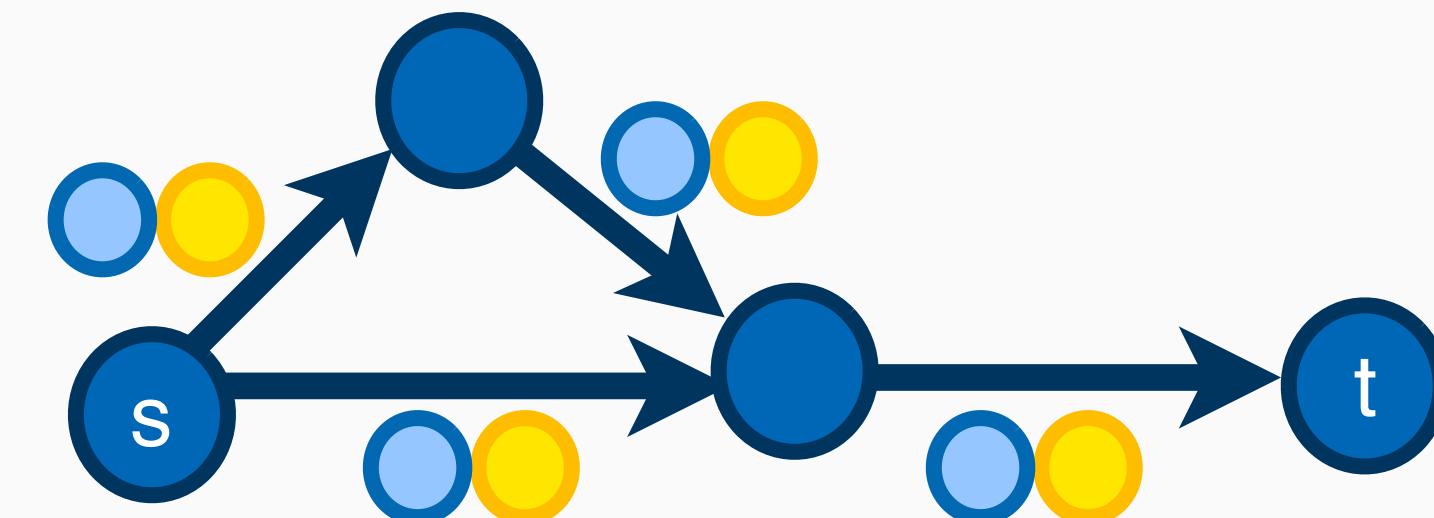
## Bipartite Matching

- ▶ How to assign A's to B's effectively?



## Flows with Costs

- ▶ What if sending flow comes with a price?



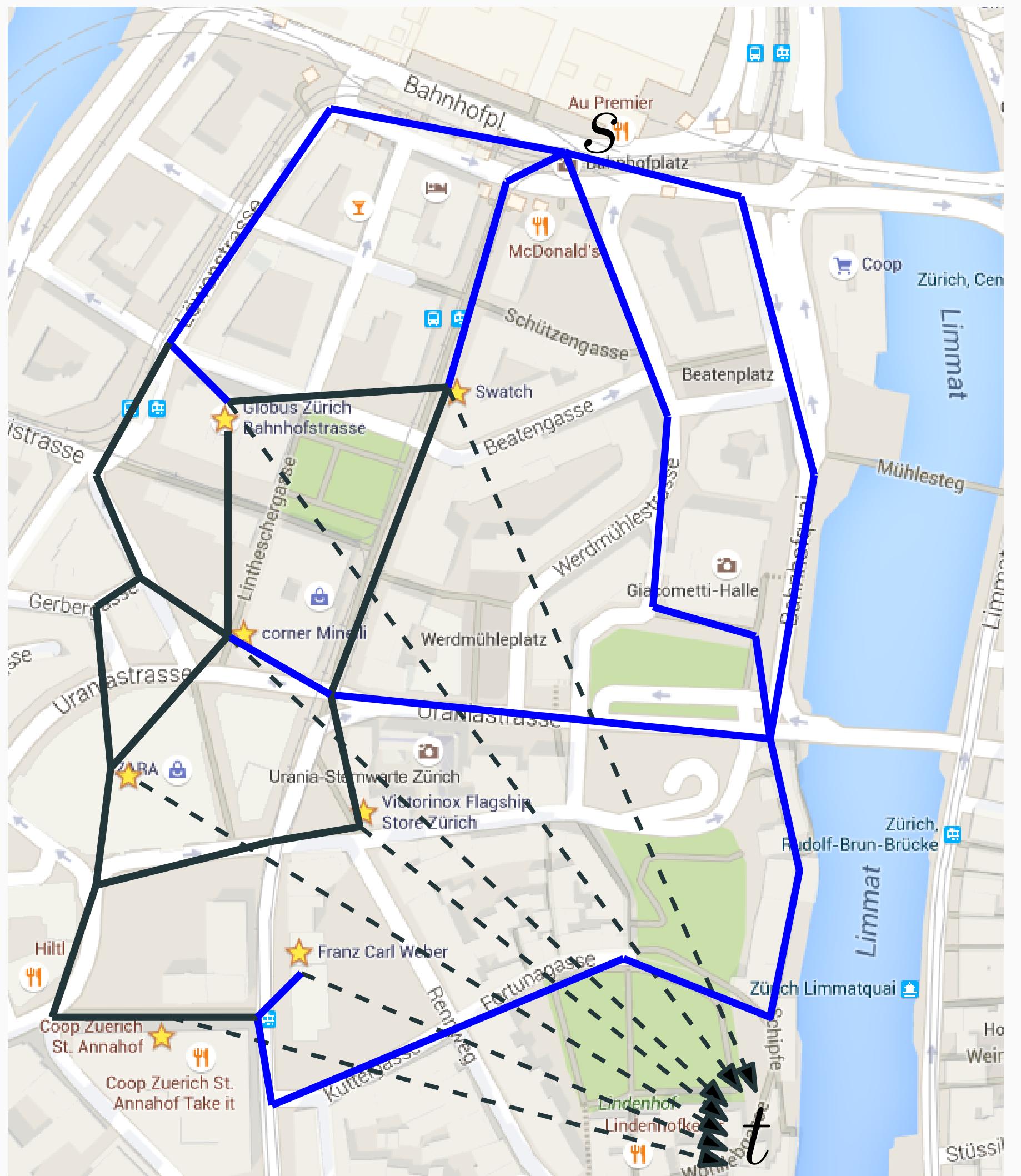
# Minimum Cut

---

# Minimum Cut: Shopping Trip



# Minimum Cut: Shopping Trip



Start from HB:

- ▶ Visit as many shops as possible.
- ▶ Return to HB after each shop.

Condition: Use each road on at most one trip.

Compute the bottleneck, i.e. the number of edge-disjoint paths.  $\Rightarrow$  Four shops.

Unrealistic condition!

(There are interesting streets in Zürich.)

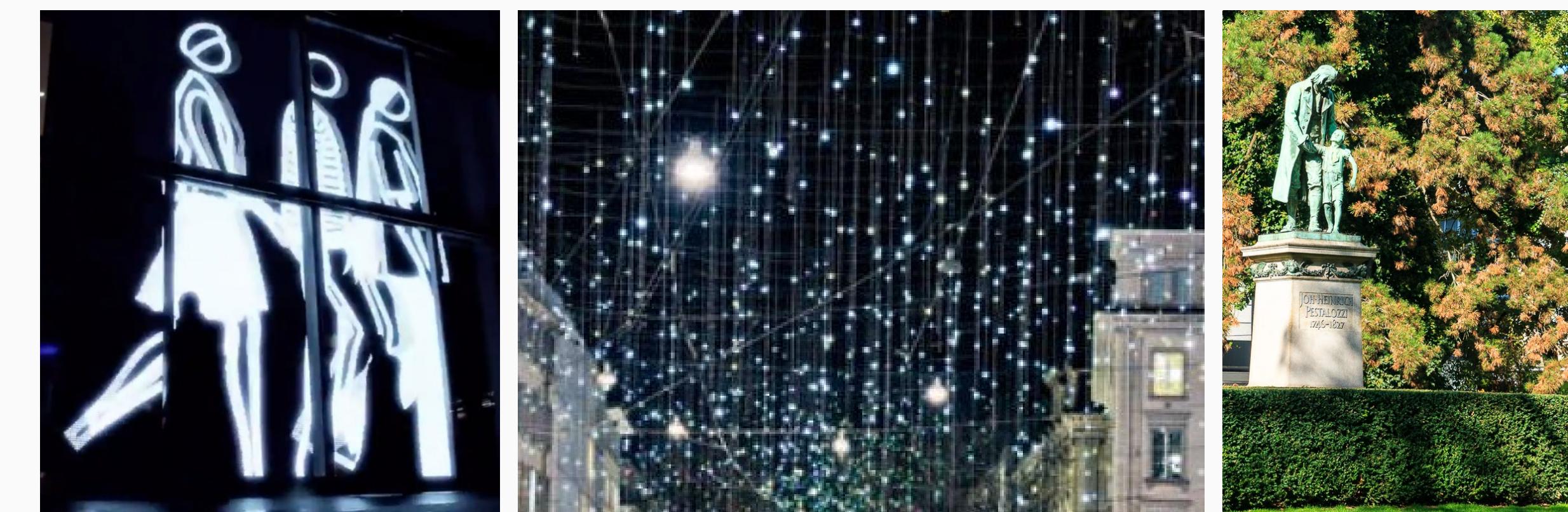
# Minimum Cut: Shopping Trip



Start from HB:

- ▶ Visit as many shops as possible.
- ▶ Return to HB after each shop.

Condition: Use beautiful roads more often.



Use Bahnhofstrasse up to three times.

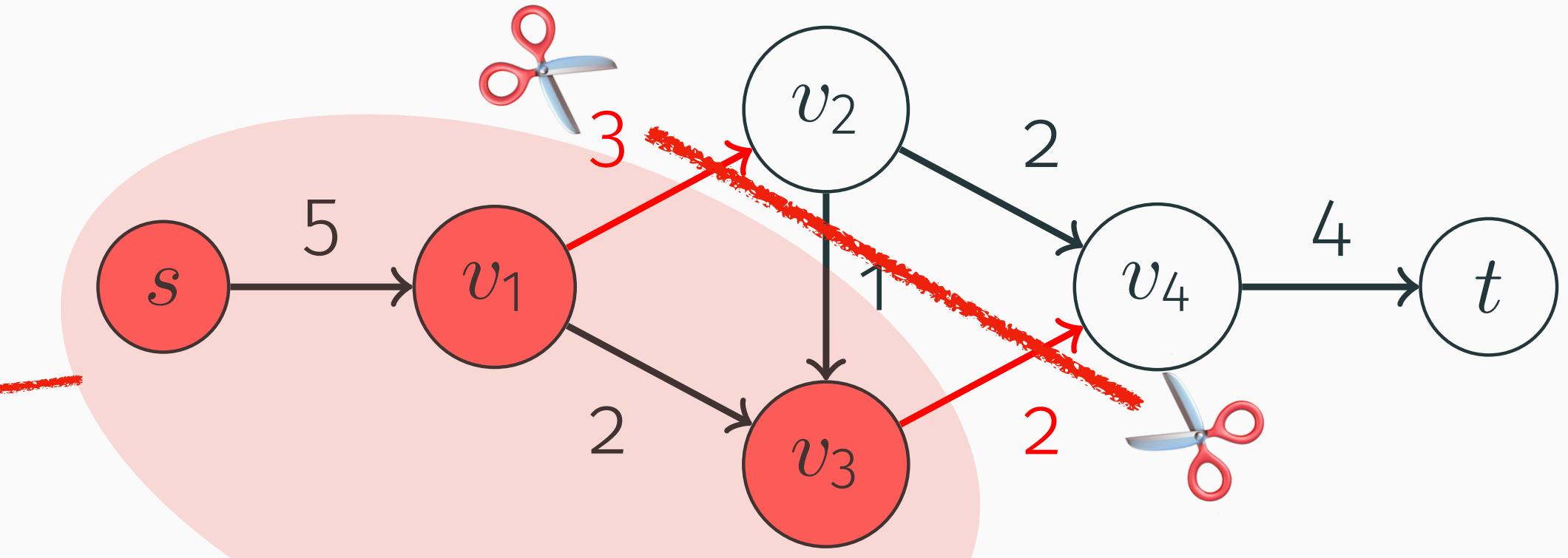
Compute the weighted bottleneck, i.e. the minimum cut between  $s$  and  $t$ .  $\Rightarrow 6$  shops.

# Minimum Cut: Cuts and Flows

$G = (V, E, s, t)$  a flow network.  $S \subset V$  s.t.  $s \in S, t \in V \setminus S$ , e.g.  $S = \{s, v_1, v_3\}$ .

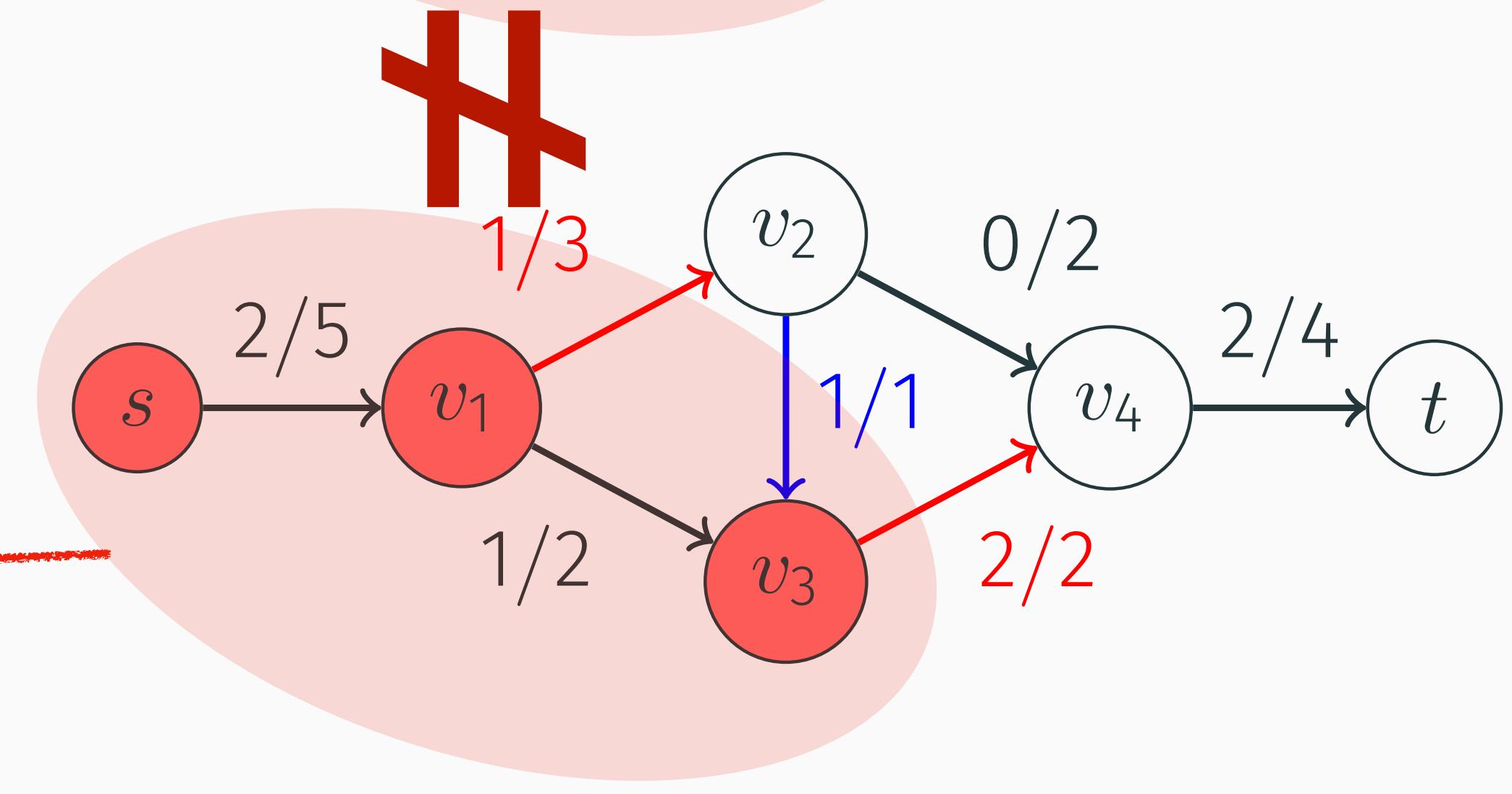
The value of the  $(S, V \setminus S)$ -cut is

$$\begin{aligned} \text{cap}(S, V \setminus S) &:= \text{outgoing capacity} \\ &= \sum_{\substack{e=(u,v) \\ u \in S, v \in V \setminus S}} \text{cap}(e) \\ &= 3 + 2 = 5. \end{aligned}$$



The value of the flow  $f$  from  $S$  to  $V \setminus S$  is

$$\begin{aligned} f(S, V \setminus S) &:= \text{outgoing flow} - \text{incoming flow} \\ &= \sum_{\substack{e=(u,v) \\ u \in S, v \in V \setminus S}} \text{flow}(e) - \sum_{\substack{e=(v,u) \\ u \in S, v \in V \setminus S}} \text{flow}(e) \\ &= 1 + 2 - 1 = 2. \end{aligned}$$



# Minimum Cut: Maxflow-Mincut-Theorem

## Theorem (Maxflow-Mincut-Theorem)

Let  $f$  be an  $s$ - $t$ -flow in a graph  $G$ . Then  $f$  is a maximum flow if and only if

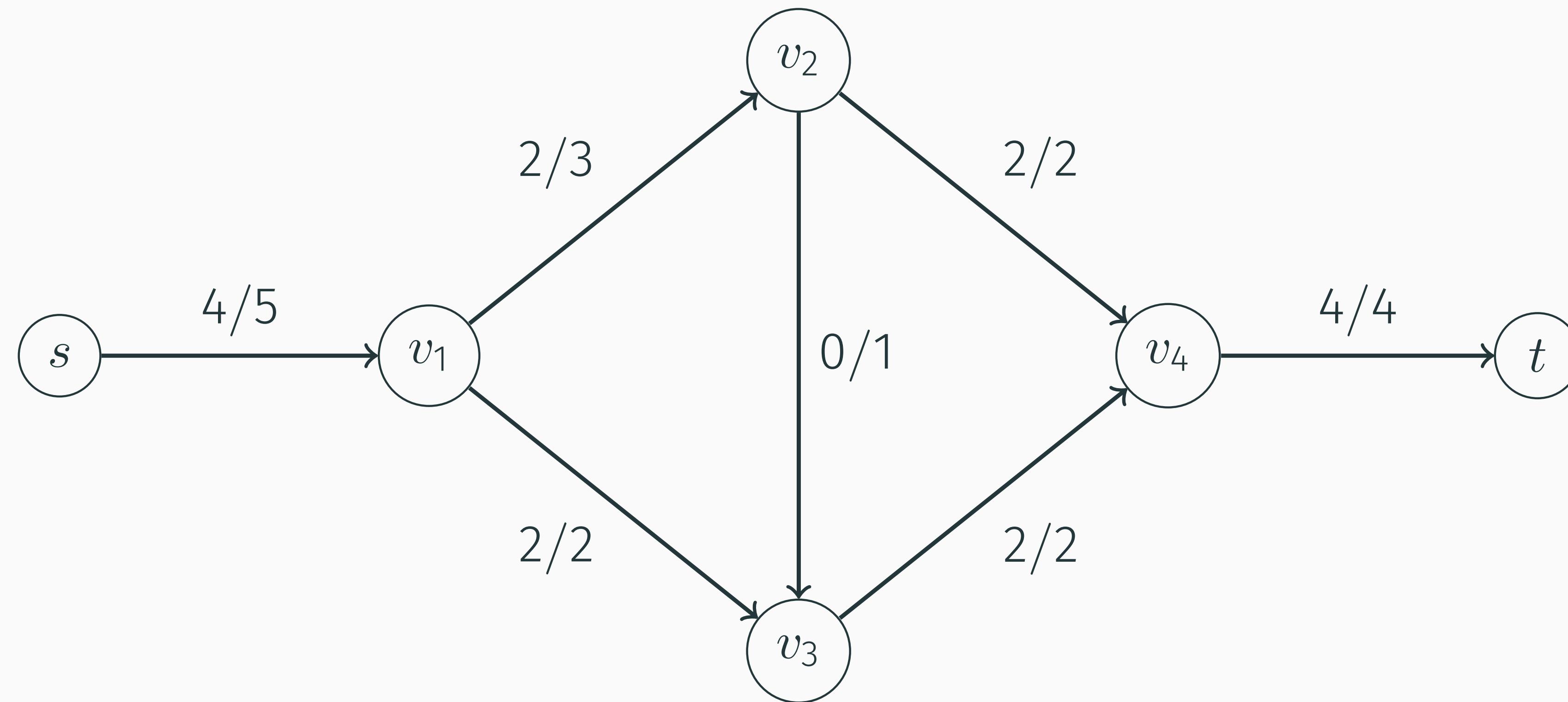
$$|f| = \min_{S: s \in S, t \notin S} \text{cap}(S, V \setminus S).$$

This allows us to easily find a minimum  $s$ - $t$ -cut:

- ▶ Construct the residual graph  $G_f := (V, E_f)$ . For each edge  $(u, v) \in G$  we have:
  - An edge  $(u, v) \in G_f$  with capacity  $\text{cap}(e) - f(e)$ , if  $\text{cap}(e) - f(e) > 0$ .
  - An edge  $(v, u) \in G_f$  with capacity  $f(e)$ , if  $f(e) > 0$ .
- ▶ Since  $f$  is a maximum flow, there is no  $s$ - $t$  path in the residual graph  $G_f$ .
- ▶ Take  $S$  to be all vertices in  $G_f$  reachable from  $s$ .  
 $\Rightarrow (S, V \setminus S)$  is a minimum  $s$ - $t$ -cut.

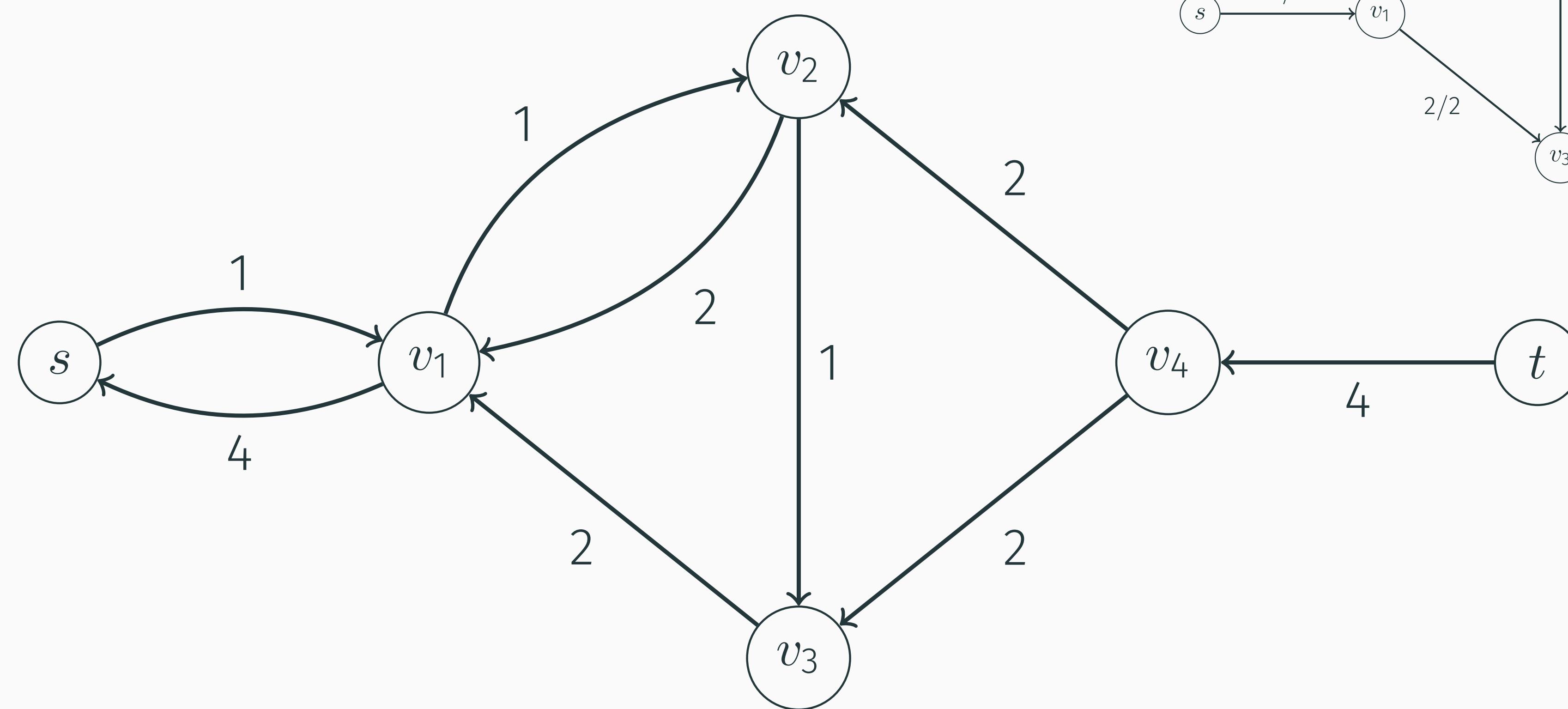
# Minimum Cut: Example

Graph  $G$  and a maximum flow  $f$ .

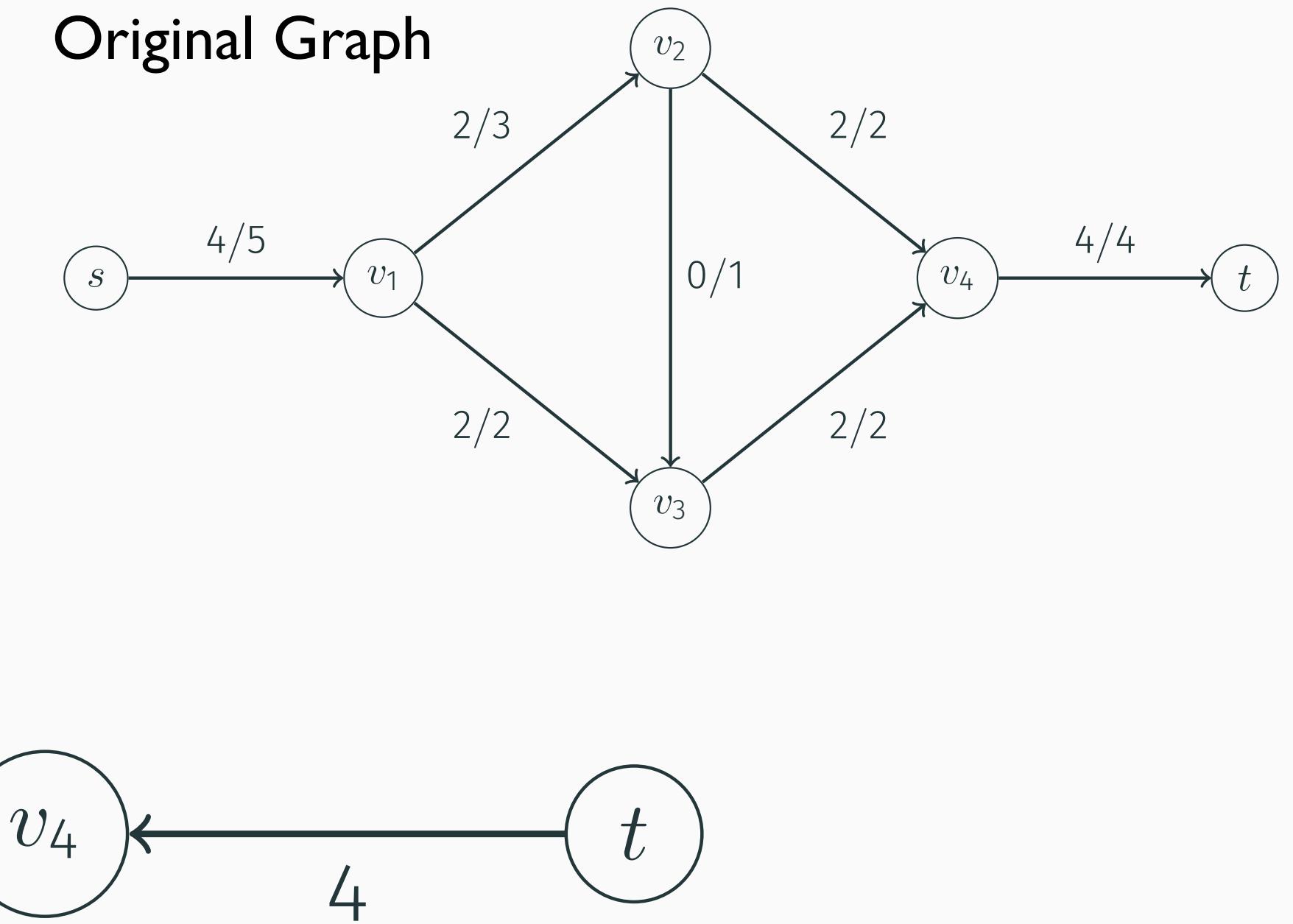


# Minimum Cut: Example

Residual graph  $G_f$ .

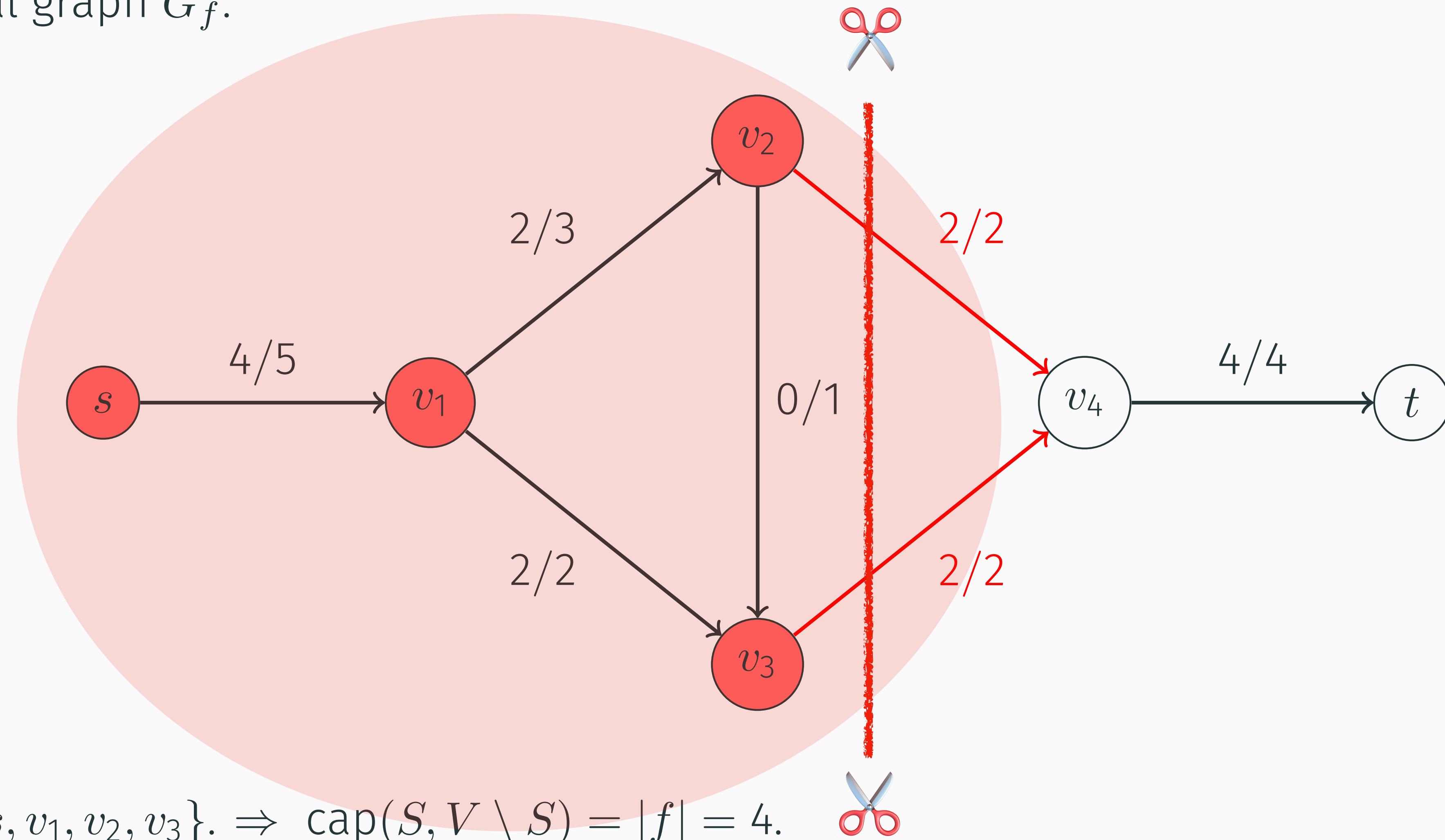


Original Graph



# Minimum Cut: Example

Residual graph  $G_f$ .



## Minimum Cut: Code

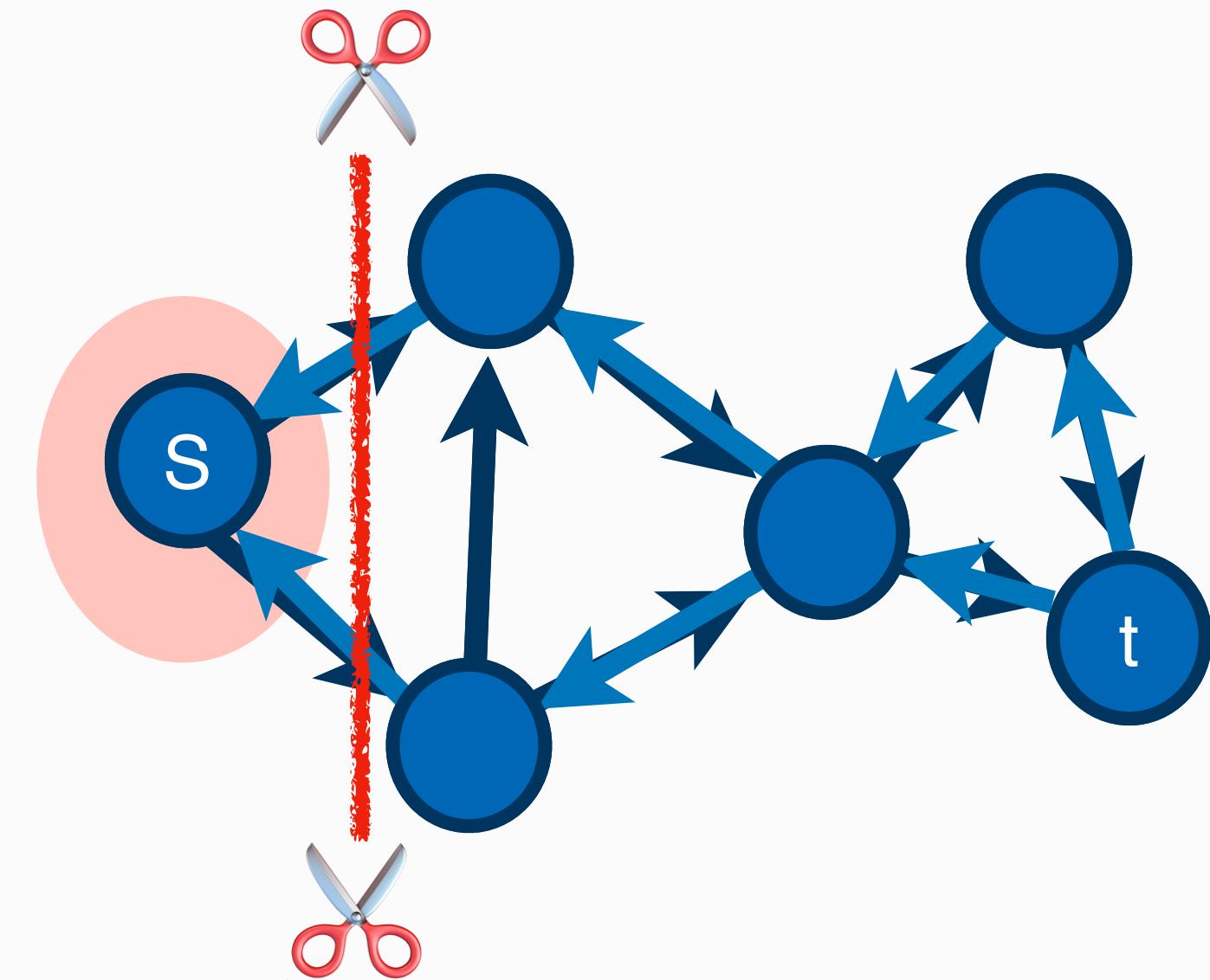
Example code: BFS on the residual graph  $G_f$ . → [bgl\\_residual\\_bfs.cpp](#)

```
90 // BFS to find vertex set S
91 std::vector<int> vis(N, false); // visited flags
92 std::queue<int> Q; // BFS queue (from std:: not boost::)
93 vis[src] = true; // Mark the source as visited
94 Q.push(src);
95 while (!Q.empty()) {
96     const int u = Q.front();
97     Q.pop();
98     out_edge_it ebeg, eend;
99     for (boost::tie(ebeg, eend) = boost::out_edges(u, G); ebeg != eend; ++ebeg) {
100         const int v = boost::target(*ebeg, G);
101         // Only follow edges with spare capacity
102         if (rc_map[*ebeg] == 0 || vis[v]) continue;
103         vis[v] = true;
104         Q.push(v);
105     }
106 }
```

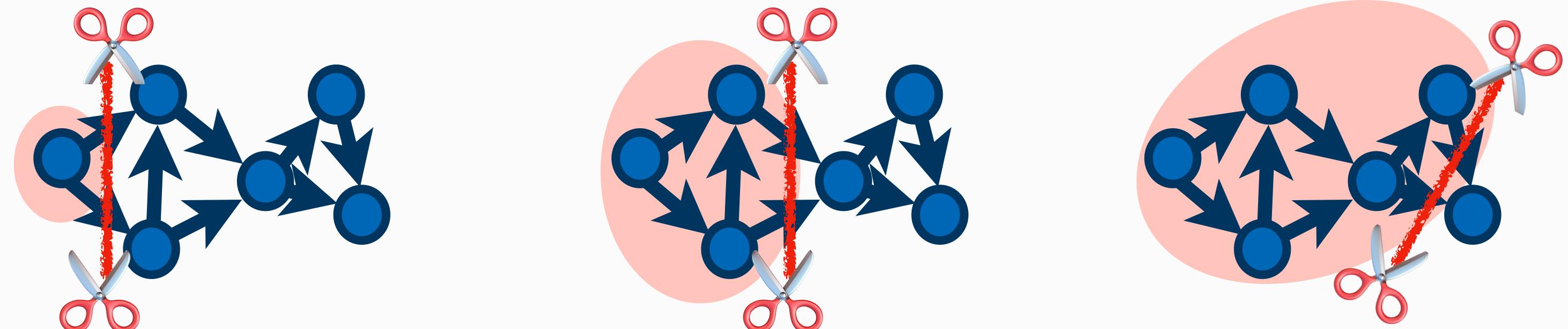
# Minimum Cut: Algorithm

Summary of what you need to do to find a minimum cut:

1. Compute maximum flow  $f$  and the residual graph  $G_f$ .
2. Compute the set of vertices  $S$ :
  - $S$  is reachable from the source  $s$  in  $G_f$ .
  - BFS on edges with residual capacity  $> 0$ .
3. Output (depending on the task):
  - All vertices in  $S$ .
  - All edges going from  $S$  to  $V \setminus S$ .



Note: Minimum cuts are not necessarily unique. But earliest and latest min-cuts are. Also computing the maximum cut is NP-hard.



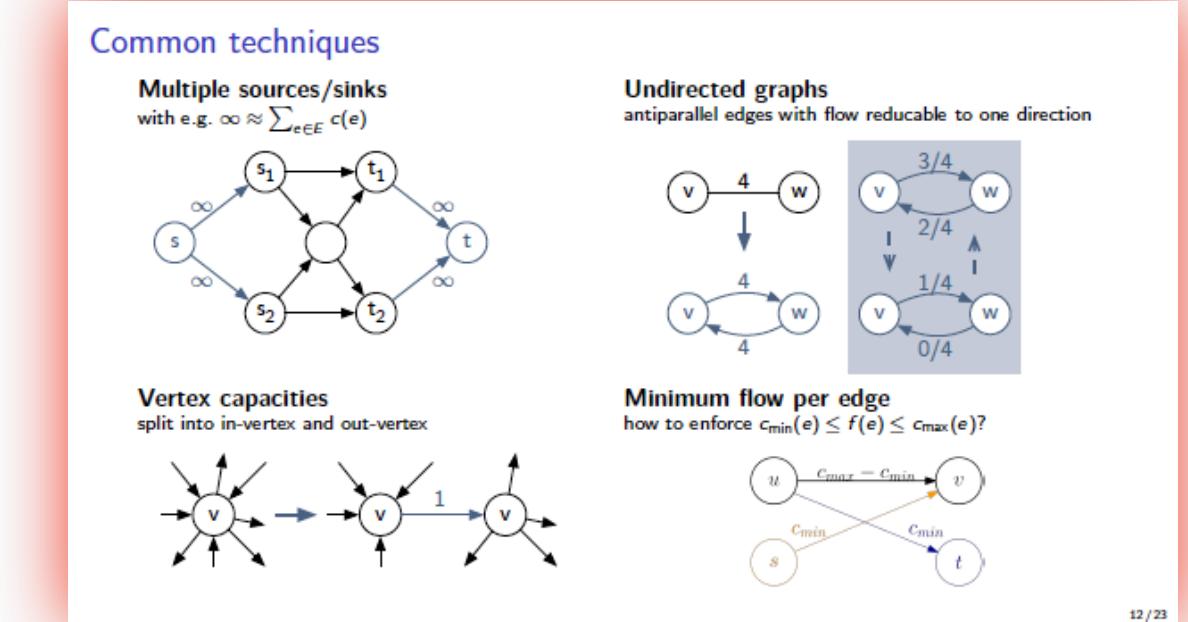
# Bipartite Matchings

---

# Maximum Matchings: Bipartite Graphs

Maximum Matching = pick as many non-adjacent edges as possible

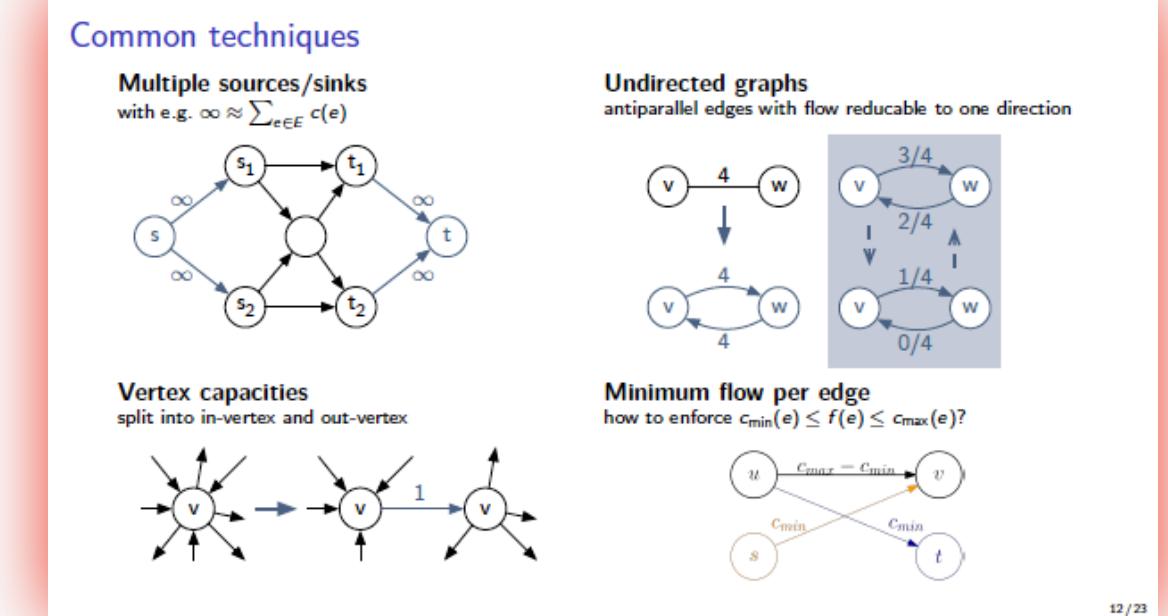
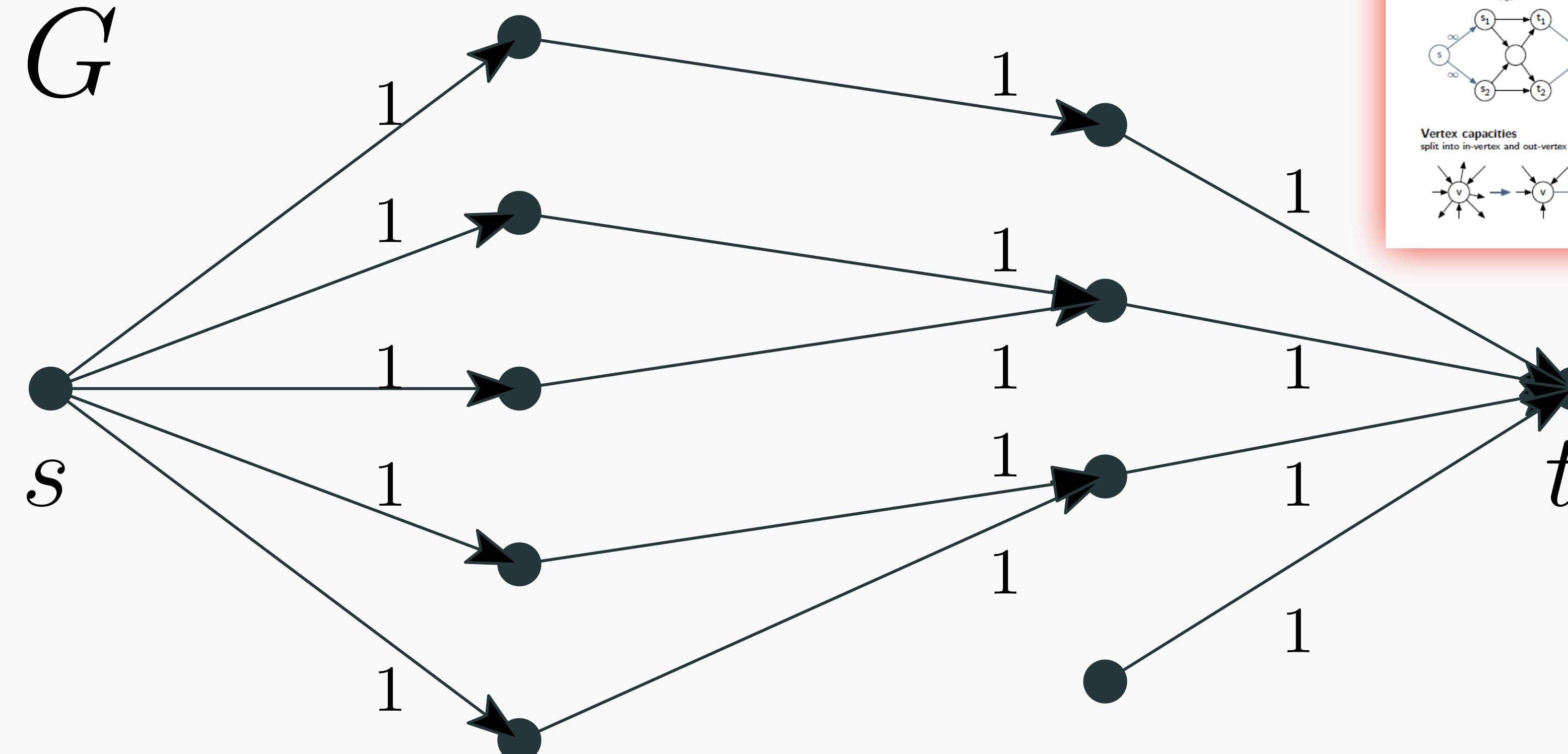
Flow formulation through circulation / vertex capacities / edges-disjoint paths:



# Maximum Matchings: Bipartite Graphs

Maximum Matching = pick as many non-adjacent edges as possible

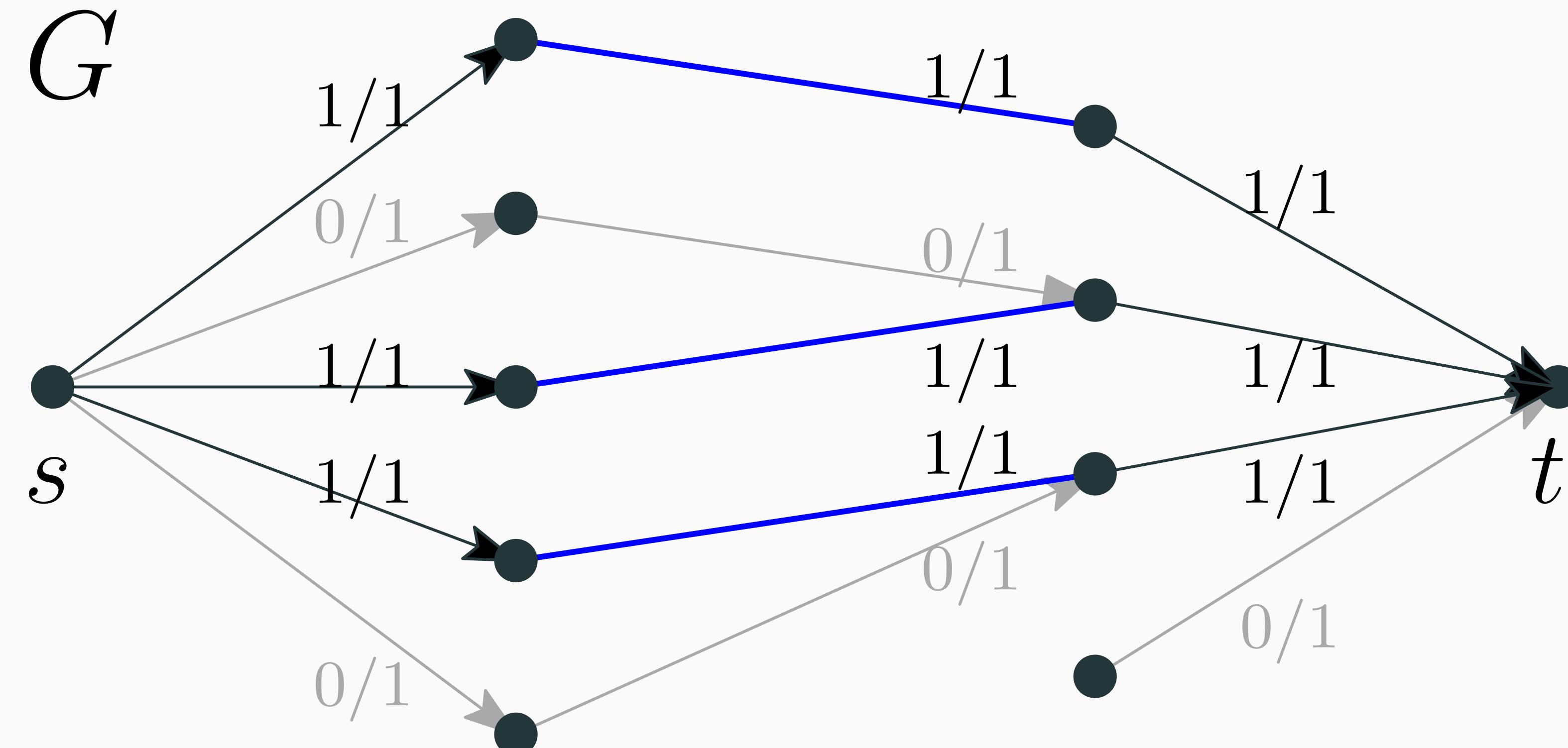
Flow formulation through circulation / vertex capacities / edges-disjoint paths:



# Maximum Matchings: Bipartite Graphs

Maximum Matching = pick as many non-adjacent edges as possible

Flow formulation through circulation / vertex capacities / edges-disjoint paths:



# Vertex Cover and Independent Set: General Graphs

- ▶ Maximum independent set (MaxIS)

Largest  $T \subseteq V$ , such that

$$\nexists u, v \in T : (u, v) \in E.$$

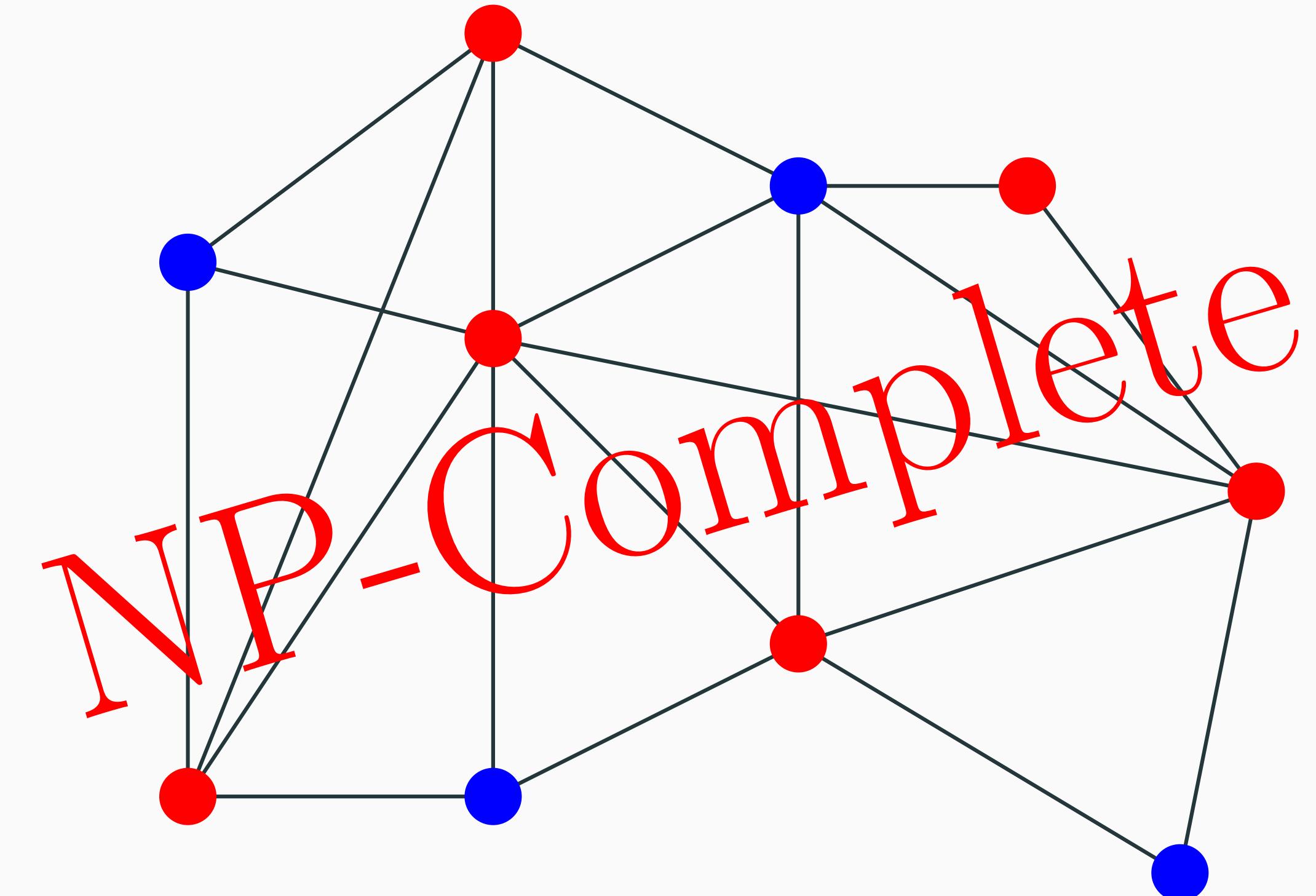
- ▶ Minimum vertex cover (MinVC)

Smallest  $S \subseteq V$ , such that

$$\forall (u, v) \in E : u \in S \vee v \in S.$$

- ▶ These problems are complementary!

$$\text{MaxIS} = V \setminus \text{MinVC}$$



# Brief Excursion: Options for Runtime Analysis

By now, we know many ways of deciding whether an algorithm is fast or slow:

- ▶ look at the input size (the classical way)
- ▶ look at the output size (e.g. fast as the answer is guaranteed to be small)  
If you know that MaxIS is very small, then it might be tractable while computing a MinVC directly is too slow
- ▶ look at some special input restrictions (e.g. Attack of the clones)
- ▶ look at detailed structure of the input (e.g. all graphs are trees)

# Vertex Cover and Independent Set: Bipartite Graphs

Theorem (König: MinVC and MaxIS is simpler on bipartite graphs!)

In a bipartite graph, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover.

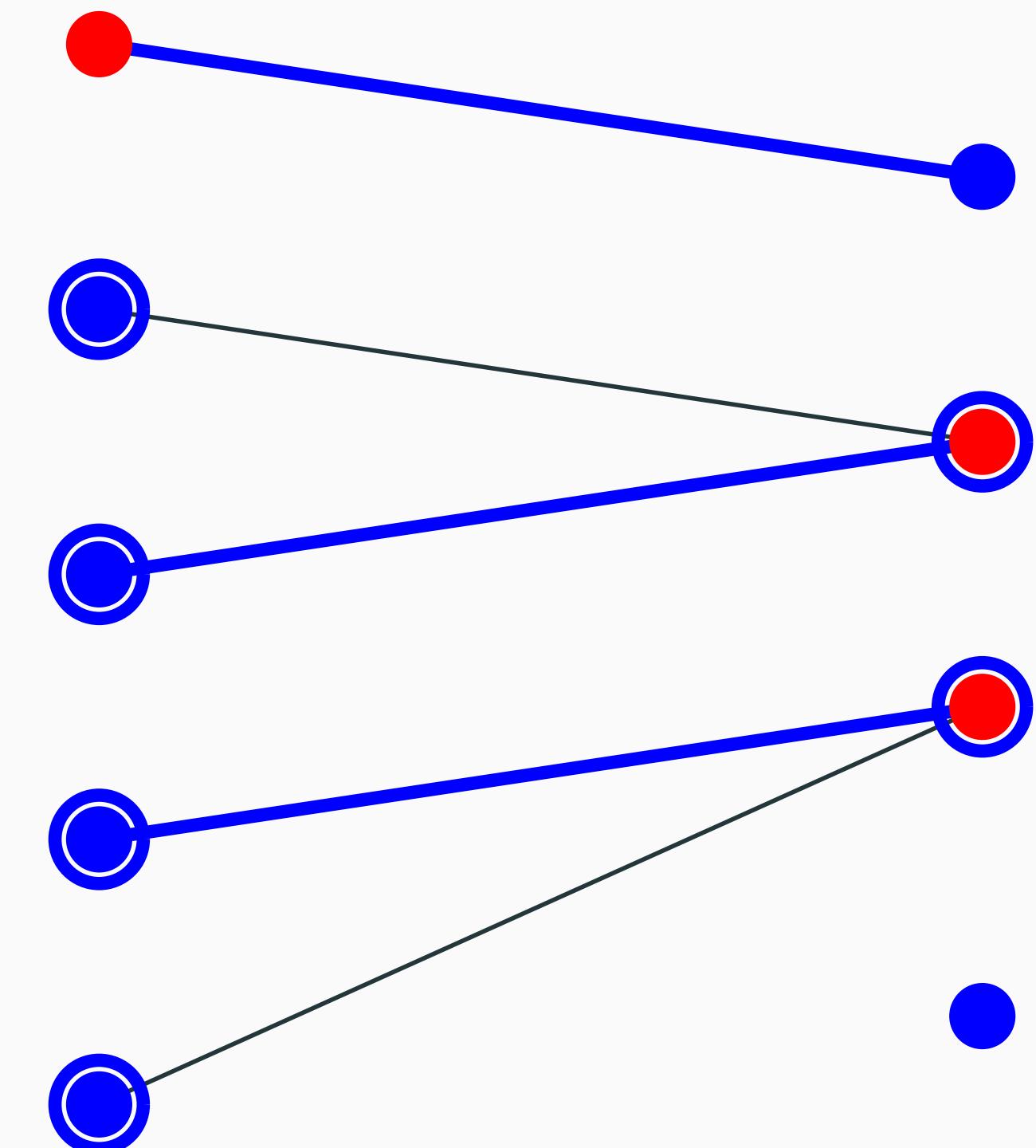
Proof: See [Wikipedia](#) for a nice and short proof.

Algorithm:

1. Maximum matching  $M$ ,  $V = L \cup R$ . Find all unmatched vertices in  $L$ , label them as visited.
2. Starting at visited vertices search (BFS) left to right along edges from  $E \setminus M$  and right to left along edges from  $M$ . Label each found vertex as visited.
3. MinVC – all unvisited in  $L$  and all visited in  $R$ .  
MaxIS – all visited in  $L$  and all unvisited in  $R$ .

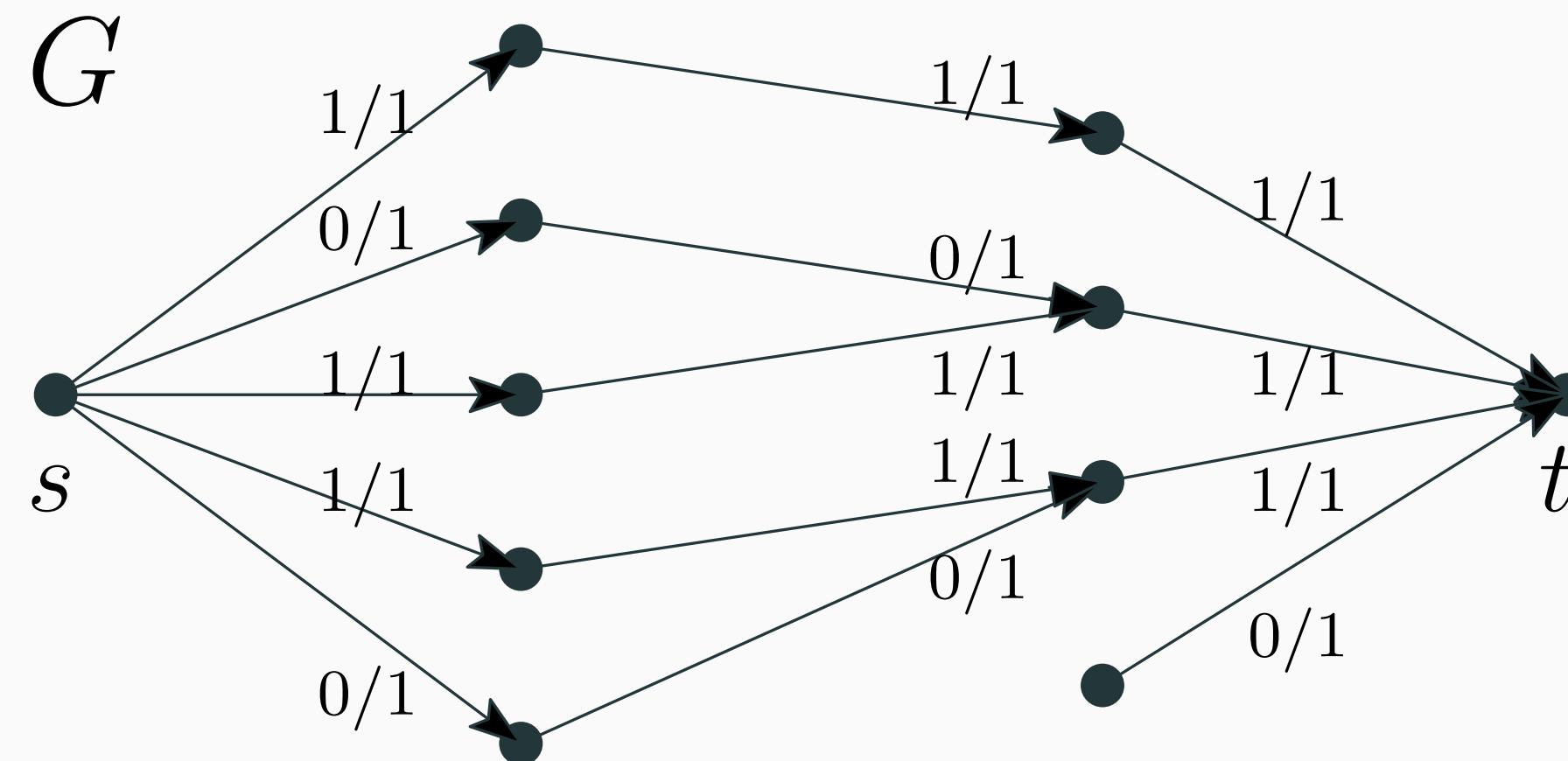
**Careful!** Step 2 can take several rounds.

Easy Implementation?

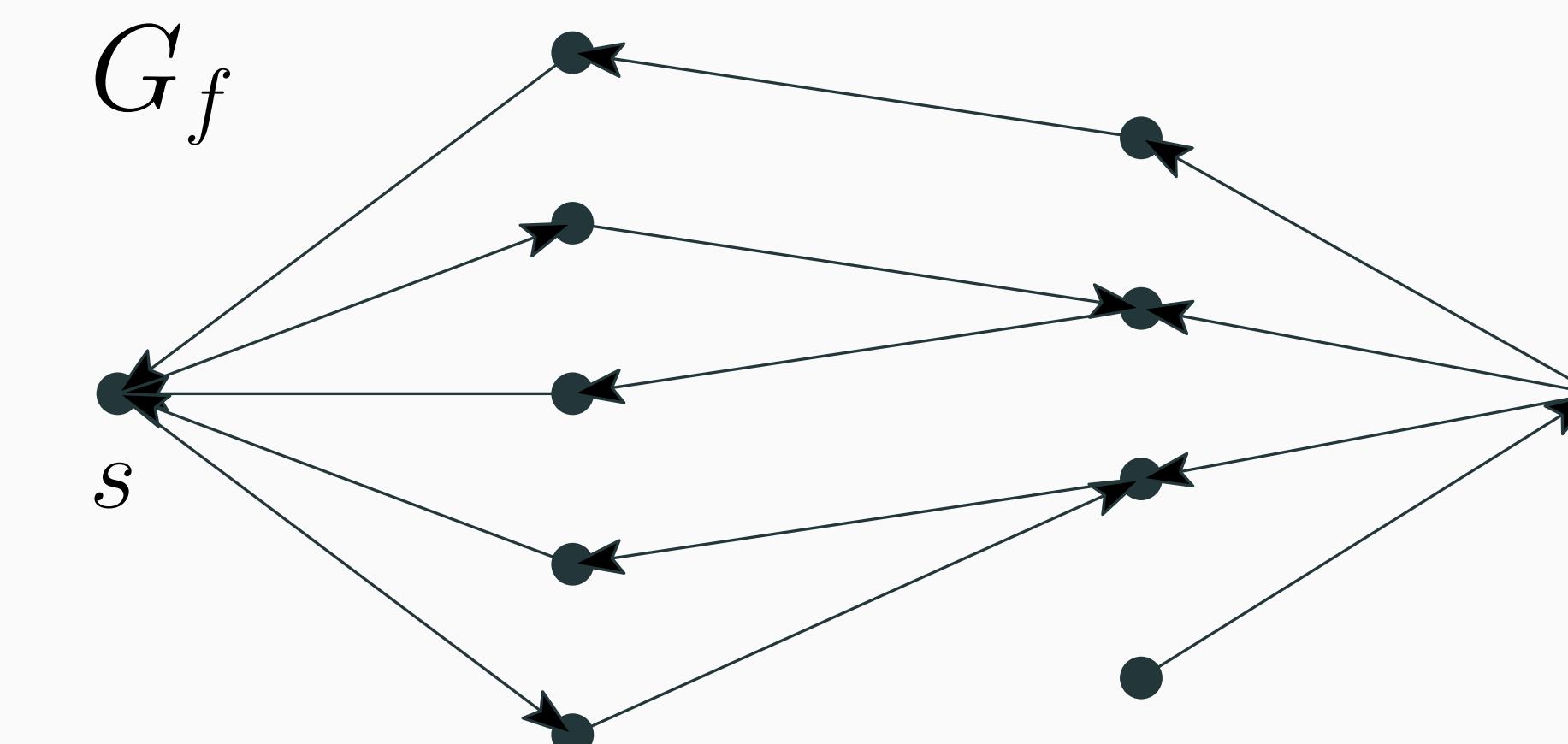


# Finding a MinVC or MaxIS in bipartite graphs: step by step

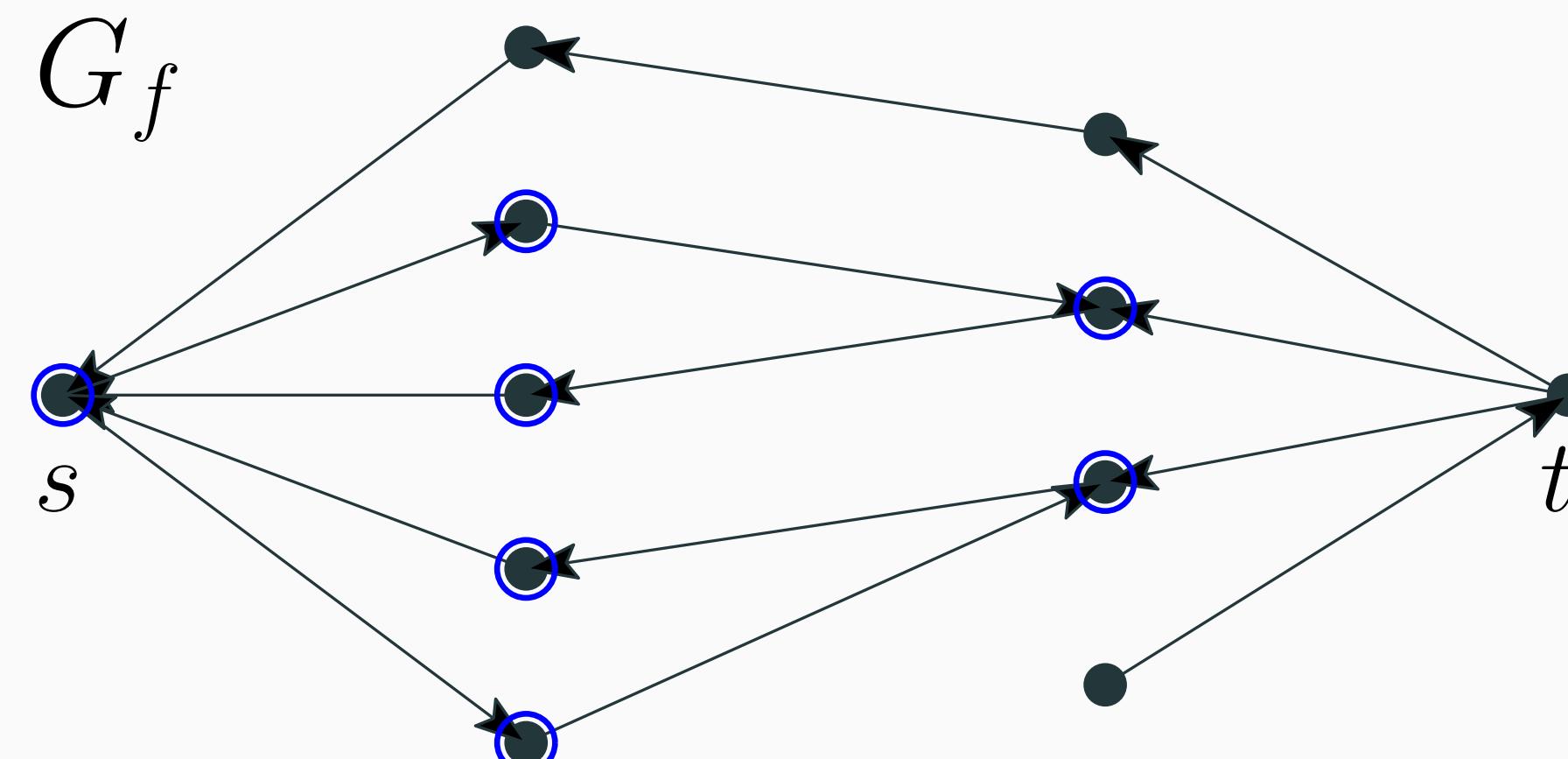
1) Formulate and compute the flow:



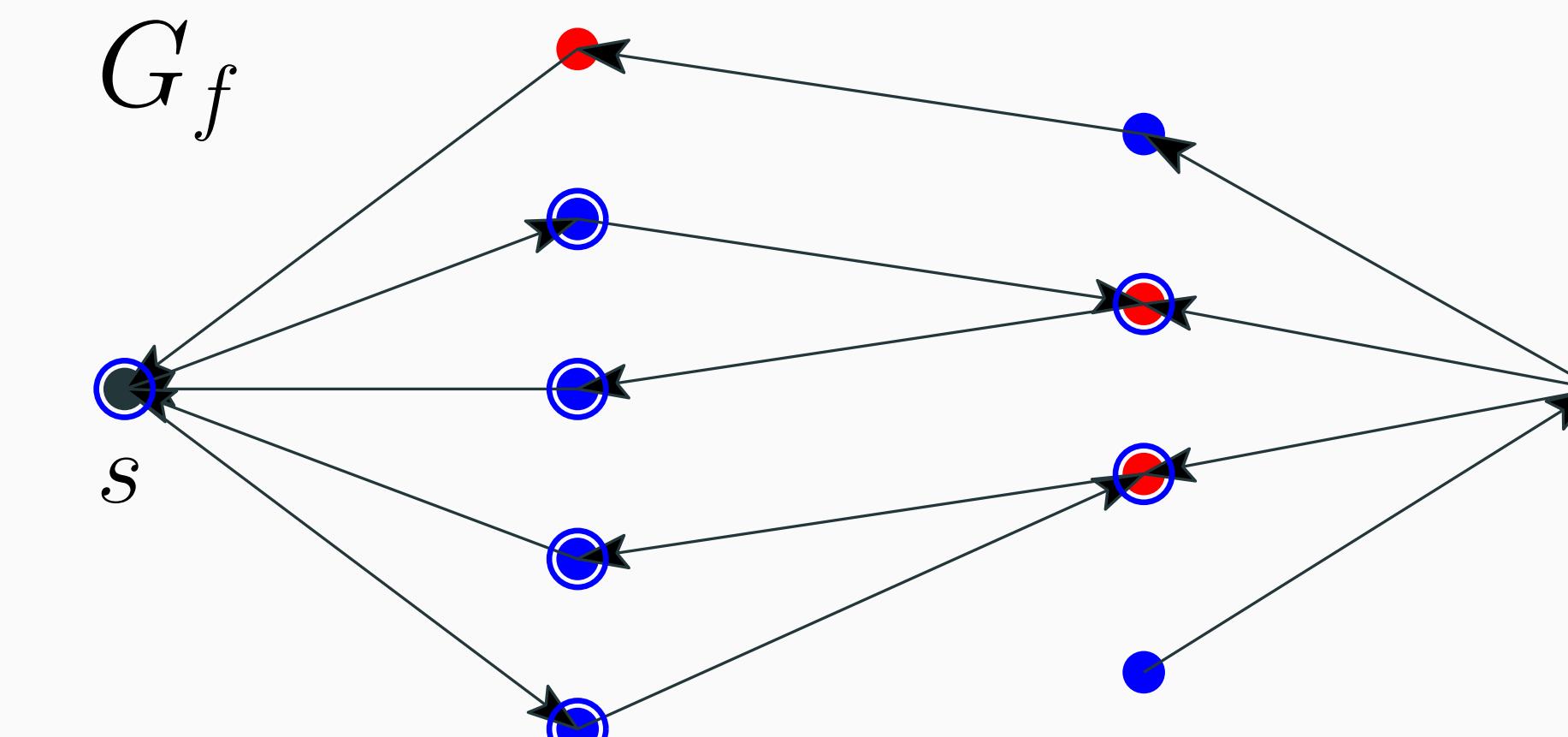
2) Compute the residual graph  $G_f$ :



3) Mark reachable vertices from  $s$  with BFS:



4) Read the MinVC or MaxIS from the marks:



# Summary: MaxFlowMinCut and Bipartite Matching

What you should remember:

## Minimum Cut

- ▶ Theorem: maximum amount of any  $s$ - $t$ -flow = minimum capacity of any  $s$ - $t$ -cut
- ▶ Finding the cut: BFS/DFS on residual graph starting from  $s$ .

## Vertex Cover

- ▶ Minimum vertex cover and maximum independent set are hard problems.
- ▶ Bipartite graphs allow fast MinVC and MaxIS (both on top of maximum matching).
- ▶ Finding the minimum vertex cover: BFS/DFS on residual graph from  $s$ .

# Min Cost Max Flow

---

# Minimum Cost for a Bipartite Matching

How to pick the *best* maximum matching? How to break ties among equally large ones?

E.g.: Which set of marriages is the most stable?

Or: Which consumer/producer pairing is the most effective?

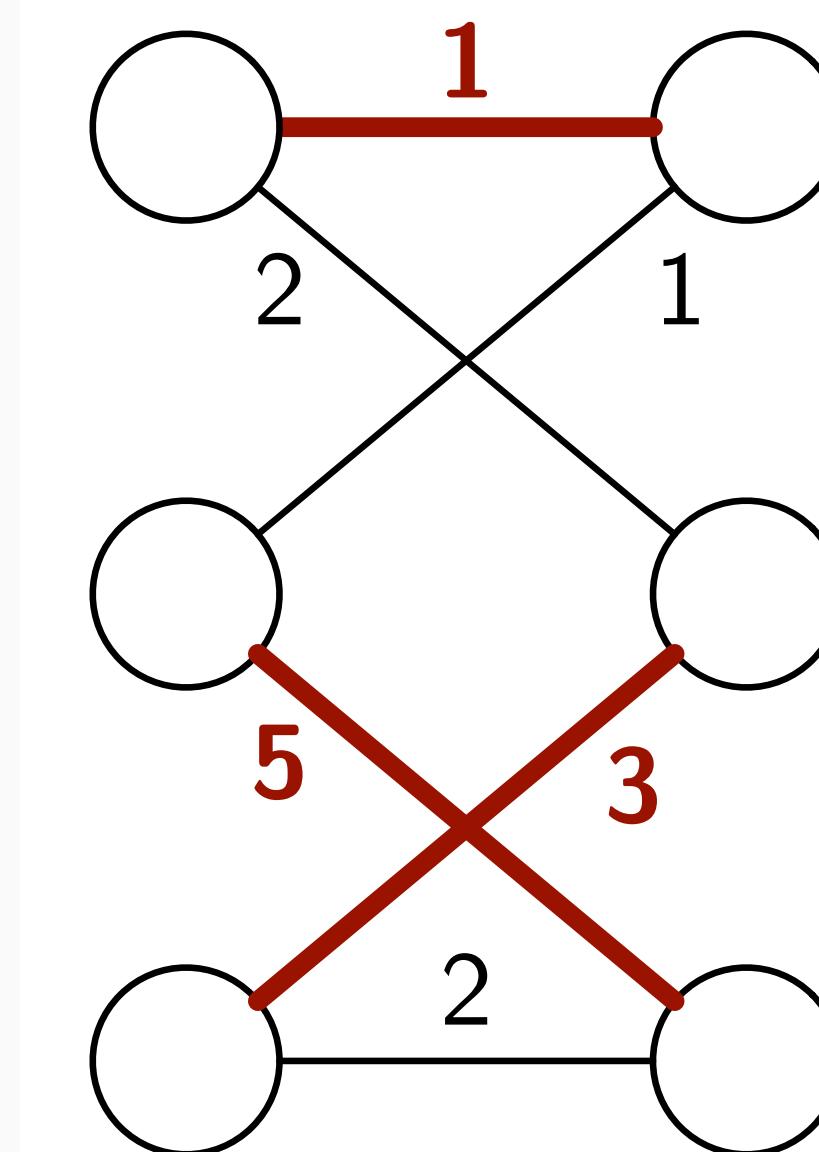
What if the edges in a matching also have costs associated?

- ▶ cardinality of the matching is no longer the only objective
- ▶ second priority: minimize the total cost

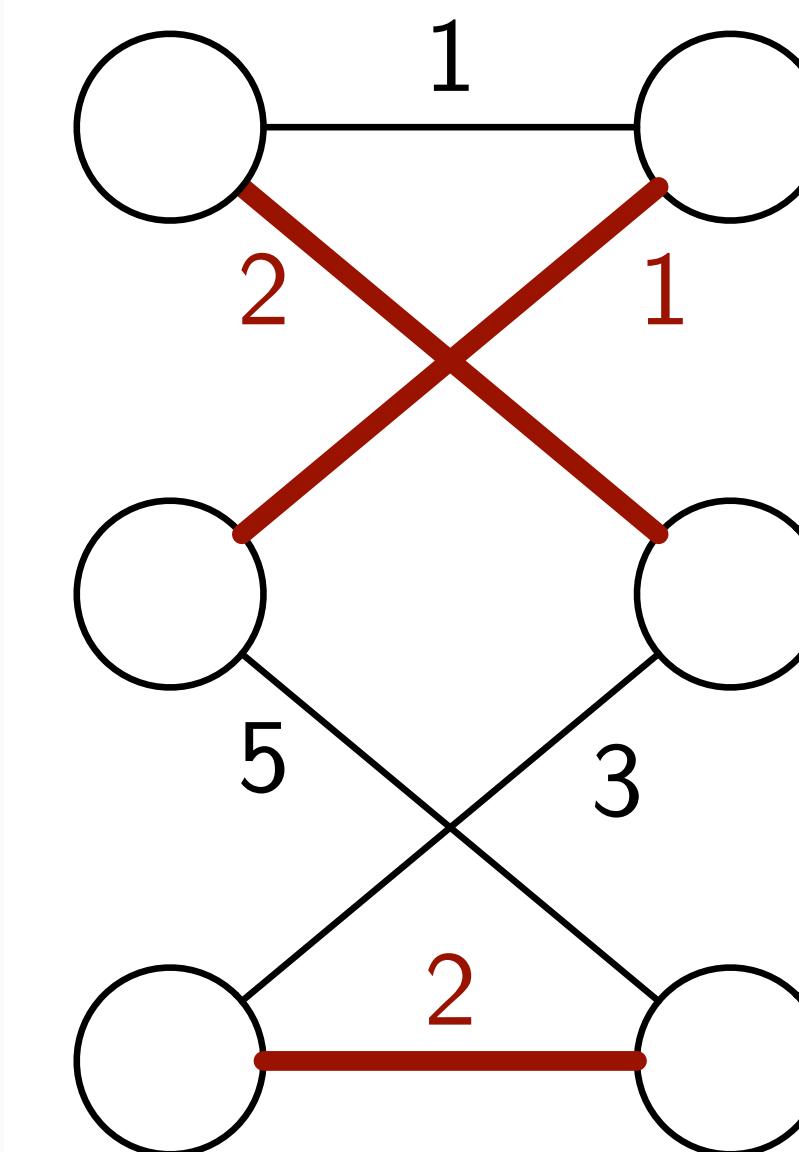
We search for the cheapest among all maximum matchings.

This does not fit into our model of flows.

two maximum matchings of different cost:



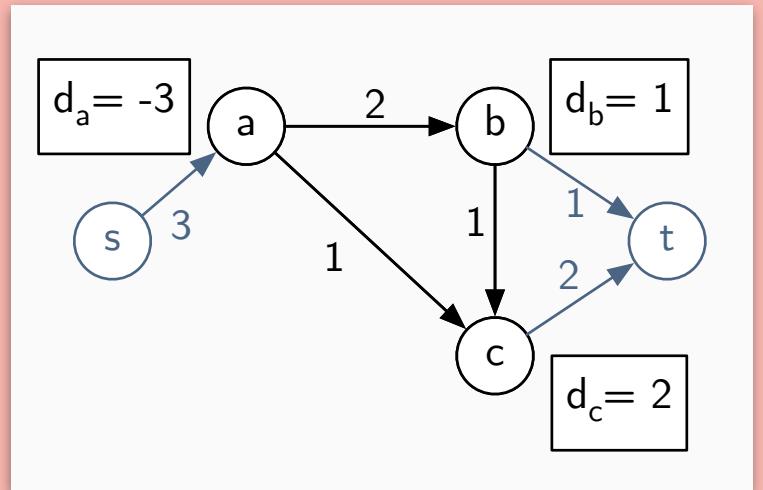
$$1 + 5 + 3 = 9$$



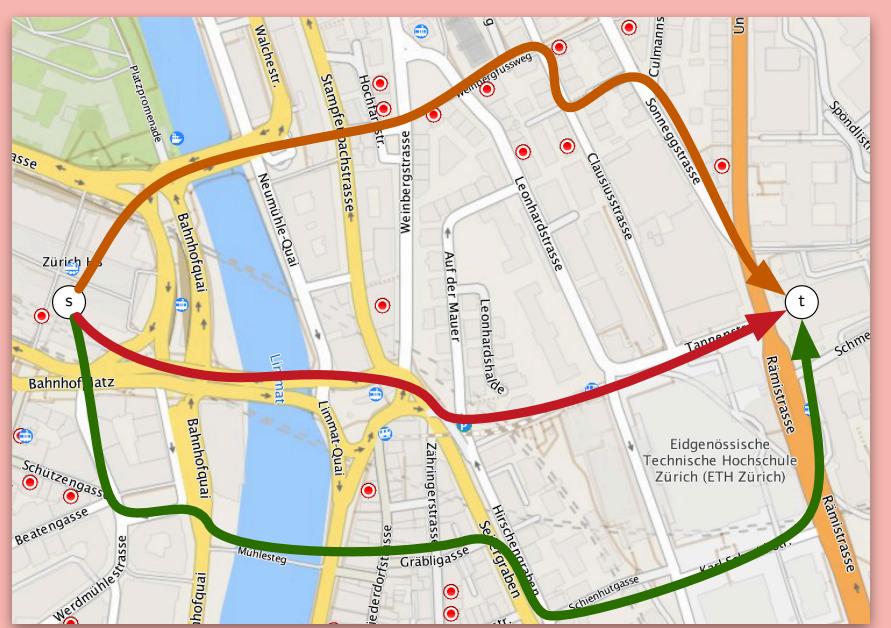
$$2 + 1 + 2 = 5$$

# Problem Landscape

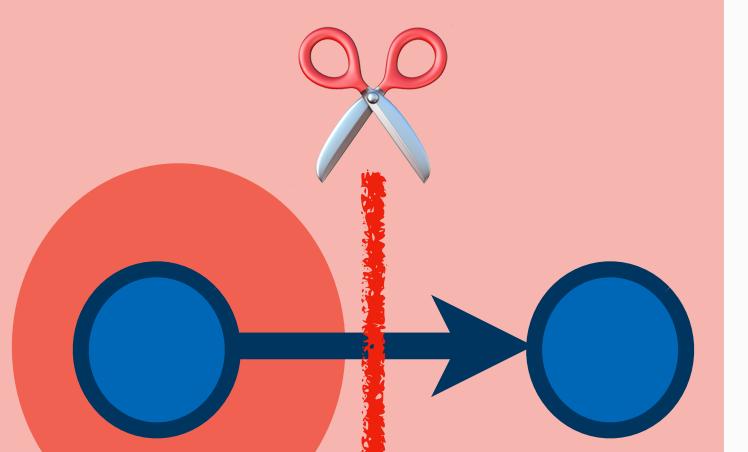
Circulation



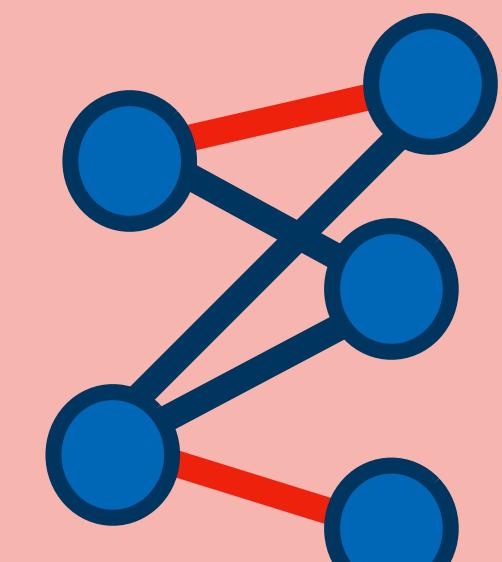
Edge-disjoint Paths



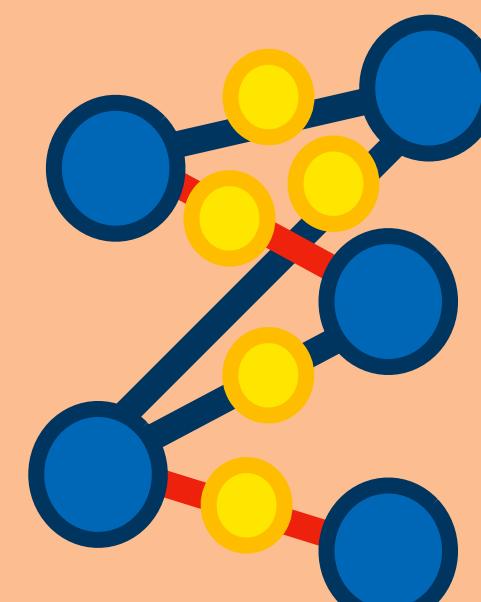
Minimum Cut



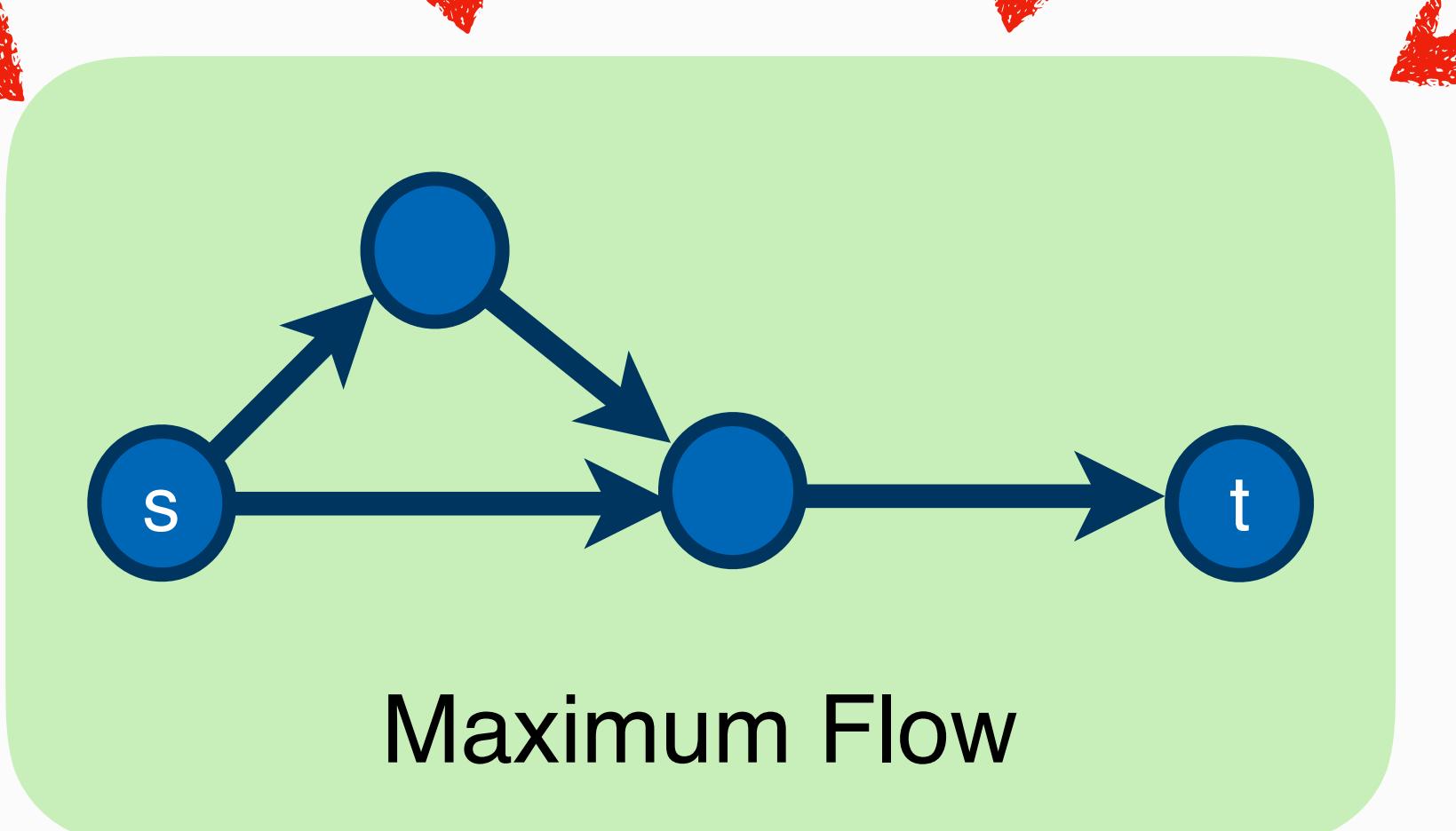
Bipartite Matching



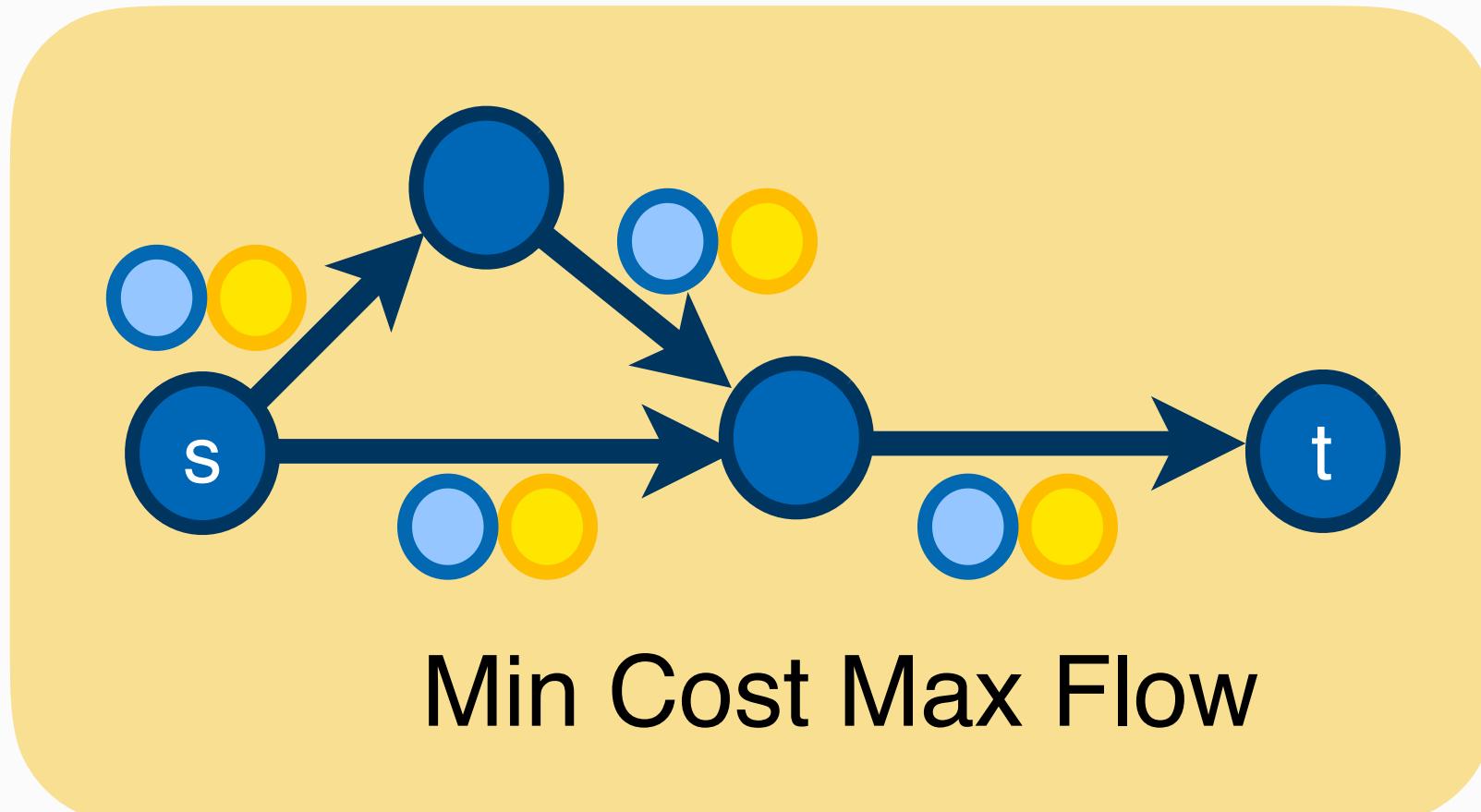
Bipartite Cost Matching



Maximum Flow



Min Cost Max Flow



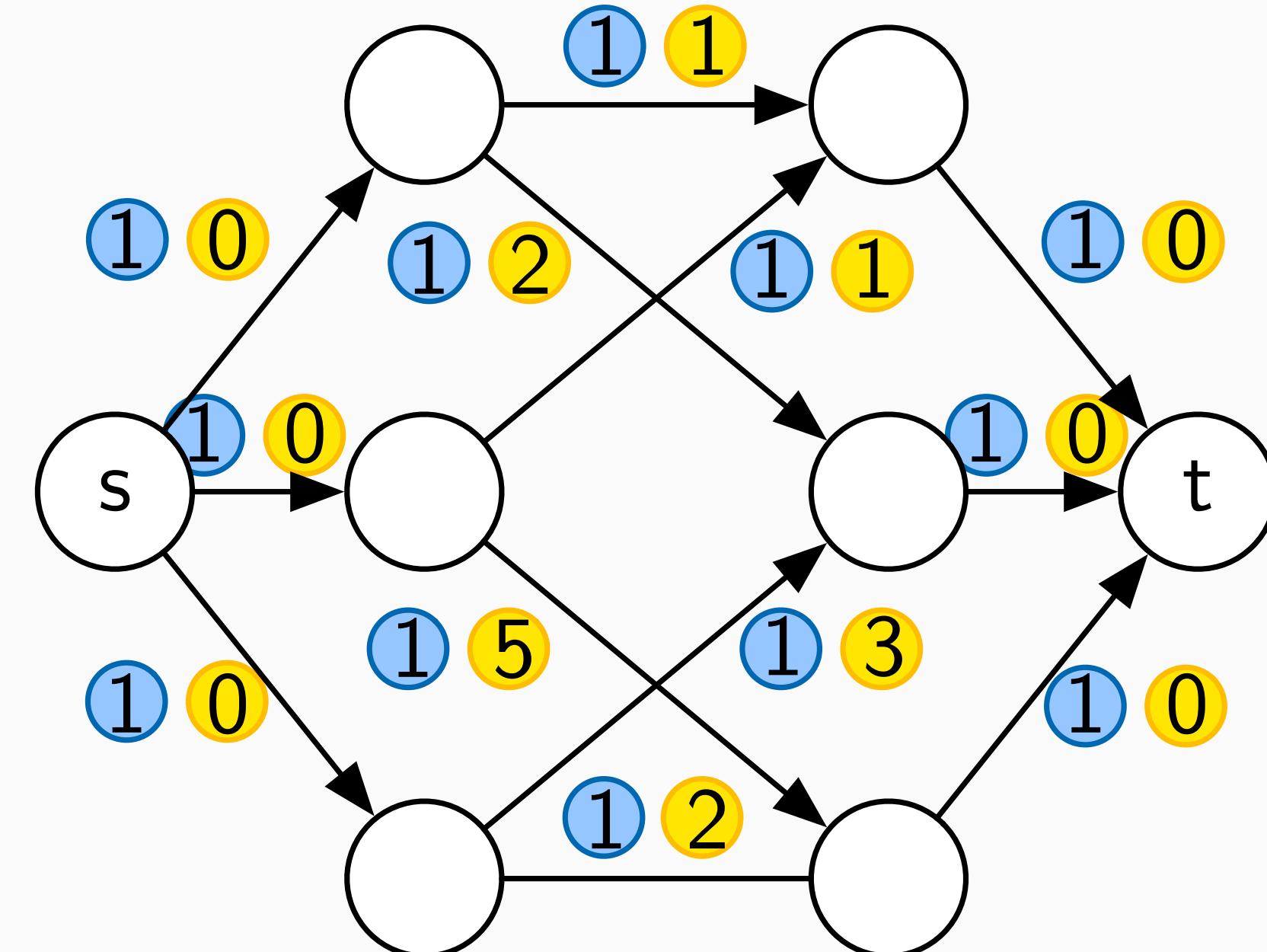
# More General Model: Minimum Cost Maximum Flow

We extend the network flow problem by allowing edge costs that occur per unit of flow.

Input: A flow network consisting of

- directed graph  $G = (V, E)$
- source and sink  $s, t \in V$
- edge capacity  $cap : E \rightarrow \mathbb{N}$
- edge cost  $cost : E \rightarrow \mathbb{Z}$ .

Output: A flow function  $f$  with minimal  $cost(f) = \sum_{e \in E} f(e) \cdot cost(e)$  among all flows with maximal  $|f|$ .



This can model much more than just minimum cost bipartite matching.

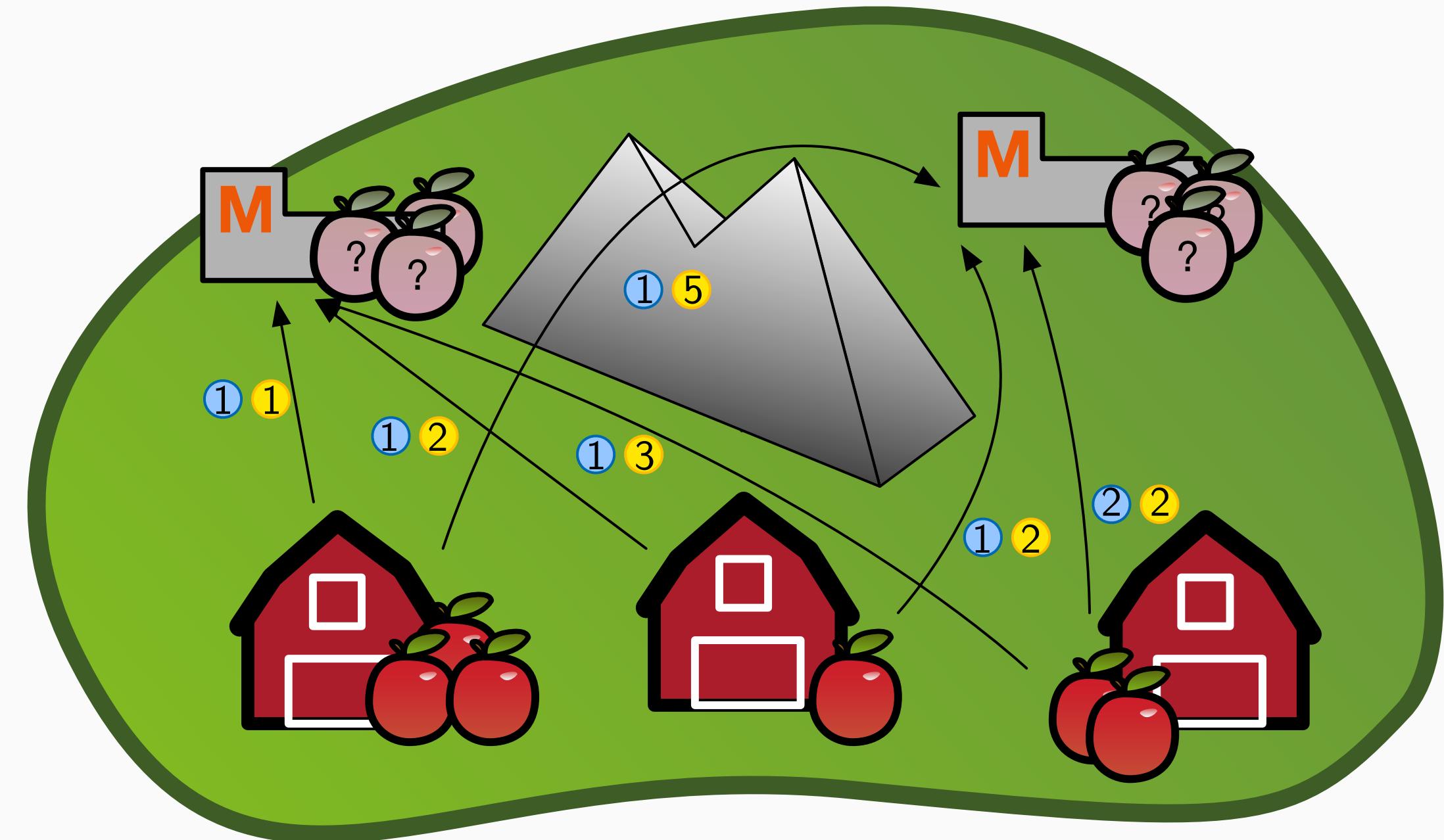
**Legend:** capacity vs cost

# Example: Fruit Delivery

Migros wants to schedule fruit deliveries from their farmers to their shops.

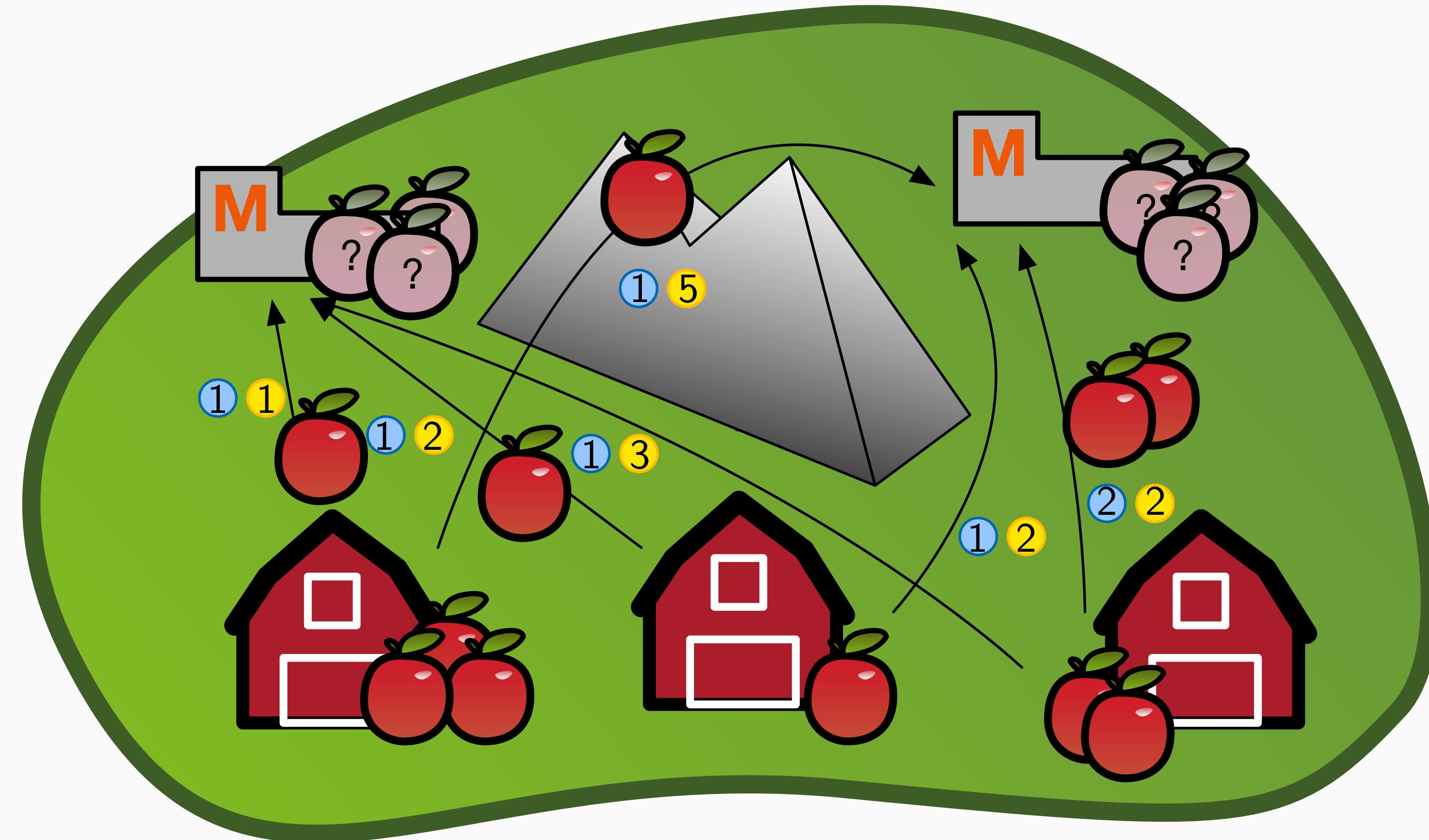
They know all important parameters;

- ▶ production per farm [in kg]
- ▶ demand per shop [in kg]
- ▶ transportation capacity [in kg] and  
transportation cost [in Fr. pro kg]  
for every farm-shop pair



Note: This is not just a bipartite matching, even though the graph is bipartite.  
One farm might deliver to multiple shops (and vice versa).

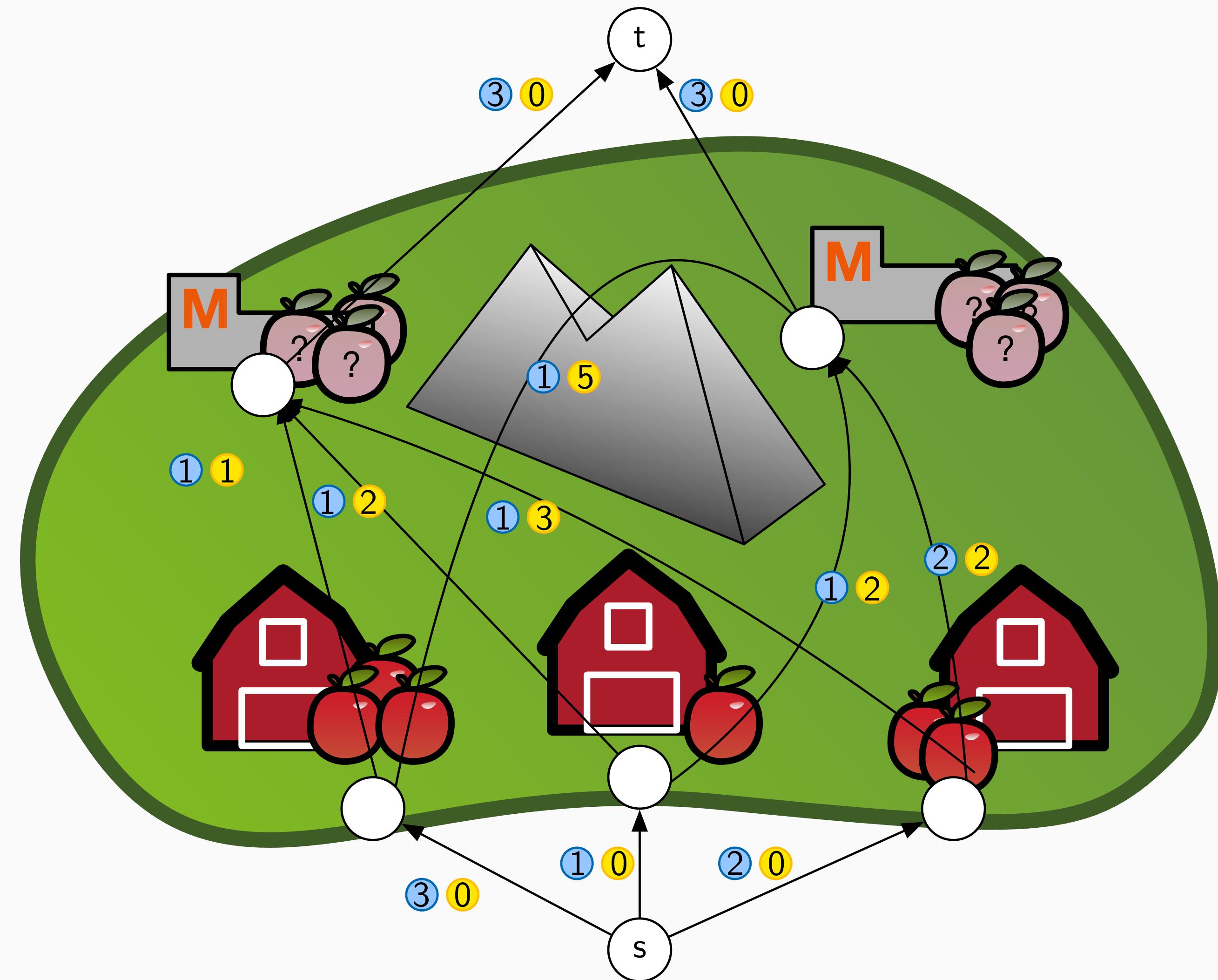
# Example: Fruit Delivery



$$\text{Flow: } 2 + 1 + 2 = 5$$

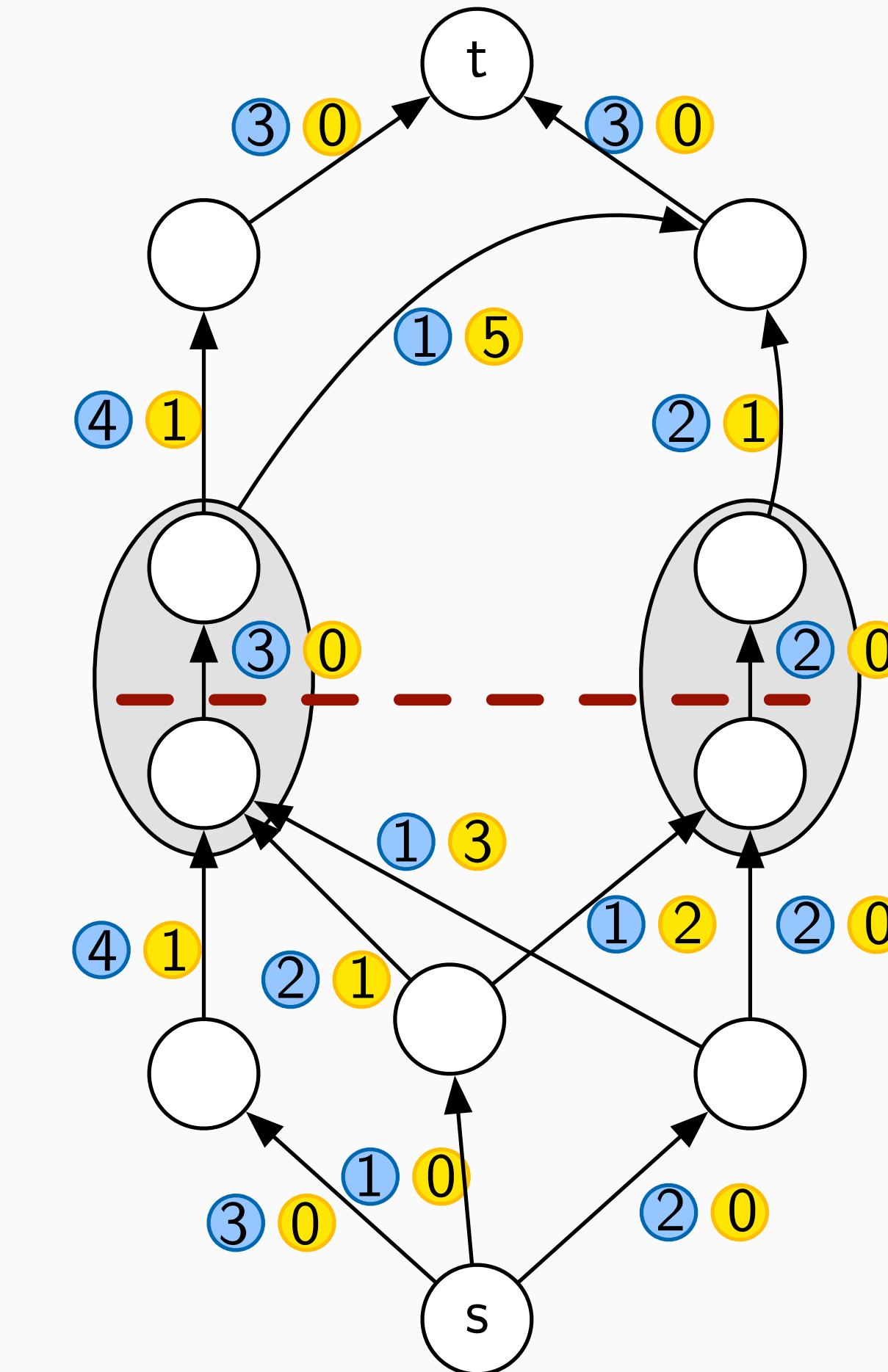
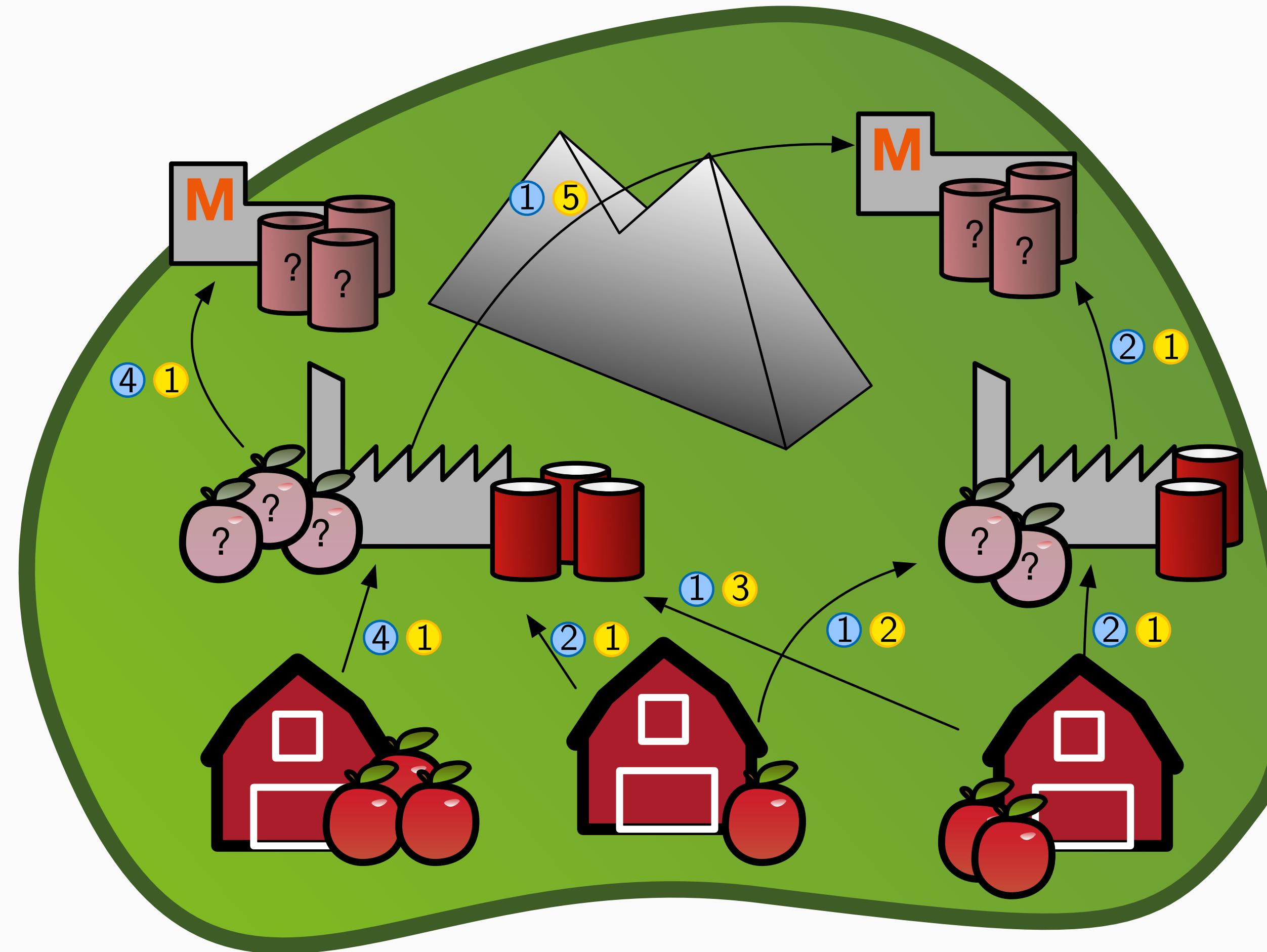
$$\text{Cost: } 1 \cdot 1 + 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 2 = 12$$

# Example: Fruit Delivery



# Extended Example: Canned Fruit Delivery

Extension: Canned fruit requires transportation to and from a canning factory.

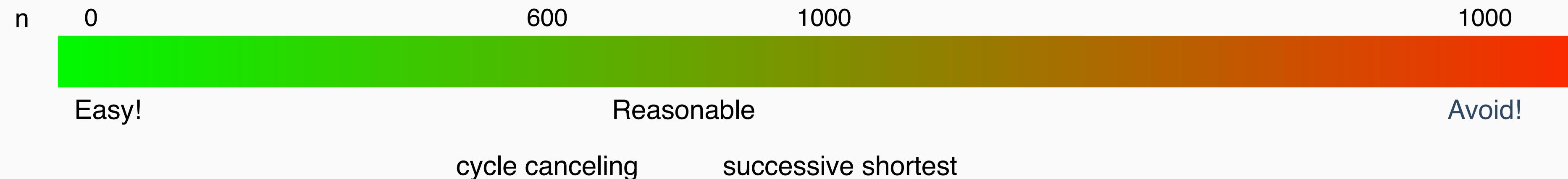


# Min Cost Max Flow with BGL

There are two algorithms available in BGL (available in BGL v1.55+):

- ▶ **`cycle_canceling()`**
  - ▶ slow, but can handle negative costs
  - ▶ needs a maximum flow to start with (call e.g. `push_relabel_max_flow` before)
  - ▶ runtime  $\mathcal{O}(C \cdot (nm))$  where  $C$  is the cost of the initial flow
  - ▶ [\[BGL documentation\]](#), [\[BGL example\]](#).
- ▶ **`successive_shortest_path_nonnegative_weights()`**
  - ▶ faster, but works only for non-negative costs
  - ▶ sum up all residual capacities at the source to get the flow value
  - ▶ runtime  $\mathcal{O}(|f| \cdot (m + n \log n))$
  - ▶ [\[BGL documentation\]](#), [\[BGL example\]](#).

Rough guide for  $m \approx n$ ,  $|C|, |f| \ll n$



# Min Cost Max Flow with BGL

There are two algorithms available in BGL (available in BGL v1.55+):

- ▶ **cycle\_canceling()**
  - ▶ slow, but can handle negative costs
  - ▶ needs a maximum flow to start with (call e.g. `push_relabel_max_flow` before)
  - ▶ runtime  $\mathcal{O}(C \cdot (nm))$  where  $C$  is the cost of the initial flow
  - ▶ [\[BGL documentation\]](#), [\[BGL example\]](#).
- ▶ **successive\_shortest\_path\_nonnegative\_weights()**
  - ▶ faster, but works only for non-negative costs
  - ▶ sum up all residual capacities at the source to get the flow value
  - ▶ runtime  $\mathcal{O}(|f| \cdot (m + n \log n))$
  - ▶ [\[BGL documentation\]](#), [\[BGL example\]](#).

Useful things to know:

- ▶ costs implemented as `edge_weight_t` property
- ▶ call `find_flow_cost()` to compute the cost of the flow

# Min Cost Max Flow with BGL

Weights and capacities, just one more nesting level in the typedefs:

```
16 // Graph Type with nested interior edge properties for Cost Flow Algorithms
17 typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> traits;
18 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
19     boost::property<boost::edge_capacity_t, long,
20     boost::property<boost::edge_residual_capacity_t, long,
21         boost::property<boost::edge_reverse_t, traits::edge_descriptor,
22             boost::property <boost::edge_weight_t, long> >>> graph; // new!
23
24 typedef boost::graph_traits<graph>::edge_descriptor edge_desc;
25 typedef boost::graph_traits<graph>::out_edge_iterator out_edge_it; // Iterator
```

Code file: → [bgl\\_mincostmaxflow.cpp](#)

# Min Cost Max Flow with BGL

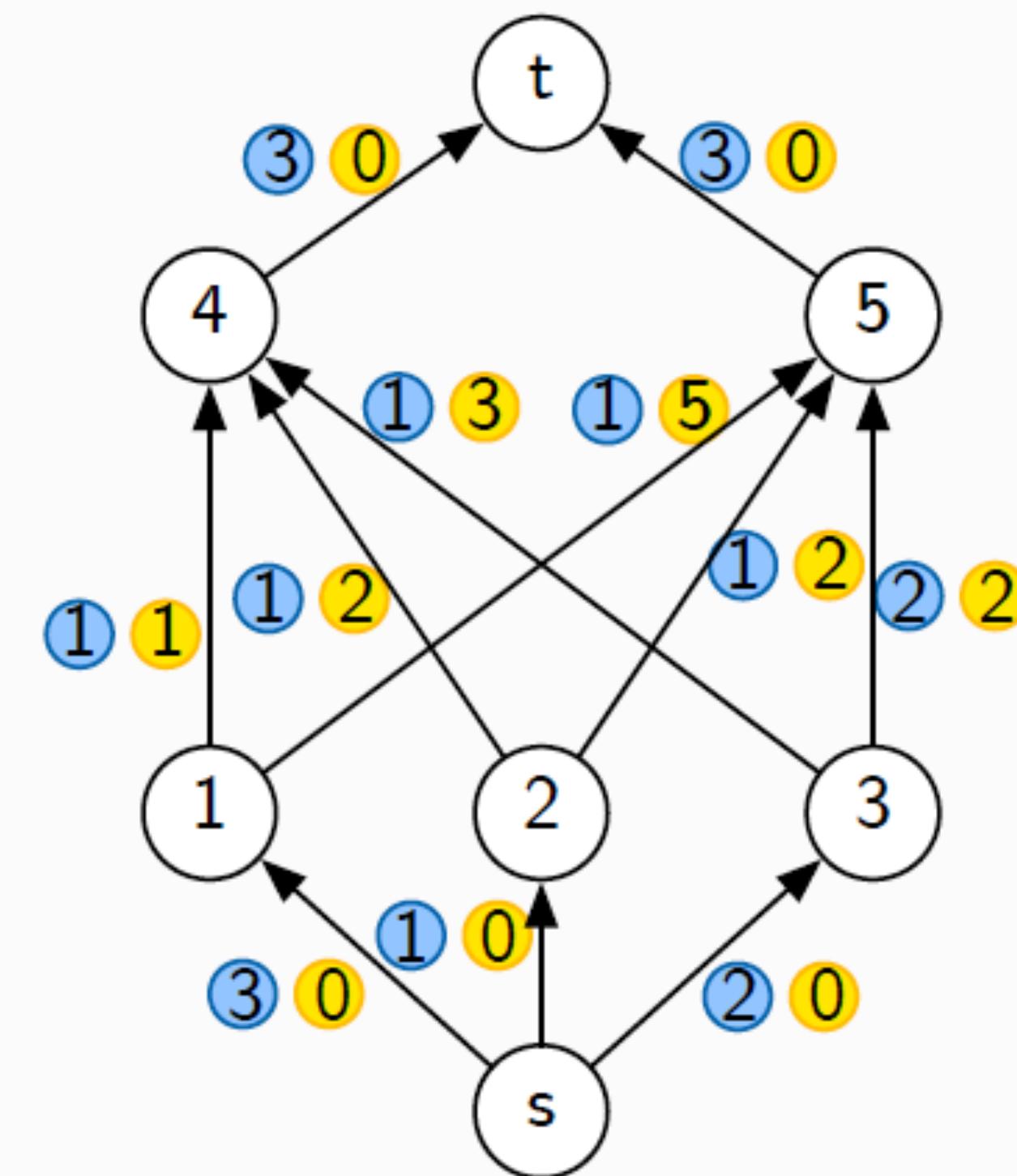
Extending the edge adder:

```
28 class edge_adder {
29     graph &G;
30
31 public:
32     explicit edge_adder(graph &G) : G(G) {}
33     void add_edge(int from, int to, long capacity, long cost) {
34         auto c_map = boost::get(boost::edge_capacity, G);
35         auto r_map = boost::get(boost::edge_reverse, G);
36         auto w_map = boost::get(boost::edge_weight, G); // new!
37         const edge_desc e = boost::add_edge(from, to, G).first;
38         const edge_desc rev_e = boost::add_edge(to, from, G).first;
39         c_map[e] = capacity;
40         c_map[rev_e] = 0;
41         r_map[e] = rev_e;
42         r_map[rev_e] = e;
43         w_map[e] = cost;    // new assign cost
44         w_map[rev_e] = -cost; // new negative cost
45     }
46 };
```

# Min Cost Max Flow with BGL

## Building the graph

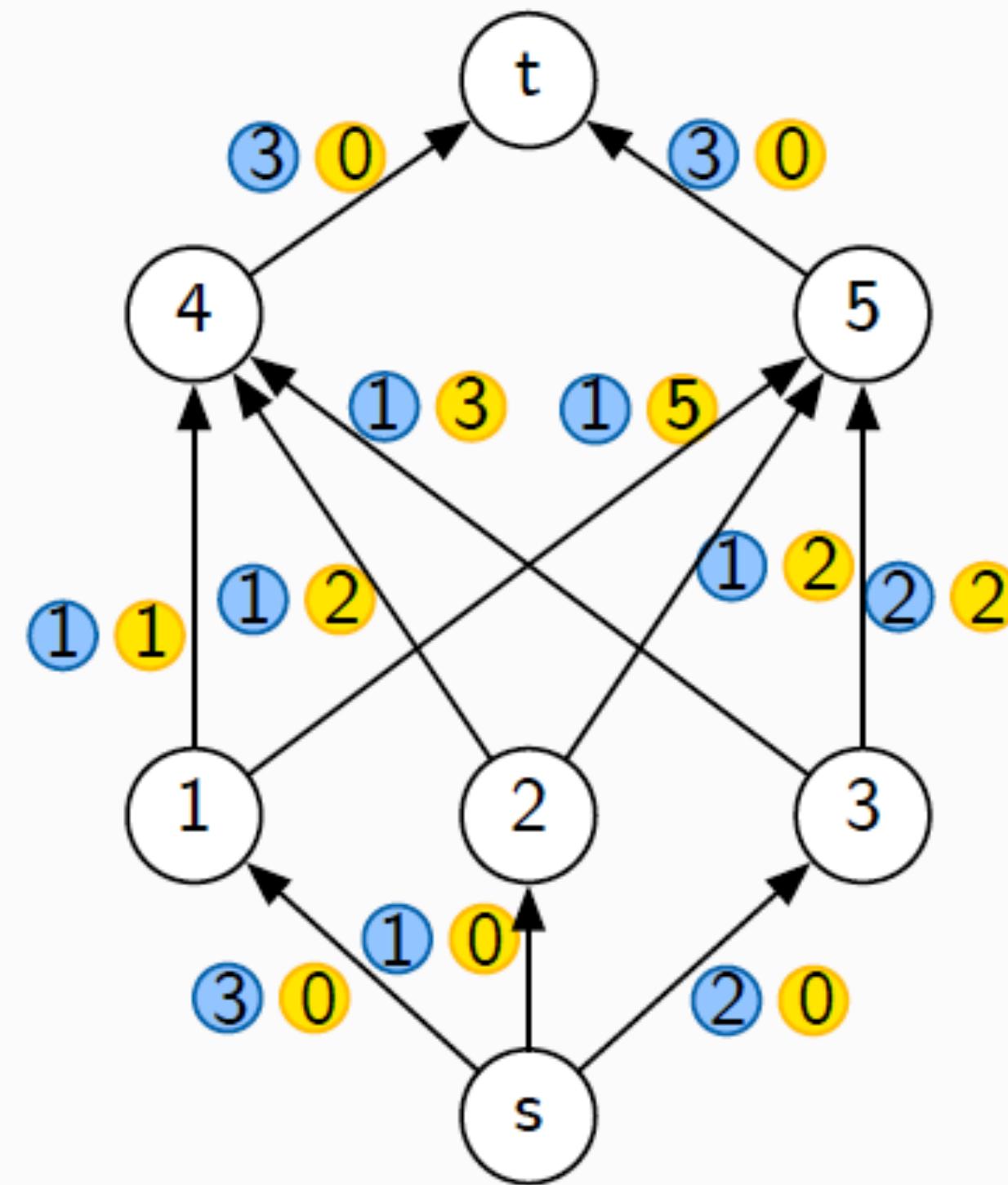
```
50 const int N=7;
51 const int v_source = 0;
52 const int v_farm1 = 1;
53 const int v_farm2 = 2;
54 const int v_farm3 = 3;
55 const int v_shop1 = 4;
56 const int v_shop2 = 5;
57 const int v_target = 6;
58
59 // Create graph, edge adder class and property maps
60 graph G(N);
61 edge_adder adder(G);
62 auto c_map = boost::get(boost::edge_capacity, G);
63 auto r_map = boost::get(boost::edge_reverse, G);
64 auto rc_map = boost::get(boost::edge_residual_capacity, G);
```



# Min Cost Max Flow with BGL

Add the edges:

```
66 // Add the edges
67 adder.add_edge(v_source, v_farm1, 3, 0);
68 adder.add_edge(v_source, v_farm2, 1, 0);
69 adder.add_edge(v_source, v_farm3, 2, 0);
70
71 adder.add_edge(v_farm1, v_shop1, 1, 1);
72 adder.add_edge(v_farm1, v_shop2, 1, 5);
73 adder.add_edge(v_farm2, v_shop1, 1, 2);
74 adder.add_edge(v_farm2, v_shop2, 1, 2);
75 adder.add_edge(v_farm3, v_shop1, 1, 3);
76 adder.add_edge(v_farm3, v_shop2, 2, 2);
77
78 adder.add_edge(v_shop1, v_target, 3, 0);
79 adder.add_edge(v_shop2, v_target, 3, 0);
```



# Min Cost Max Flow with BGL

Running the algorithm:

```
83 // Option 1: Min Cost Max Flow with cycle_canceling
84 int flow1 = boost::push_relabel_max_flow(G, v_source, v_target);
85 boost::cycle_canceling(G);
86 int cost1 = boost::find_flow_cost(G);
87 std::cout << "flow" << flow1 << "\n"; // 5
88 std::cout << "cost" << cost1 << "\n"; // 12
```

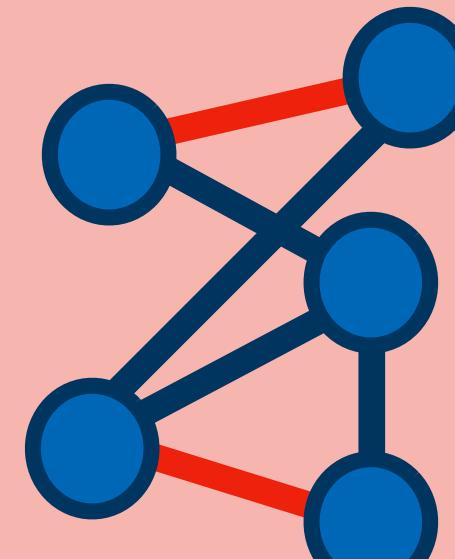
# Min Cost Max Flow with BGL

Running the algorithm:

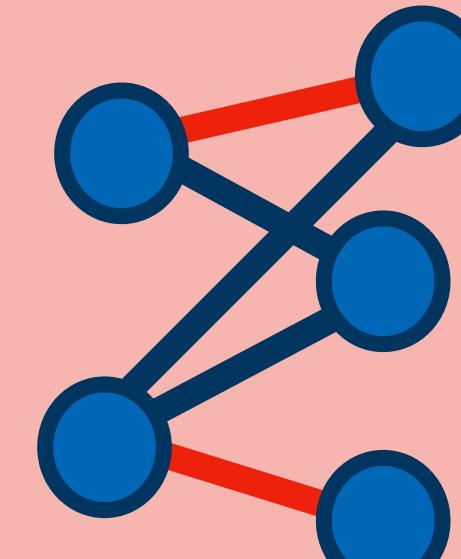
```
92 // Option 2: Min Cost Max Flow with successive_shortest_path_nonnegative_weights
93 boost::successive_shortest_path_nonnegative_weights(G, v_source, v_target);
94 int cost2 = boost::find_flow_cost(G);
95 std::cout << "cost" << cost2 << "\n"; // 12
96 // Iterate over all edges leaving the source to sum up the flow values.
97 int s_flow = 0;
98 out_edge_it e, eend;
99 for(boost::tie(e, eend) = boost::out_edges(boost::vertex(v_source,G), G); e != eend; ++e)
100     s_flow += c_map[*e] - rc_map[*e];
101 std::cout << "s-out_flow" << s_flow << "\n"; // 5
102 // Or equivalently, you can do the summation at the sink, but with reversed edge.
103 int t_flow = 0;
104 for(boost::tie(e, eend) = boost::out_edges(boost::vertex(v_target,G), G); e != eend; ++e)
105     t_flow += rc_map[*e] - c_map[*e];
106 std::cout << "t-in_flow" << t_flow << "\n"; // 5
```

# Problem Landscape

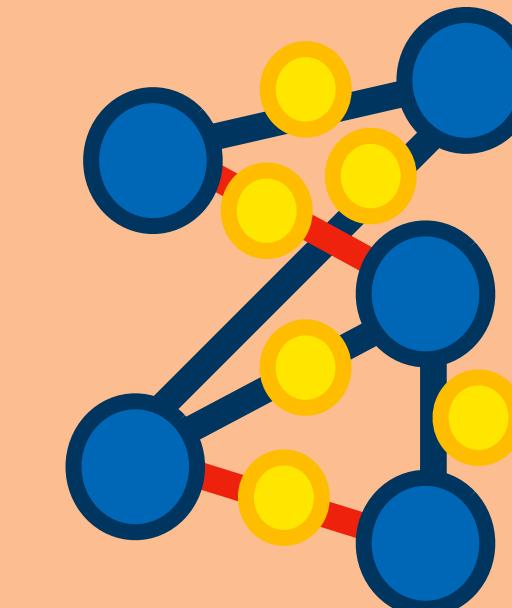
General Matching



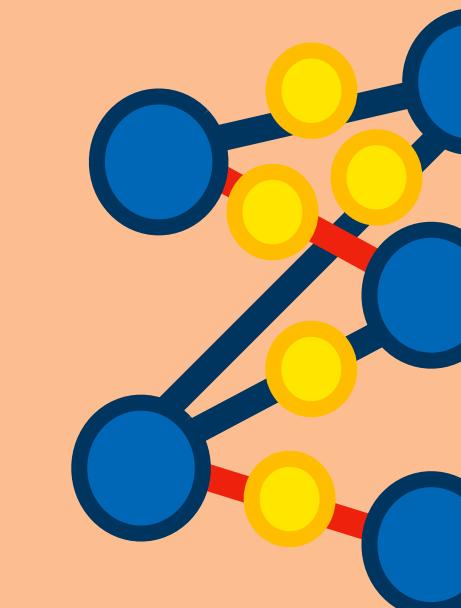
Bipartite Matching



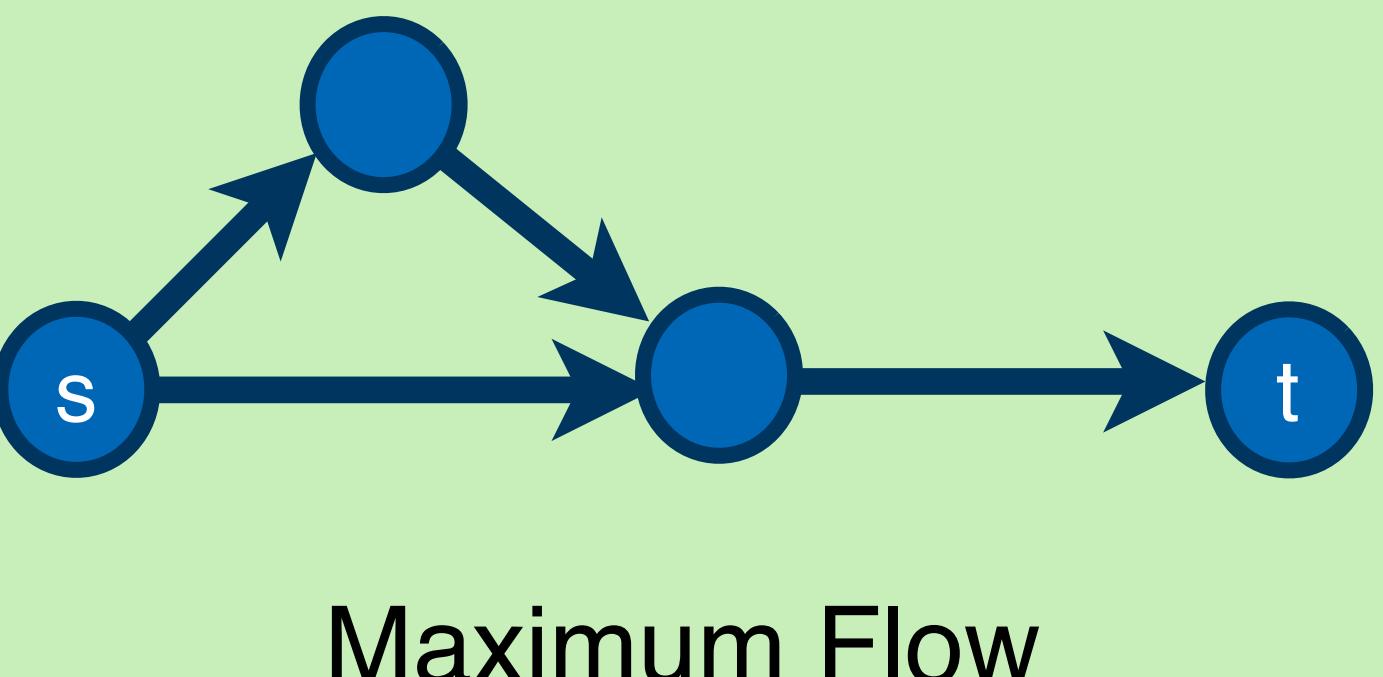
General Cost Matching



Bipartite Cost Matching

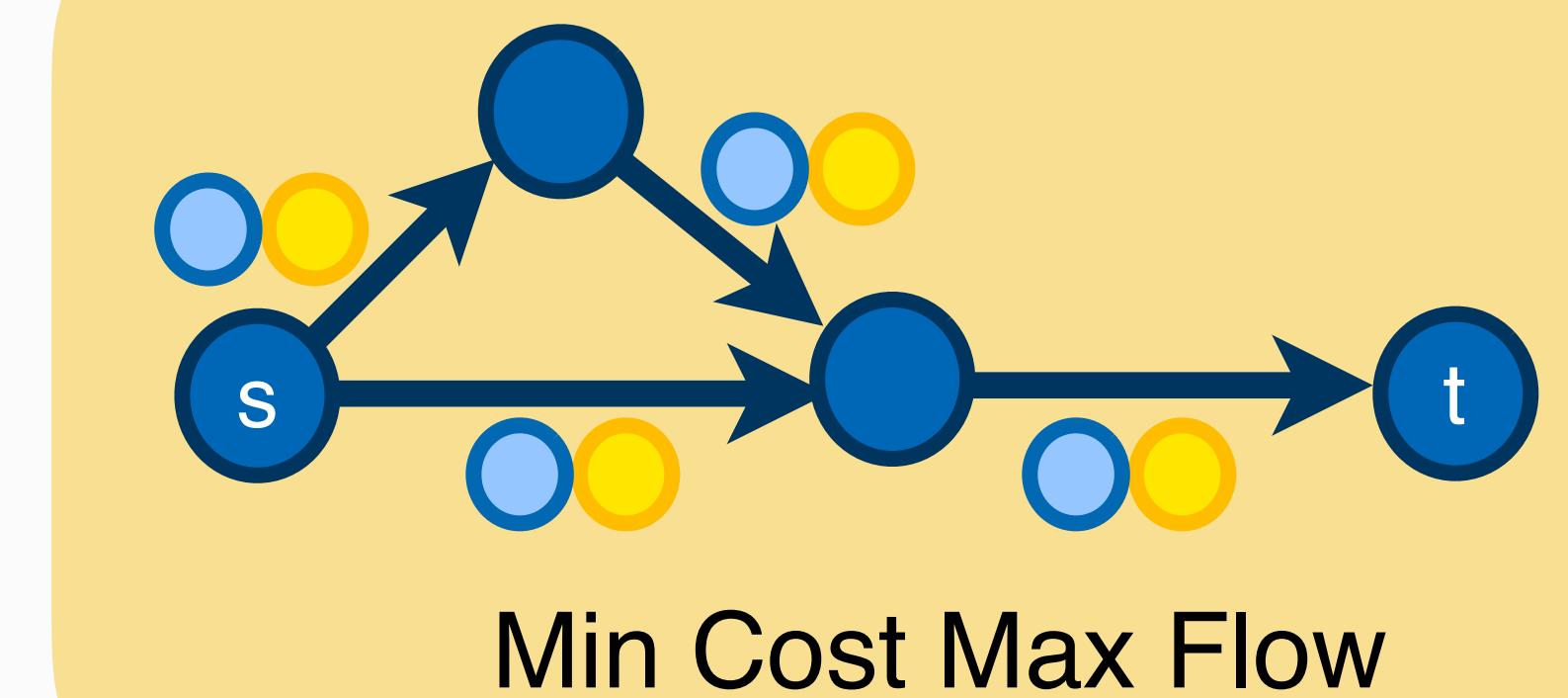


Edmonds  
Maximum  
Cardinality  
Matching



Maximum Flow

???  
not in  
BGL



Min Cost Max Flow

# Summary: Min Cost Max Flow with BGL

What you should remember from this part:

## Minimum Cost Maximum Flow

- ▶ is a powerful and versatile modeling tool.
- ▶ is a tiebreaker among several maximum flows (but might still not be unique).
- ▶ = *maximum* cost maximum flow with negated costs.
- ▶ can often be reformulated without negative costs which allows us to use a faster algorithm in BGL (key step in many problems).
- ▶ can easily be implemented when starting with [our template](#)
- ▶ is not harder to use in BGL than the regular network flow algorithms, see [BGL Docs](#).

If you are interested in the theory behind these algorithms: (not needed for this course)

- ▶ Stanford CS 261 by Prof. Tim Roughgarden, Spring 2016, [full lectures on Youtube](#)

# A little bit of a conclusion...

No new theory and no new tools past this point

But be prepared to combine all your skills:

- ▶ On top of a flow problem, do binary search for the answer
- ▶ LP formulation vs. flow formulation?
- ▶ Some graph problems can be solved greedily (e.g. MST), others not (e.g. flow)
- ▶ Dijkstra and MinCostMaxFlows are just “special” dynamic programs
- ▶ Find a Min Cost Max Flow formulation where greedy fails for non-unit weights
- ▶ Do BFS on Delaunay triangulation or do Union-Find on Euclidean MST
- ▶ ...

Starting next week:

- ▶ How to balance reading, solving, coding, debugging under time constraints
- ▶ No more problems labeled by topic – figure it out yourself
- ▶ In-class exercises on Wednesdays