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

Tim Taubner[1]

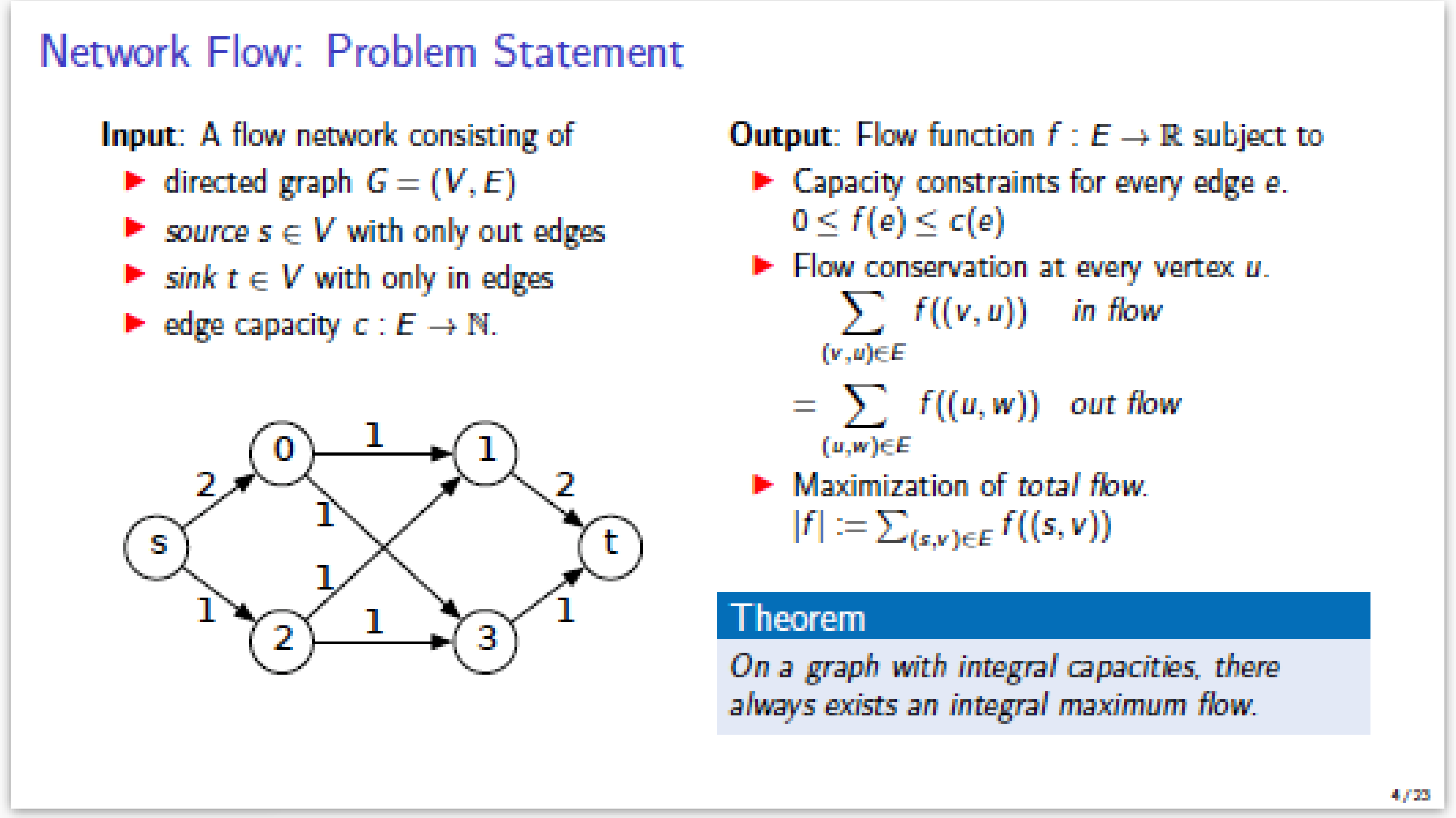
November 13, 2019

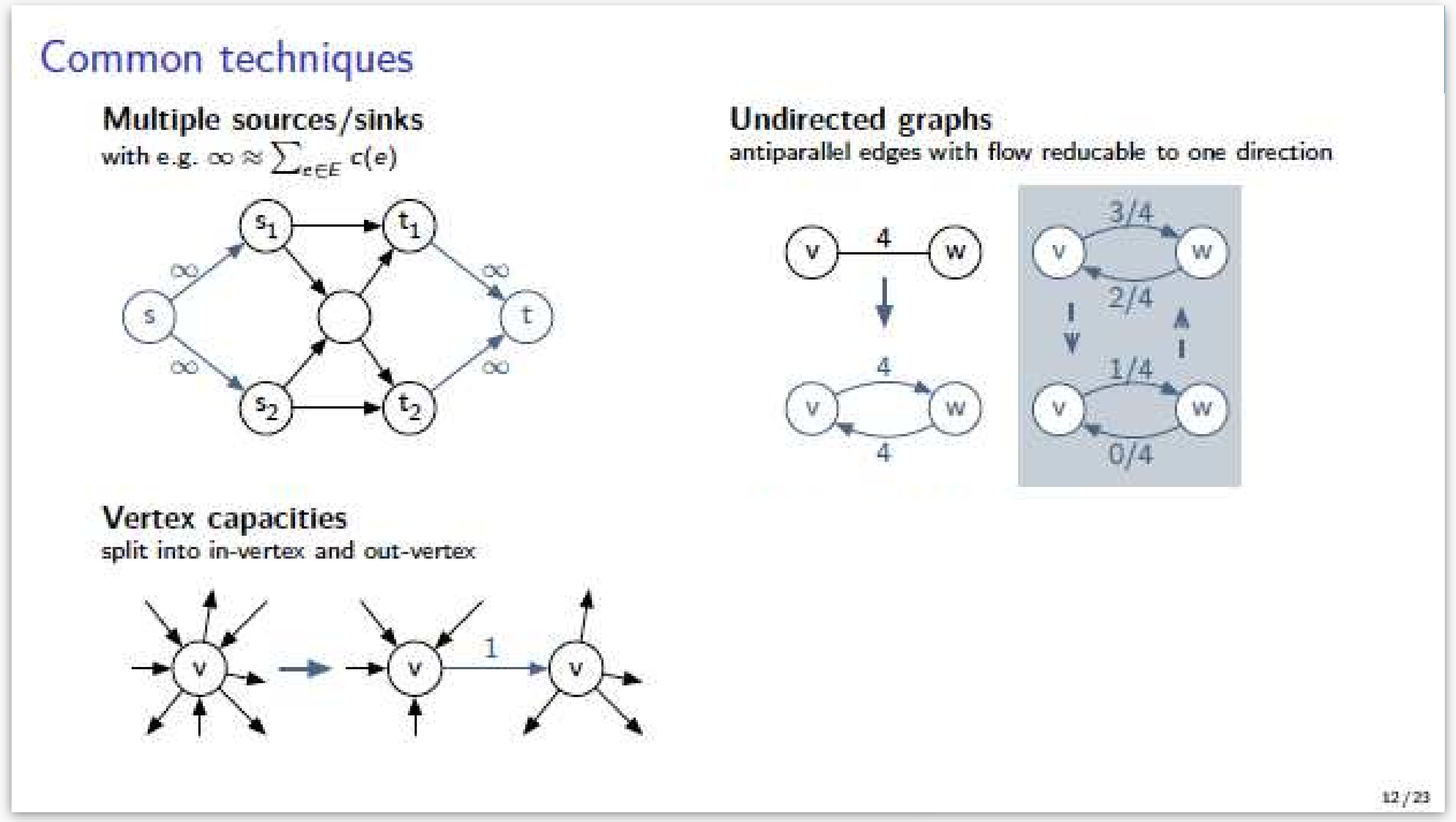
ETH Zürich,

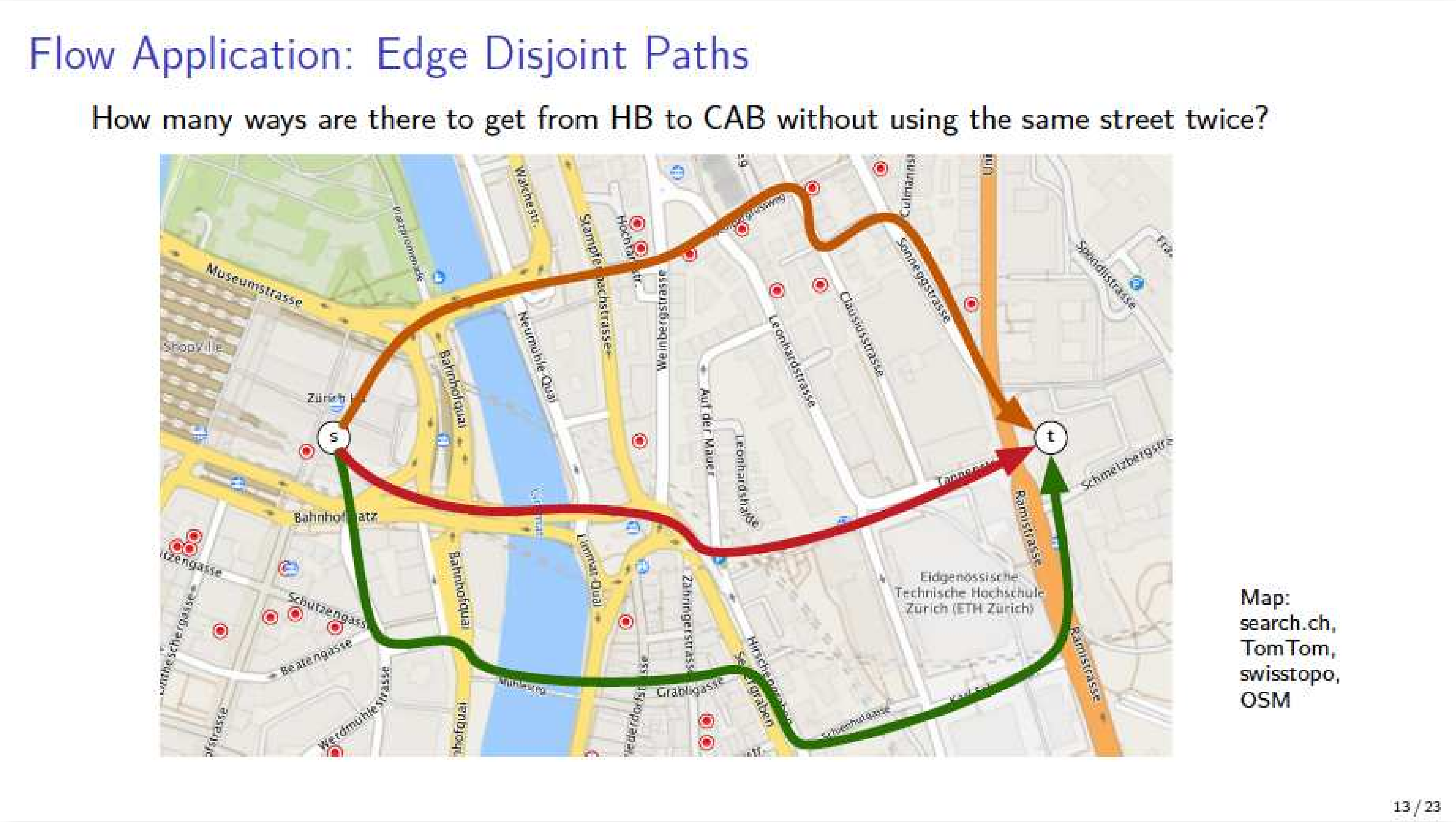
[1] *based on material from Daniel Graf and Andreas Bärtschi*

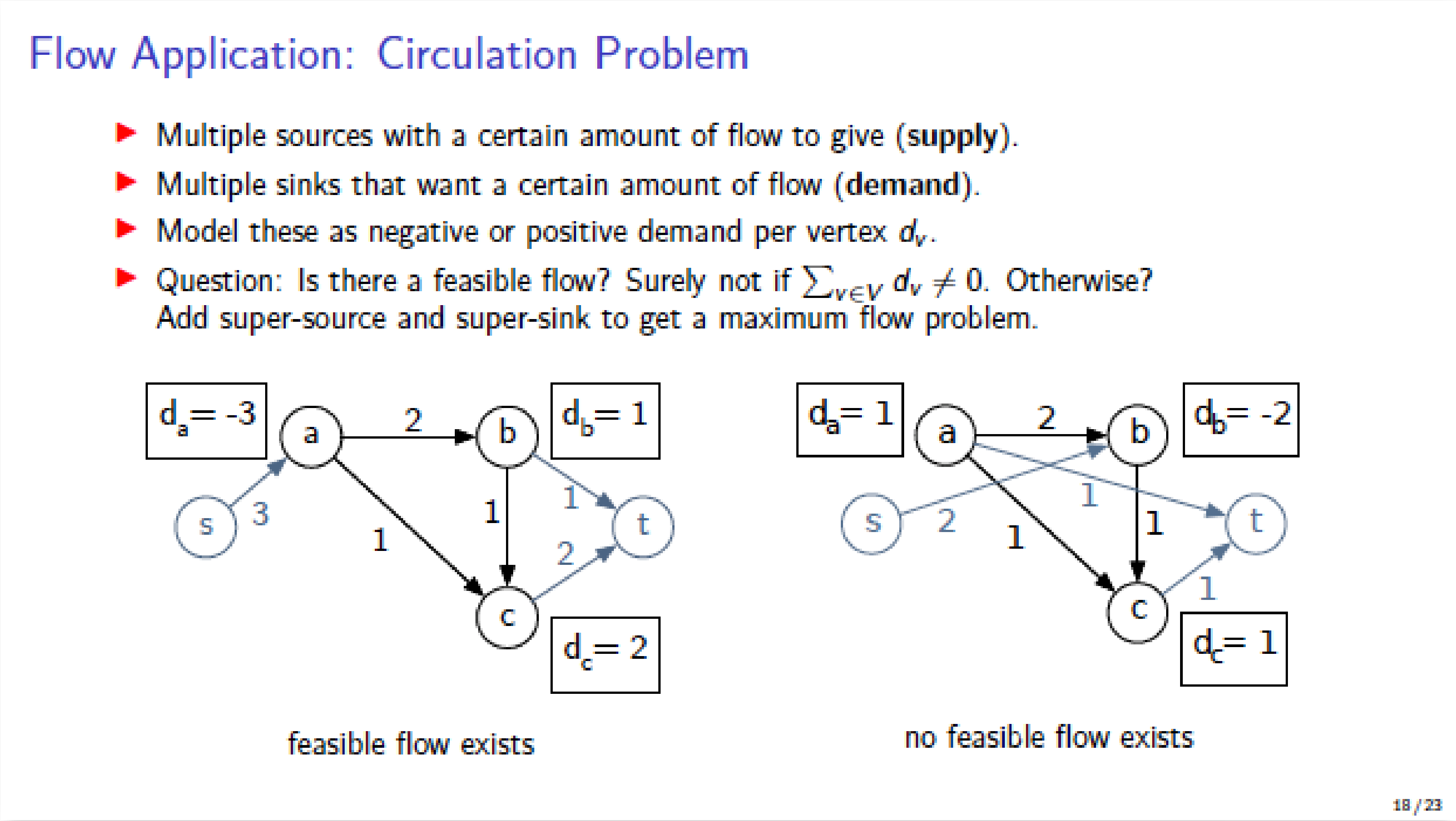
Recap: Basic Network Flows – What did we see last time?

One problem to rule them all…





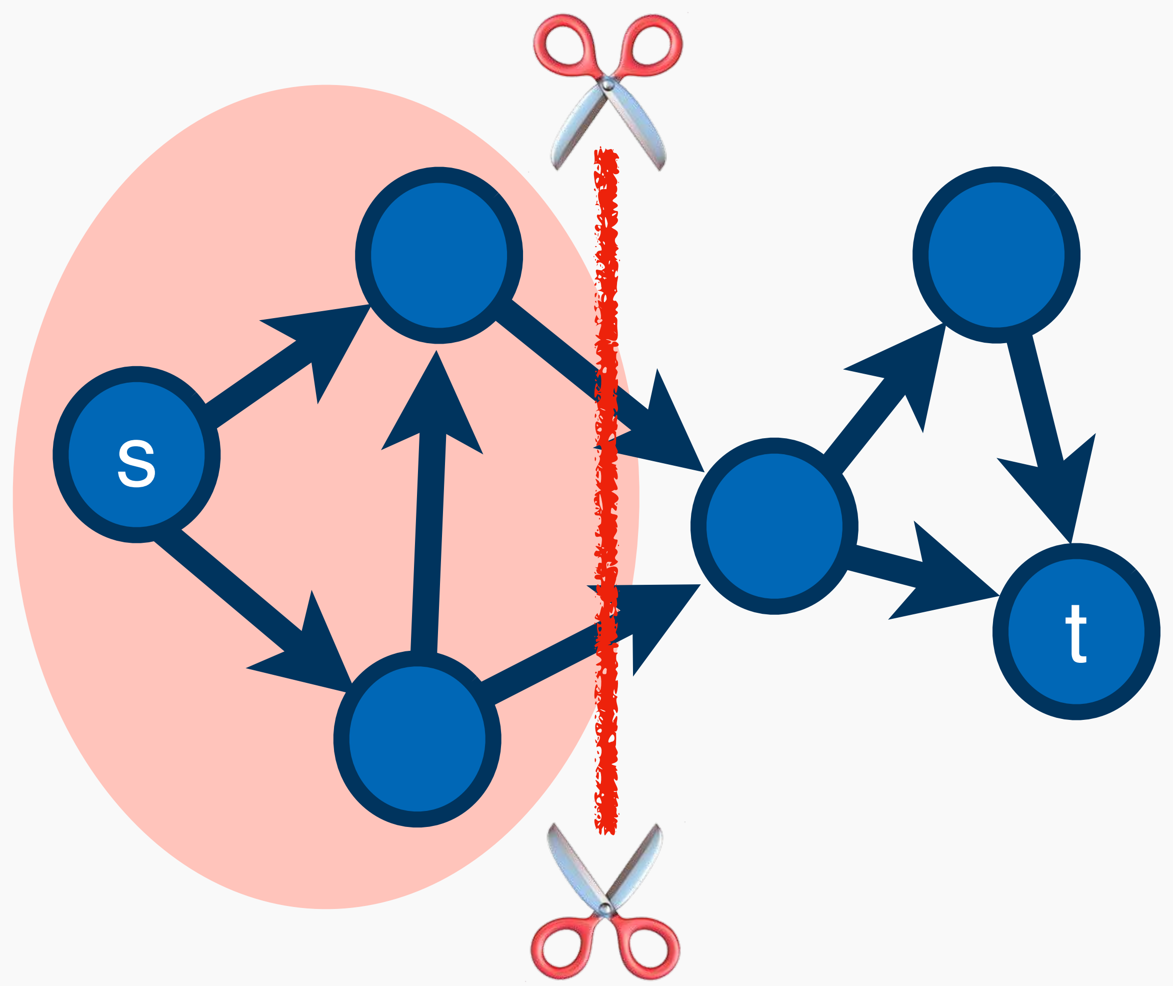




Today: Advanced Network Flow – What else are flows useful for?

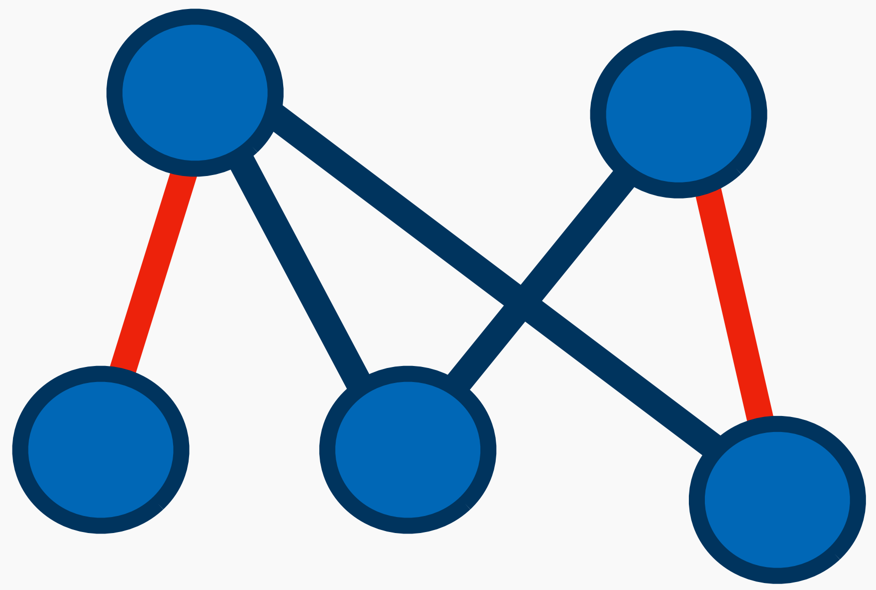
Minimum Cuts

- How to disconnect t from s cheaply?



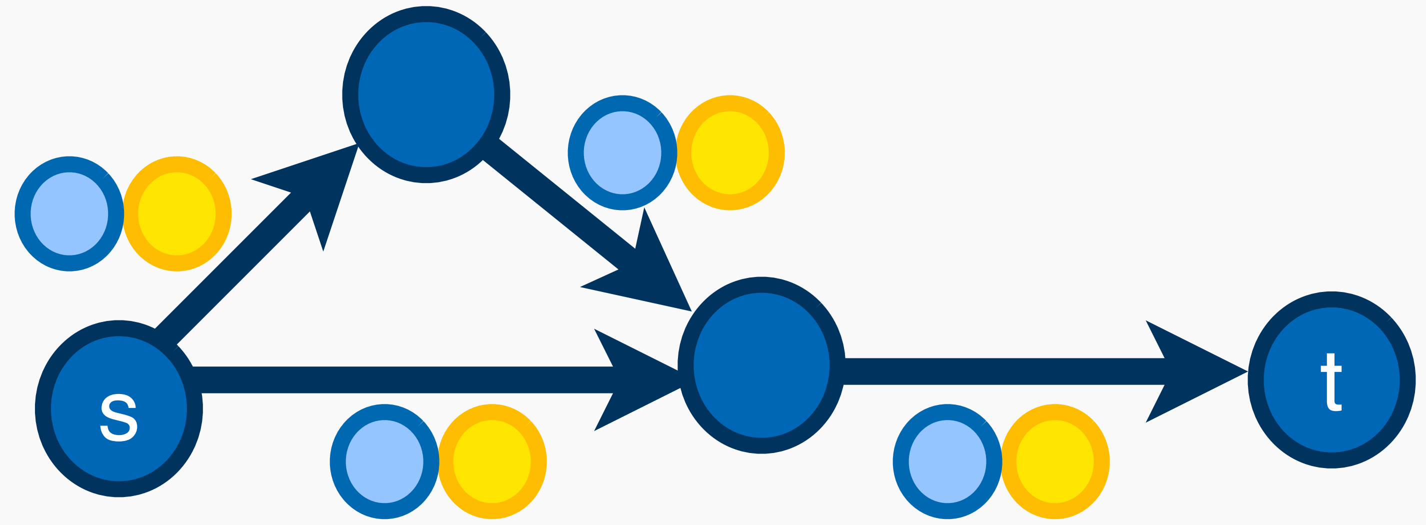
Bipartite Matching

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



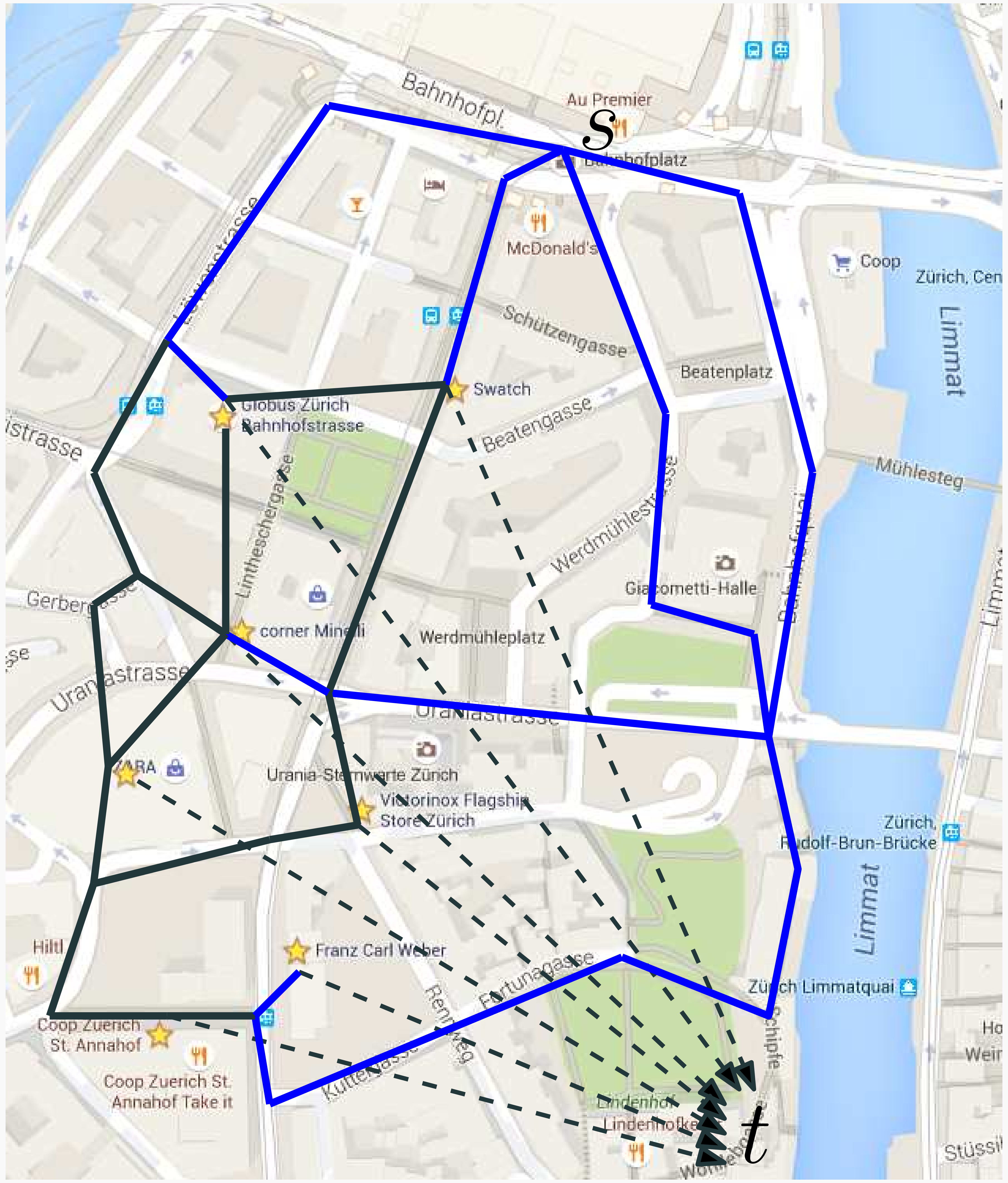
Flows with Costs

- What if sending flow comes with a price?



Minimum Cut

Minimum Cut: Shopping Trip



Start from HB:

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

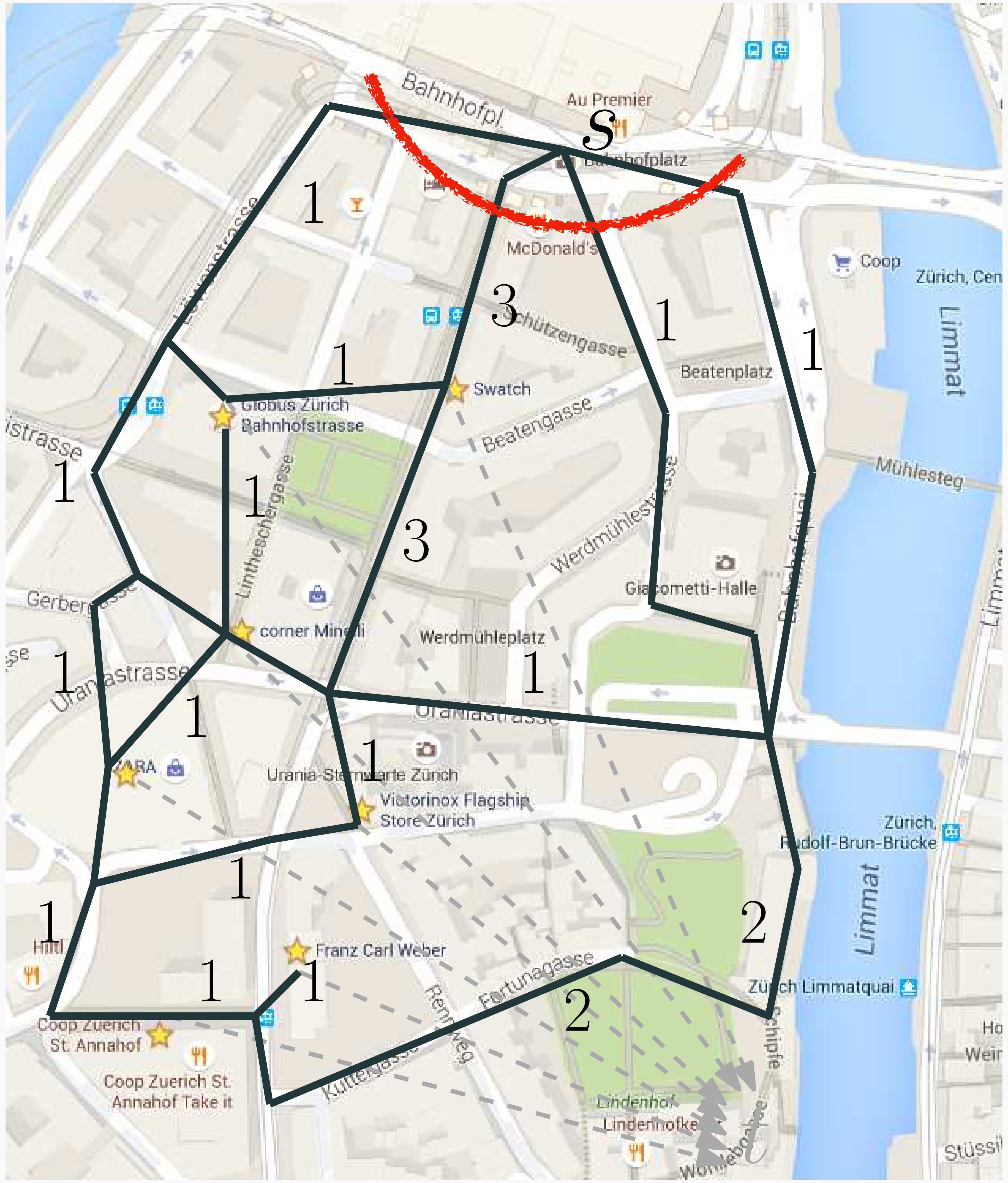
Condition: Use each road on at most one trip.

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

Unrealistic condition!

(There are interesting streets in Zürich.)

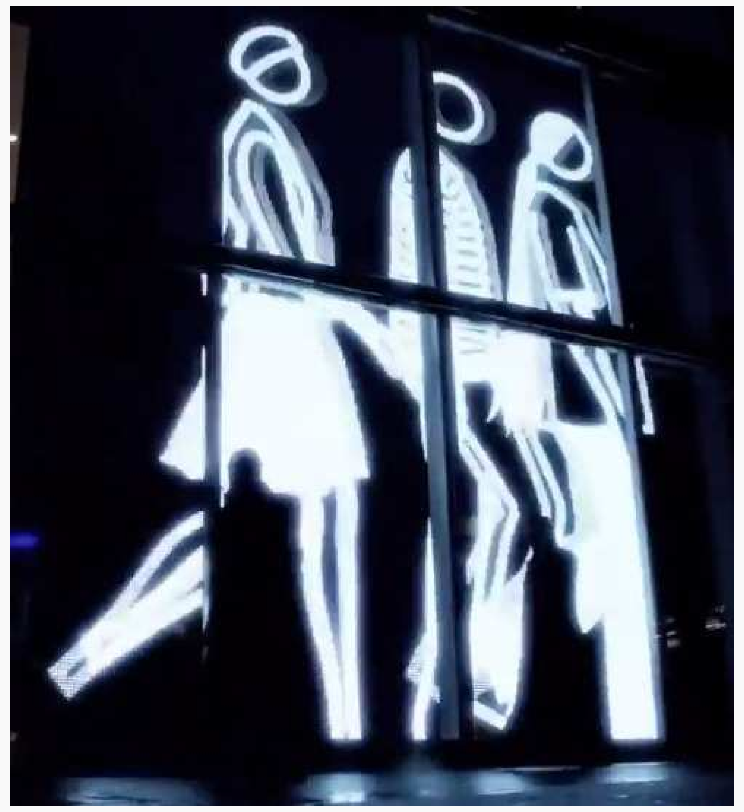
Minimum Cut: Shopping Trip

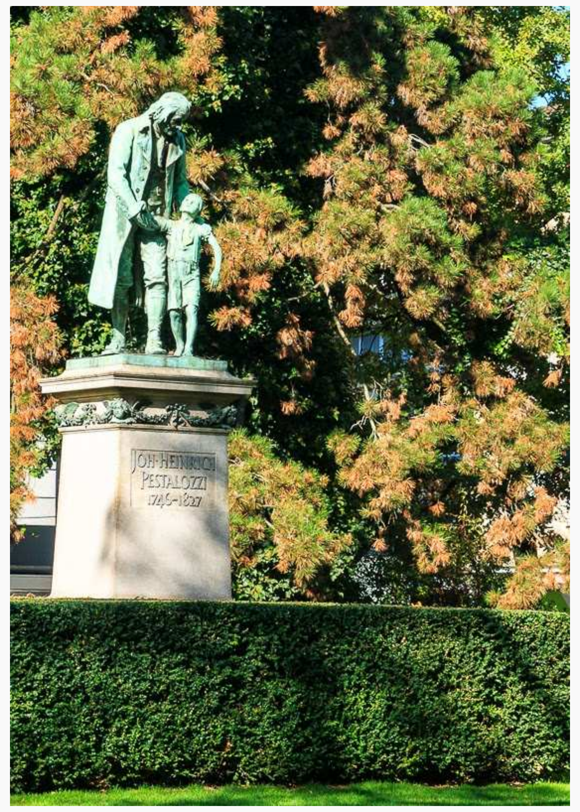


Start from HB:

- 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

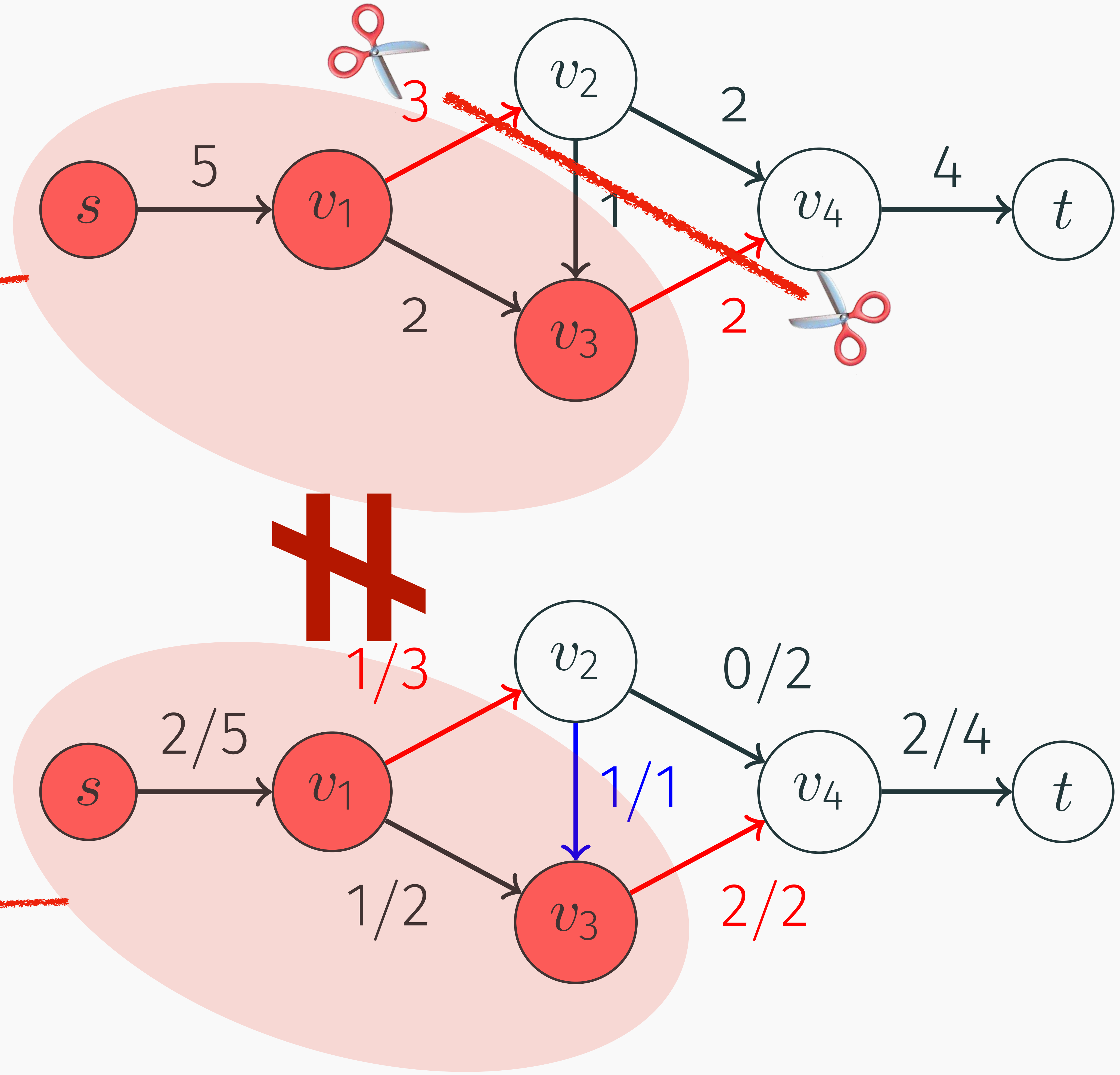
$G = (V, E, s, t)$ a flow network. $S \subset V$ s.t. $s \in S, t \in V \setminus S$, e.g. $\boldsymbol{S = \{s, v_1, v_3\}}$.

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

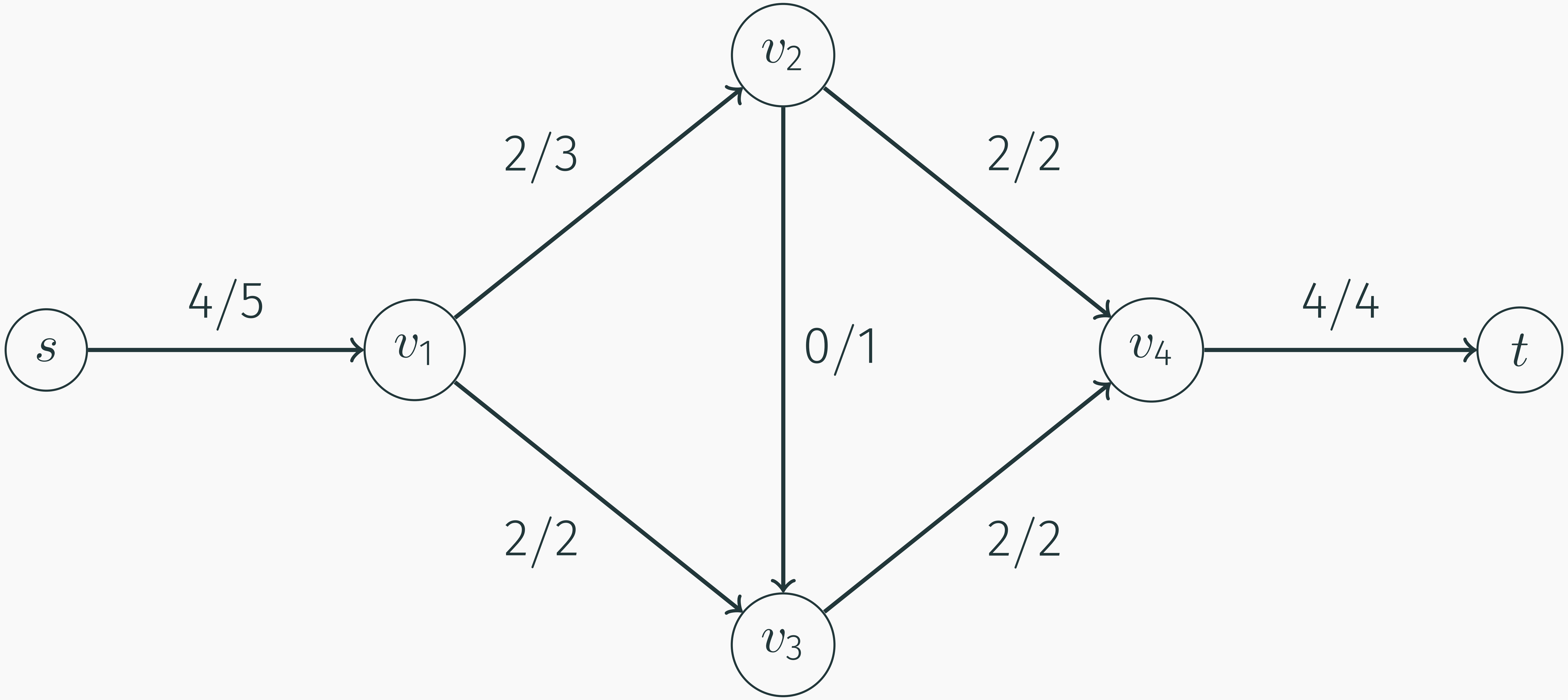
$$|f| = \min_{S:\ s \in S, t \notin S} \mathit{cap}(S, V \setminus S).$$

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 cap$(e) - f(e)$, if 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.

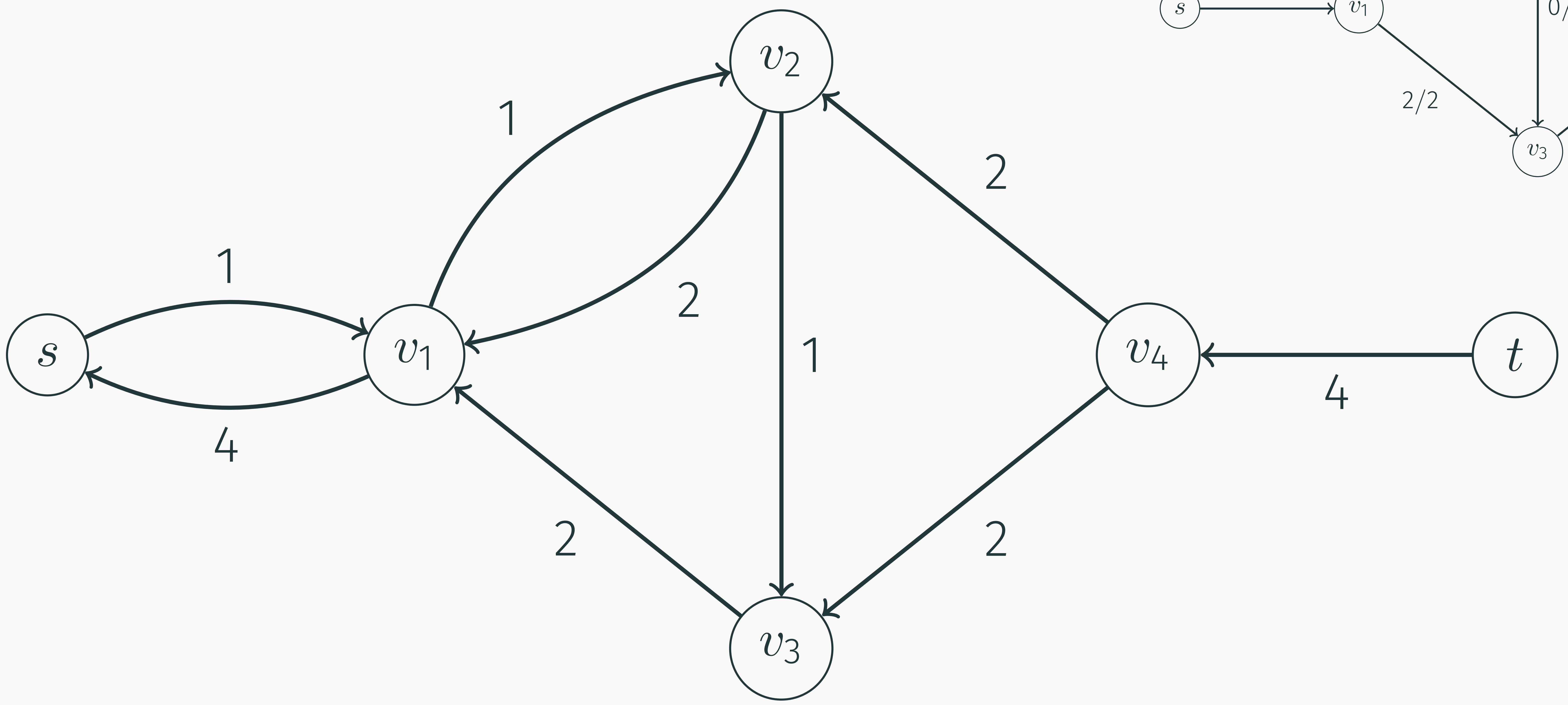
Minimum Cut: Example

Graph G and a maximum flow f.

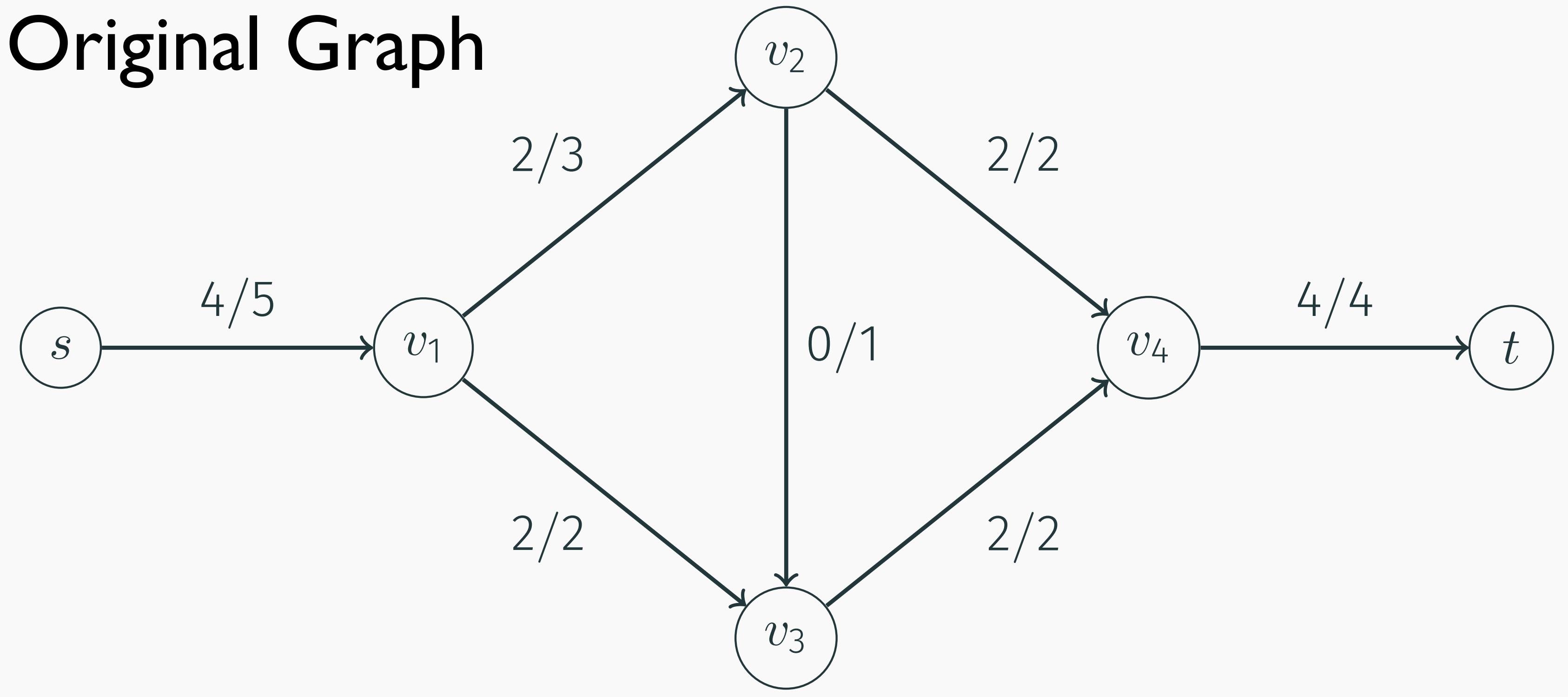


Minimum Cut: Example

Residual graph G_f.

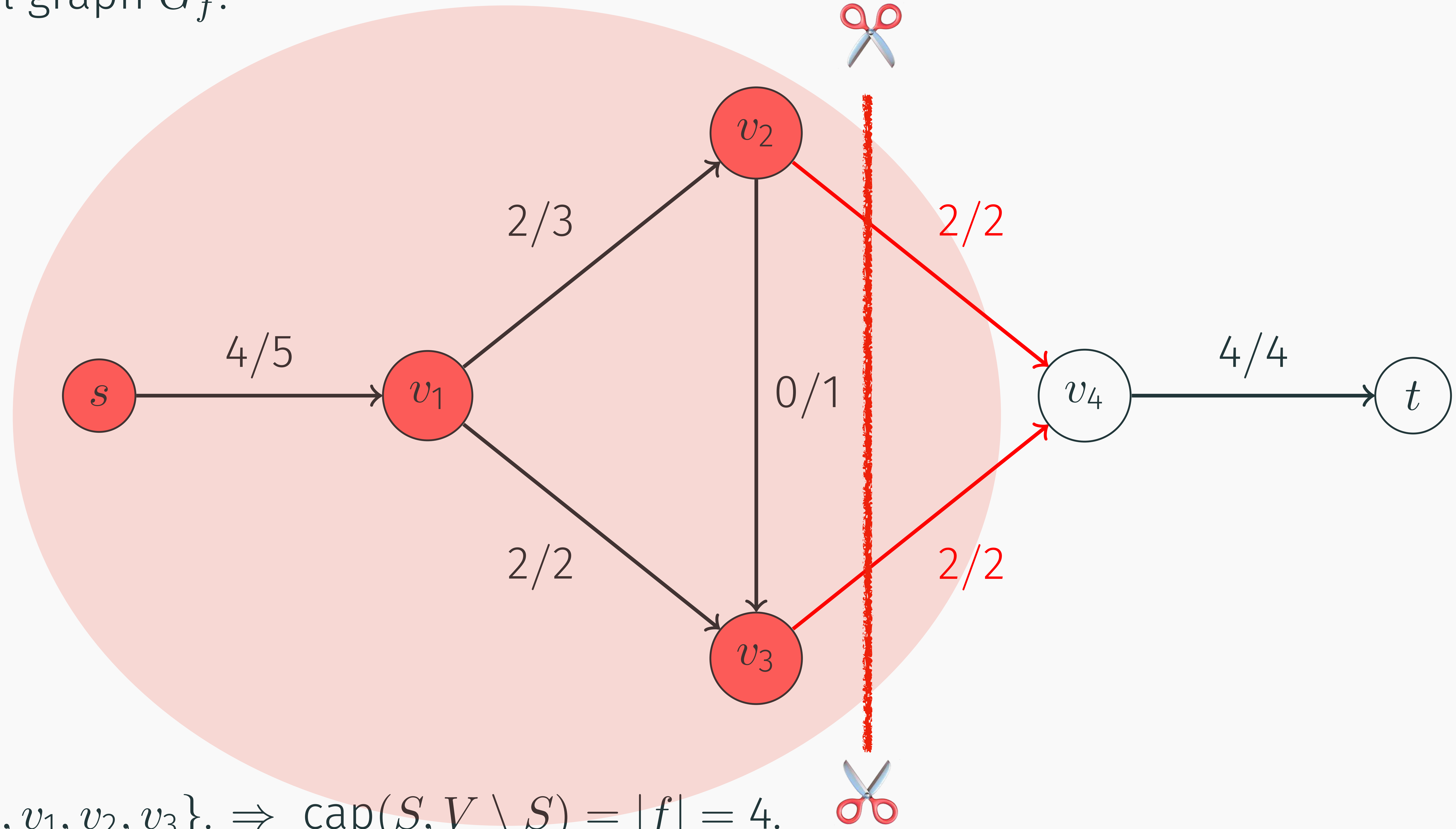


Original Graph



Minimum Cut: Example

Residual graph G_f.



$|S| = \{s, v_1, v_2, v_3\}$. $\Rightarrow$ cap$(S, V \setminus S) = |f| = 4$.

Minimum Cut: Code

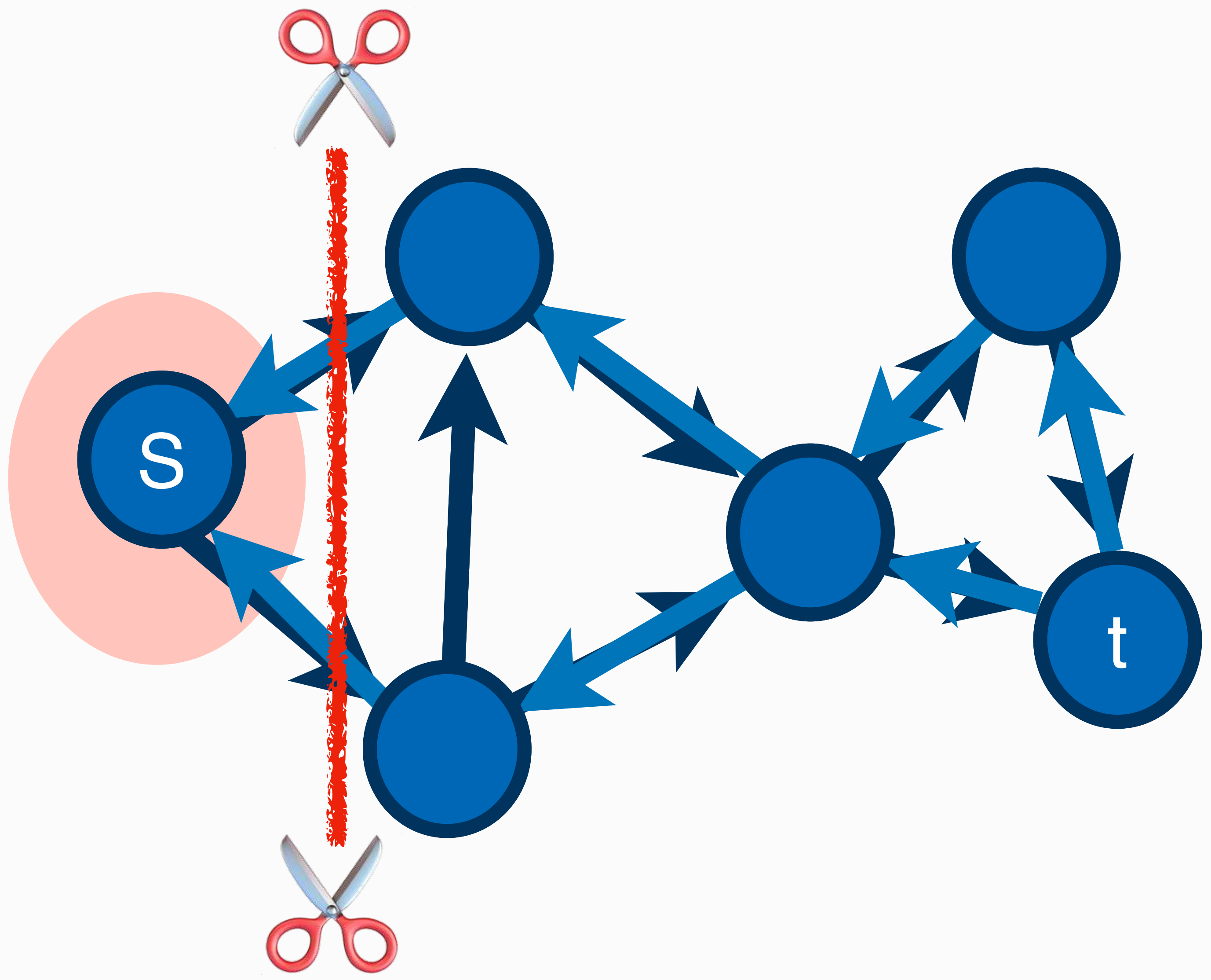
Example code: BFS on the residual graph G_f. $\rightarrow$ `bgl_residual_bfs.cpp`

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

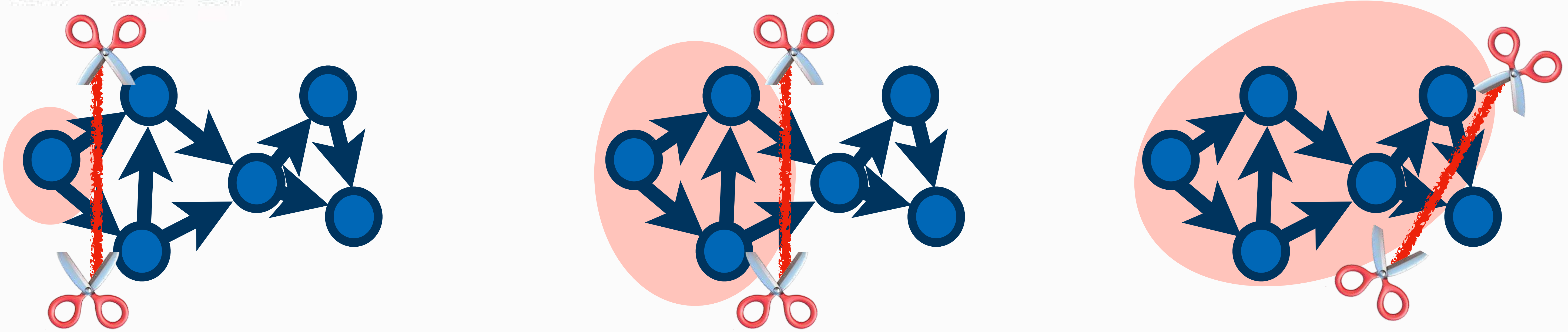
Minimum Cut: Algorithm

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.

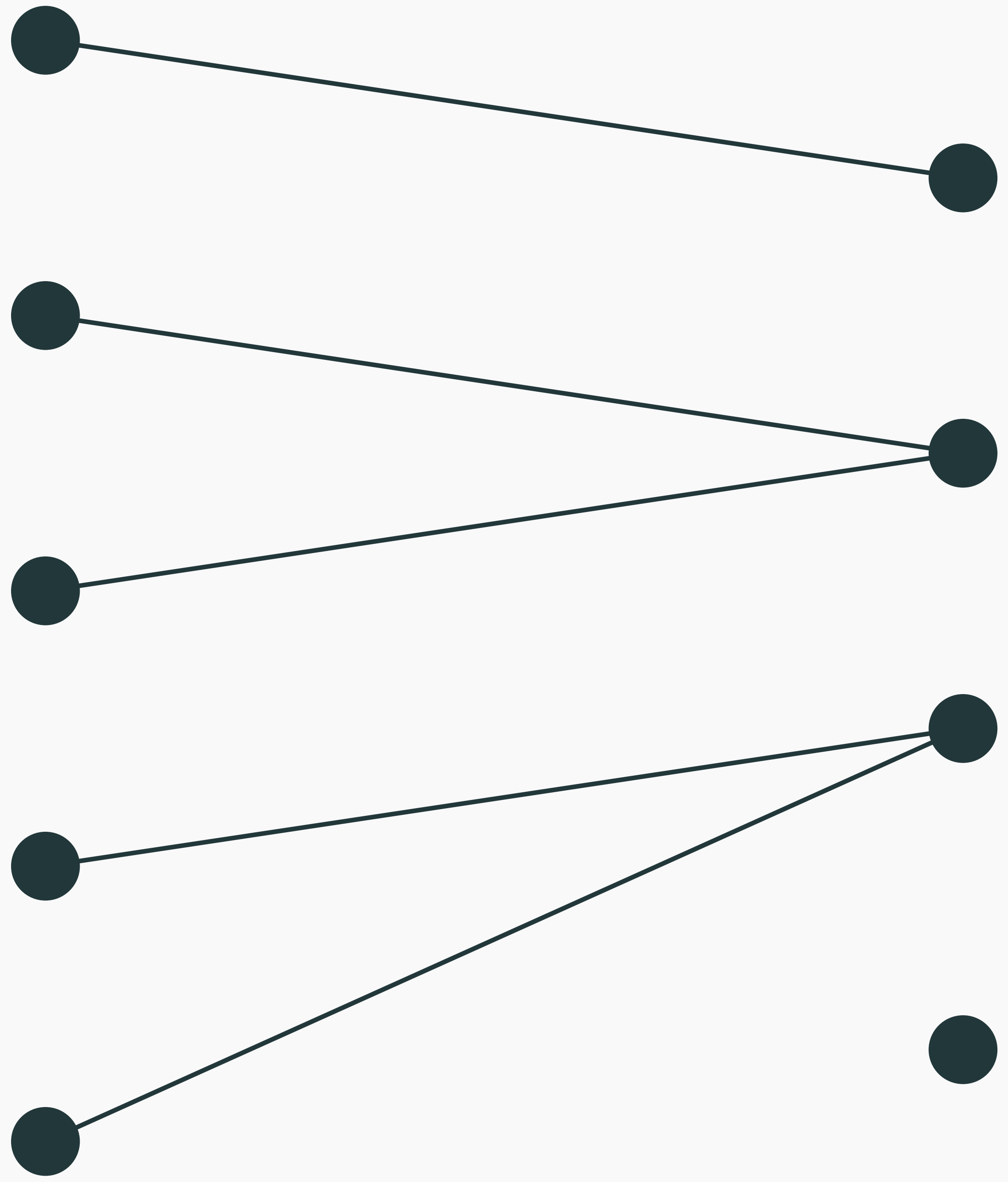


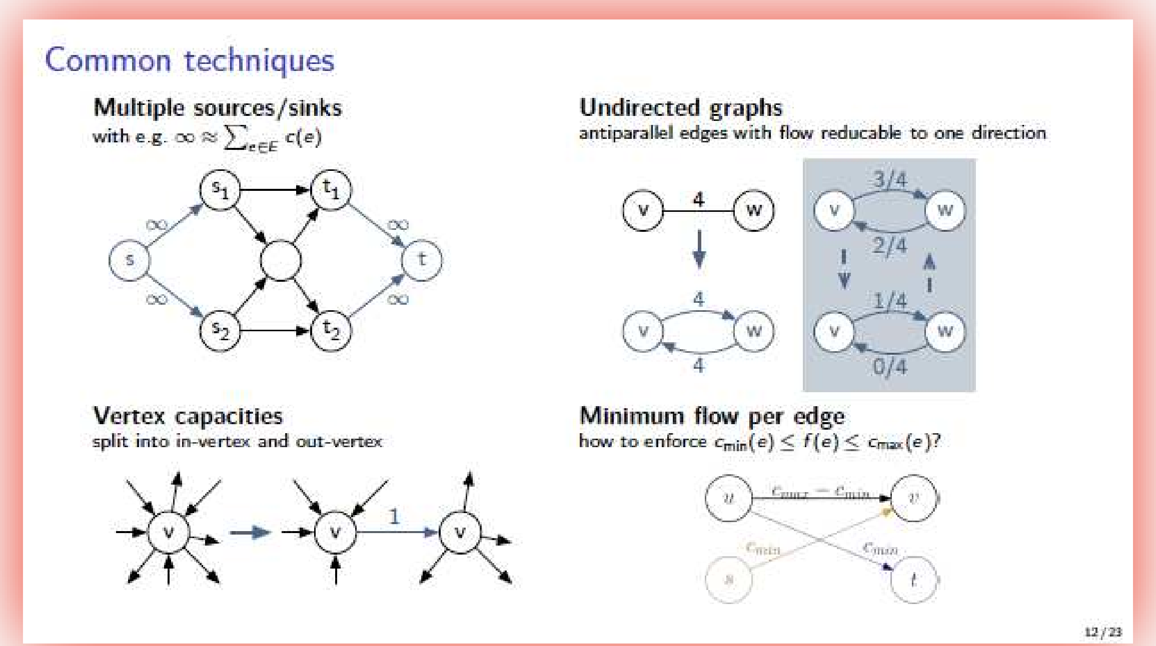
Bipartite Matchings

Maximum Matchings: Bipartite Graphs

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

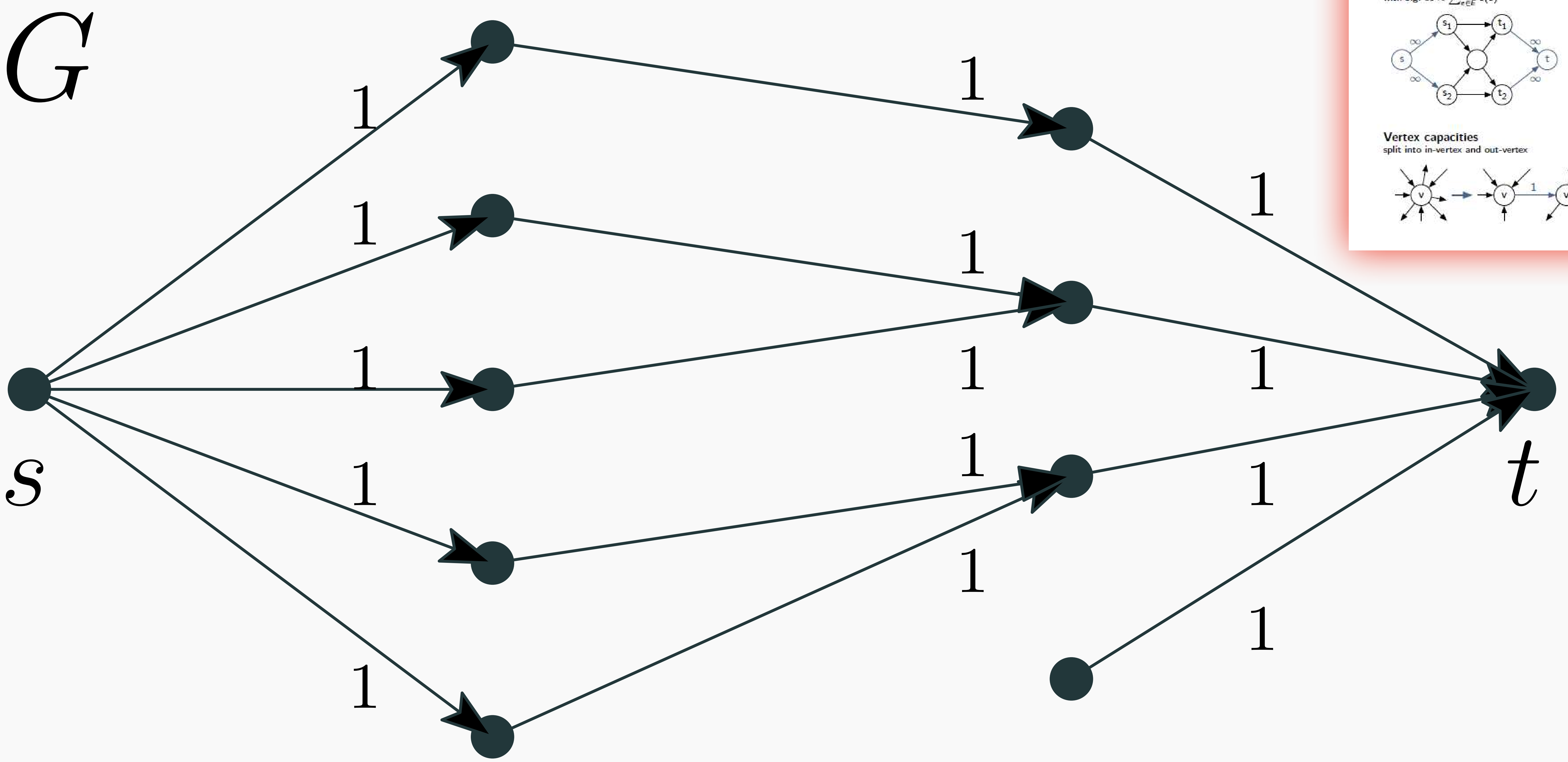
Flow formulation through circulation / vertex capacities / edges-disjoint paths:

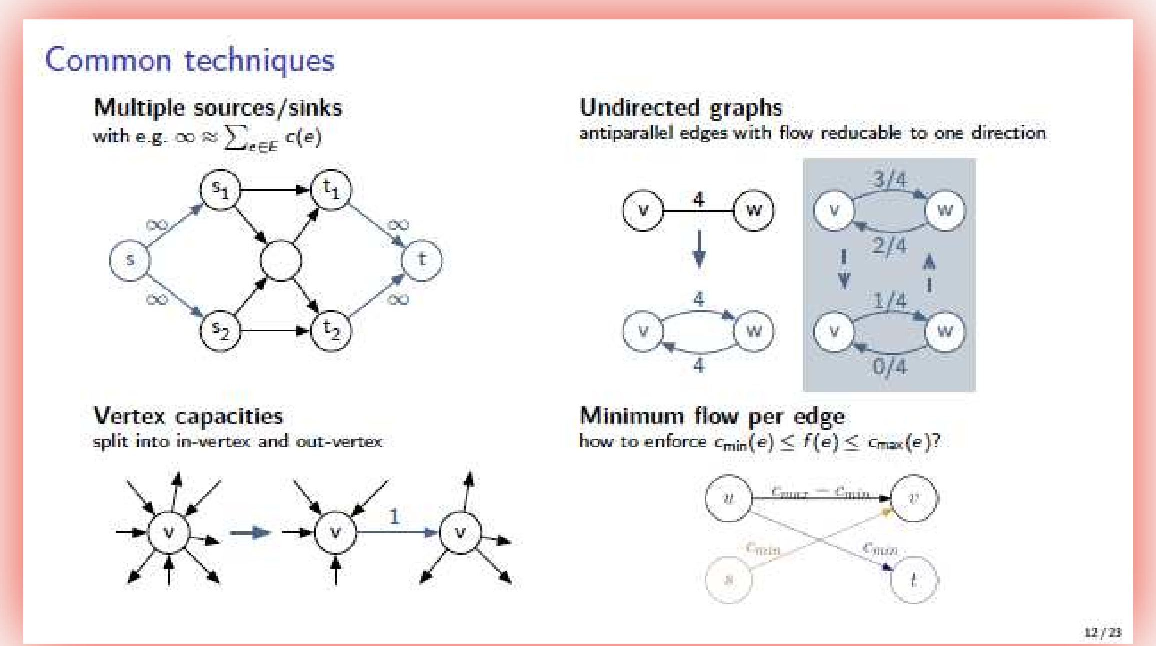




Maximum Matching = pick as many non-adjacent edges as possible

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

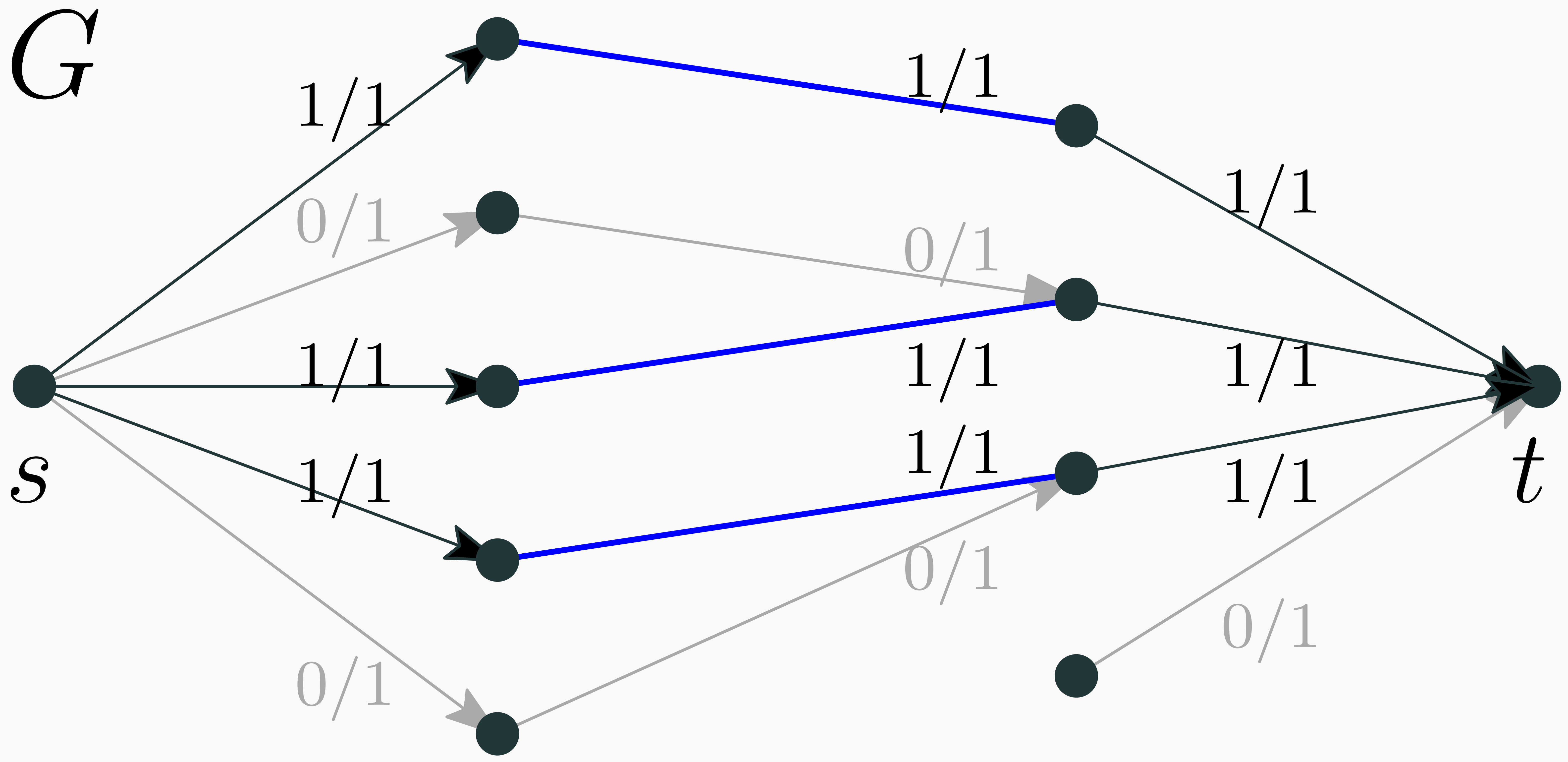




Maximum Matchings: Bipartite Graphs

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

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

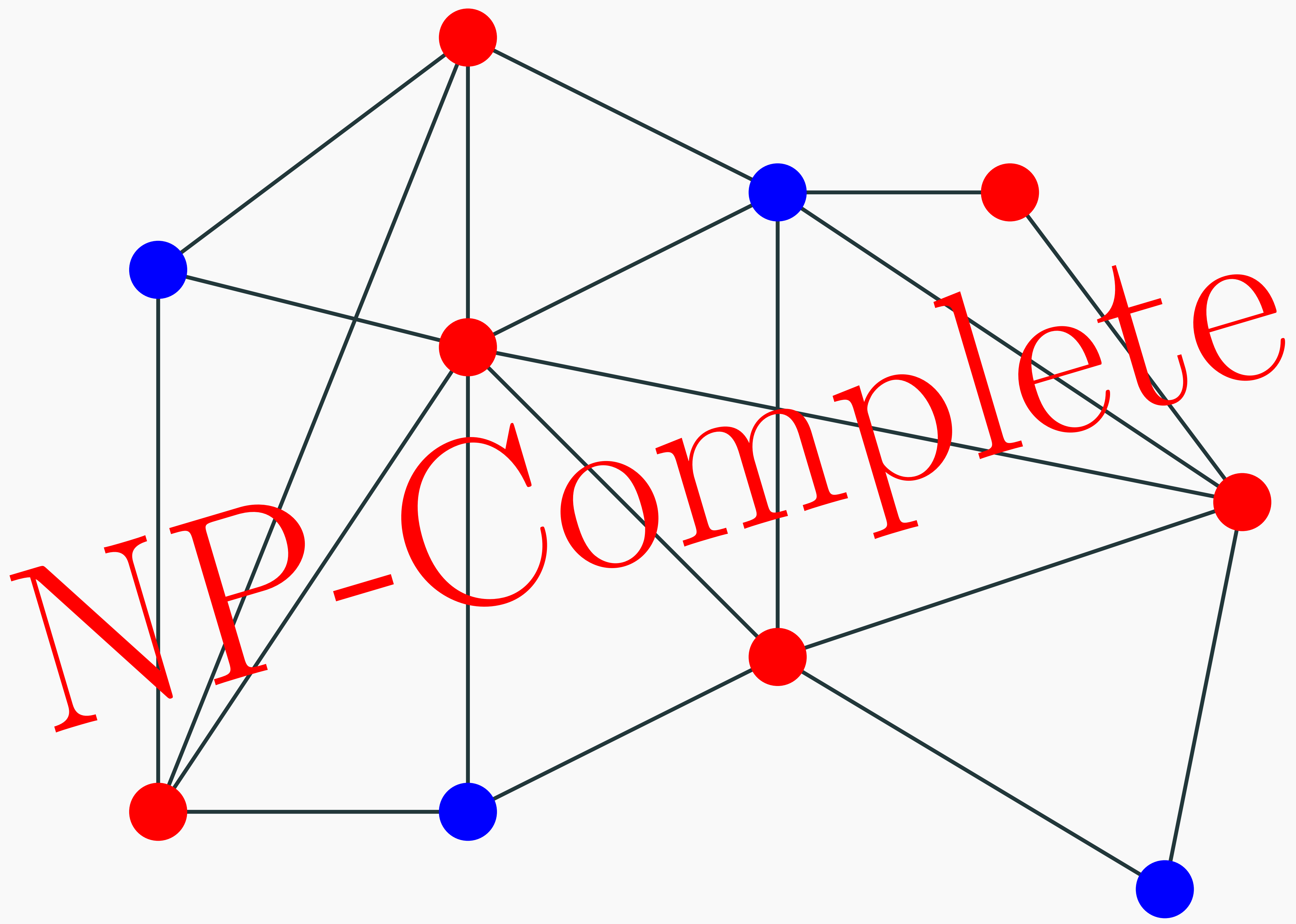


Vertex Cover and Independent Set: General Graphs

- **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!
  MaxIS $= V \setminus$ 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

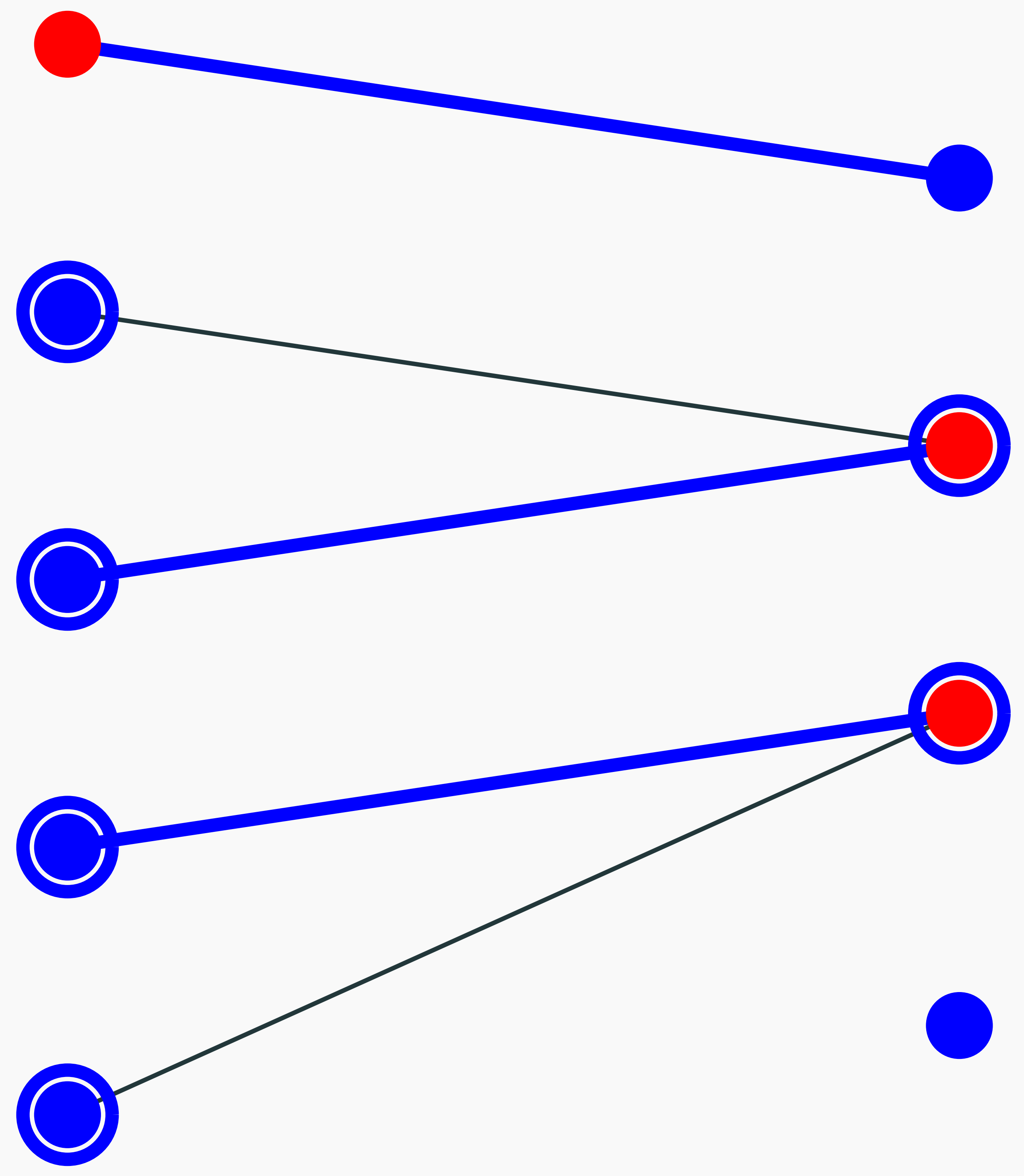
Theorem (König: MinVC and MaxIS is simpler on bipartite graphs!)

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.

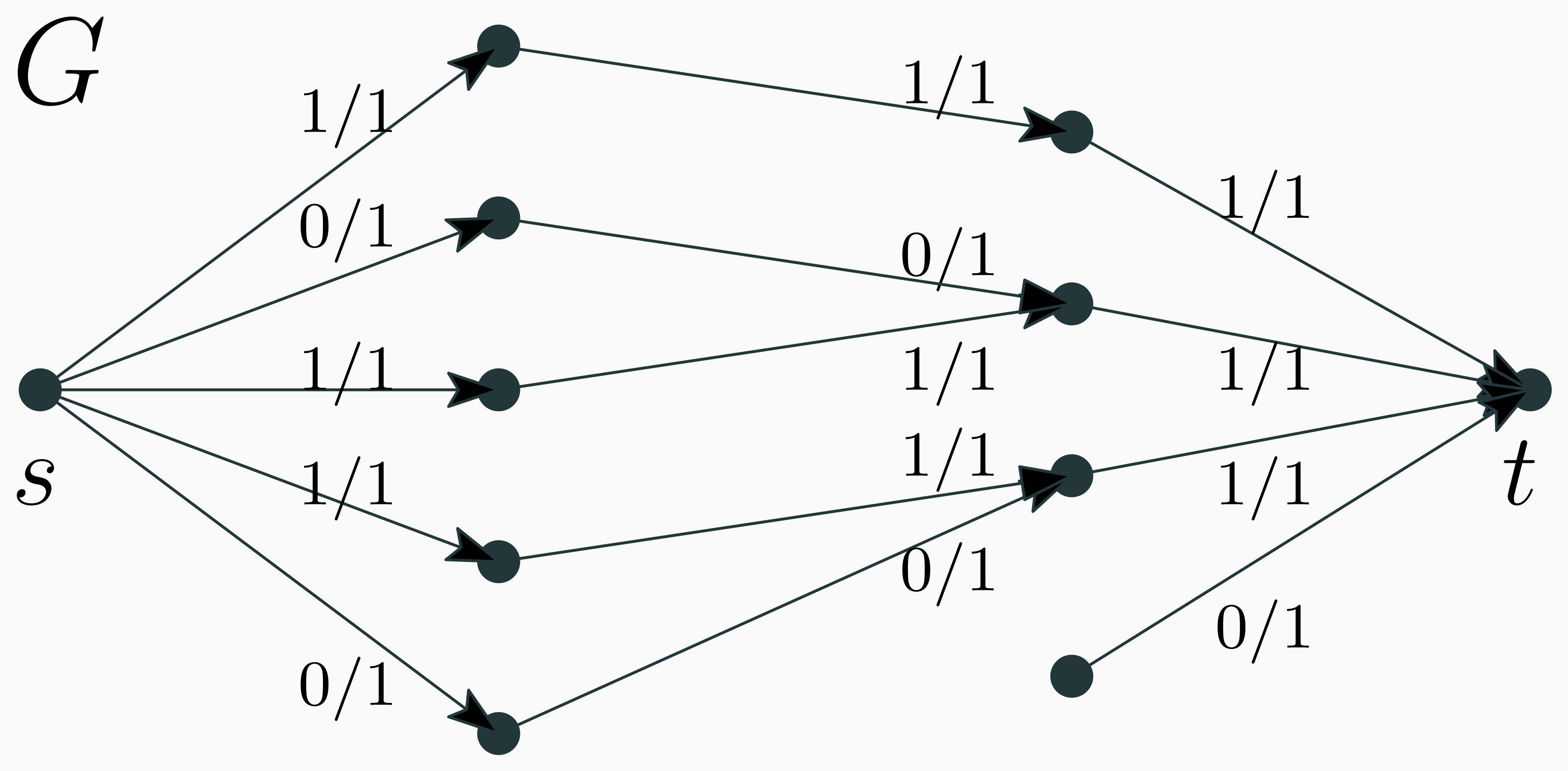


Careful! Step 2 can take several rounds.

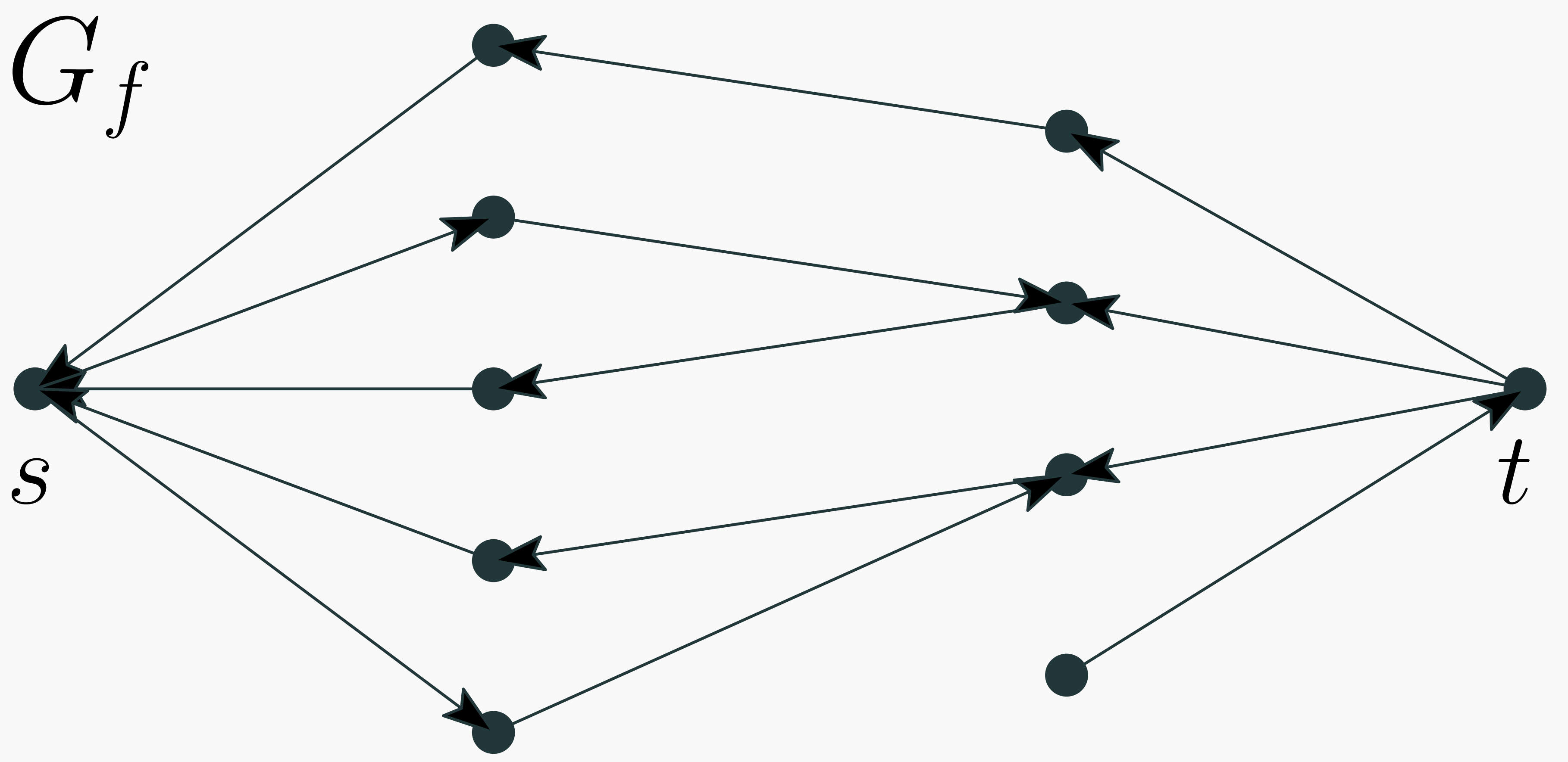
Easy Implementation?

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

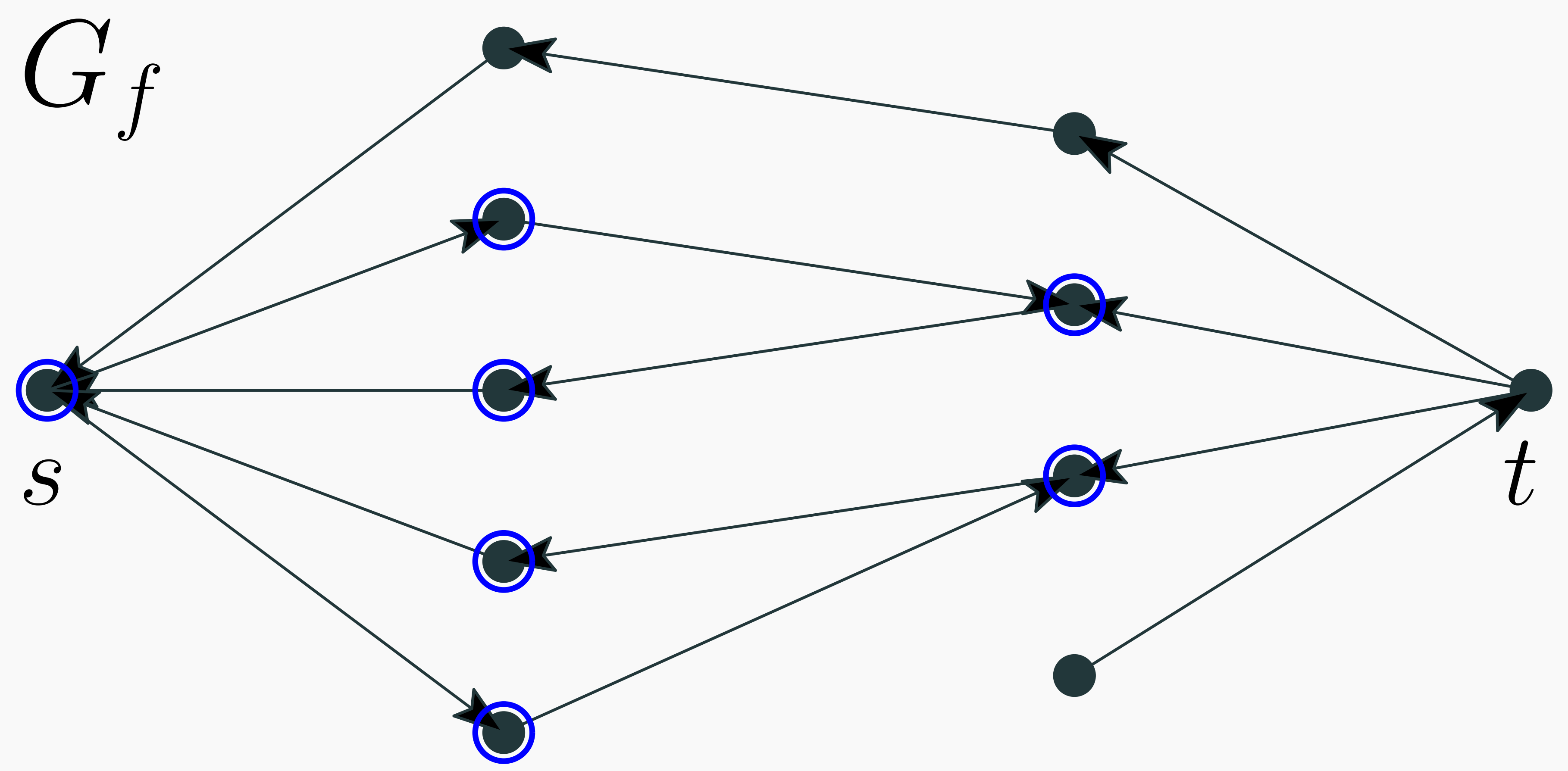
1) Formulate and compute the flow:



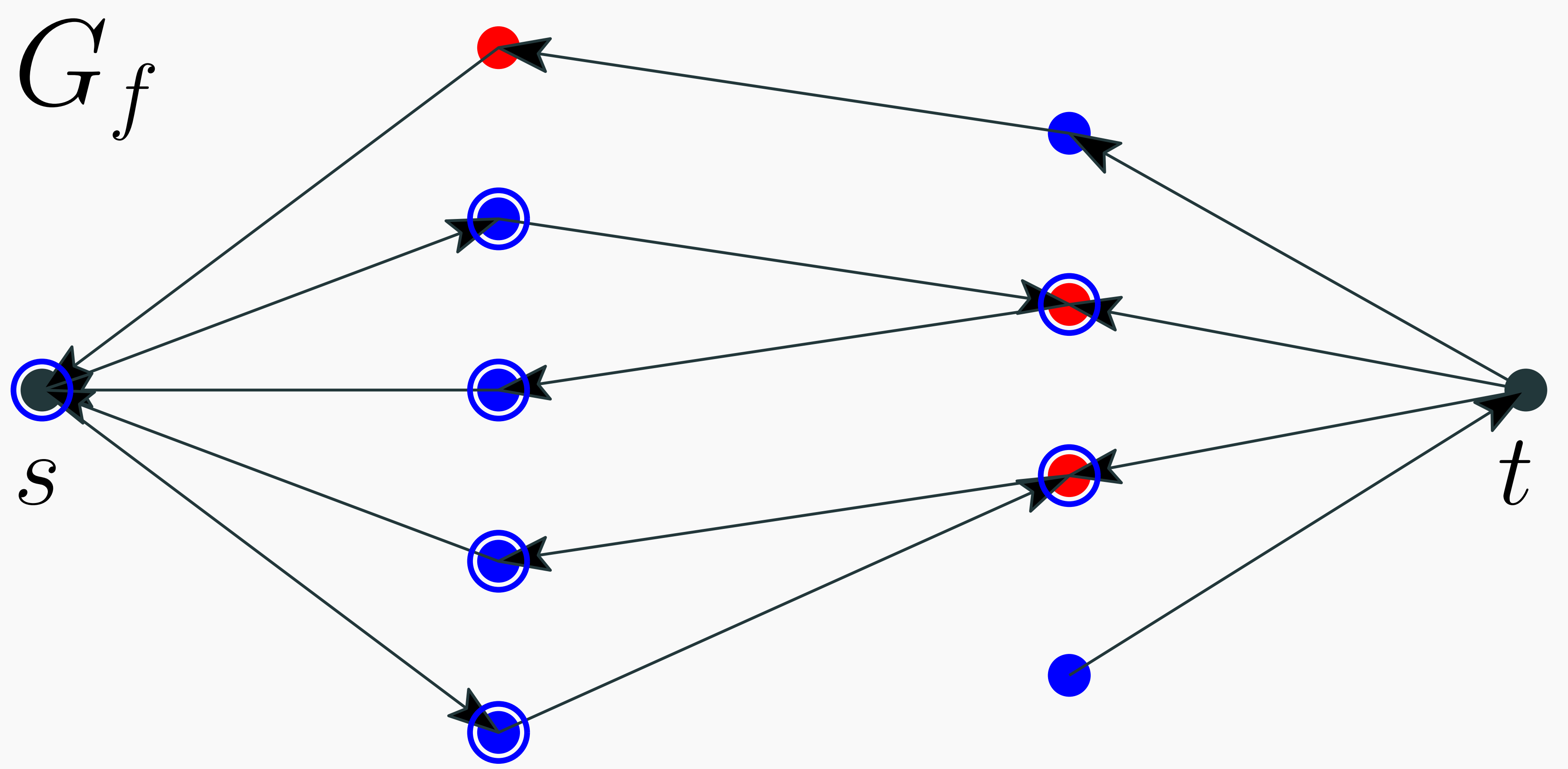
2) Compute the residual graph G_f:



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?

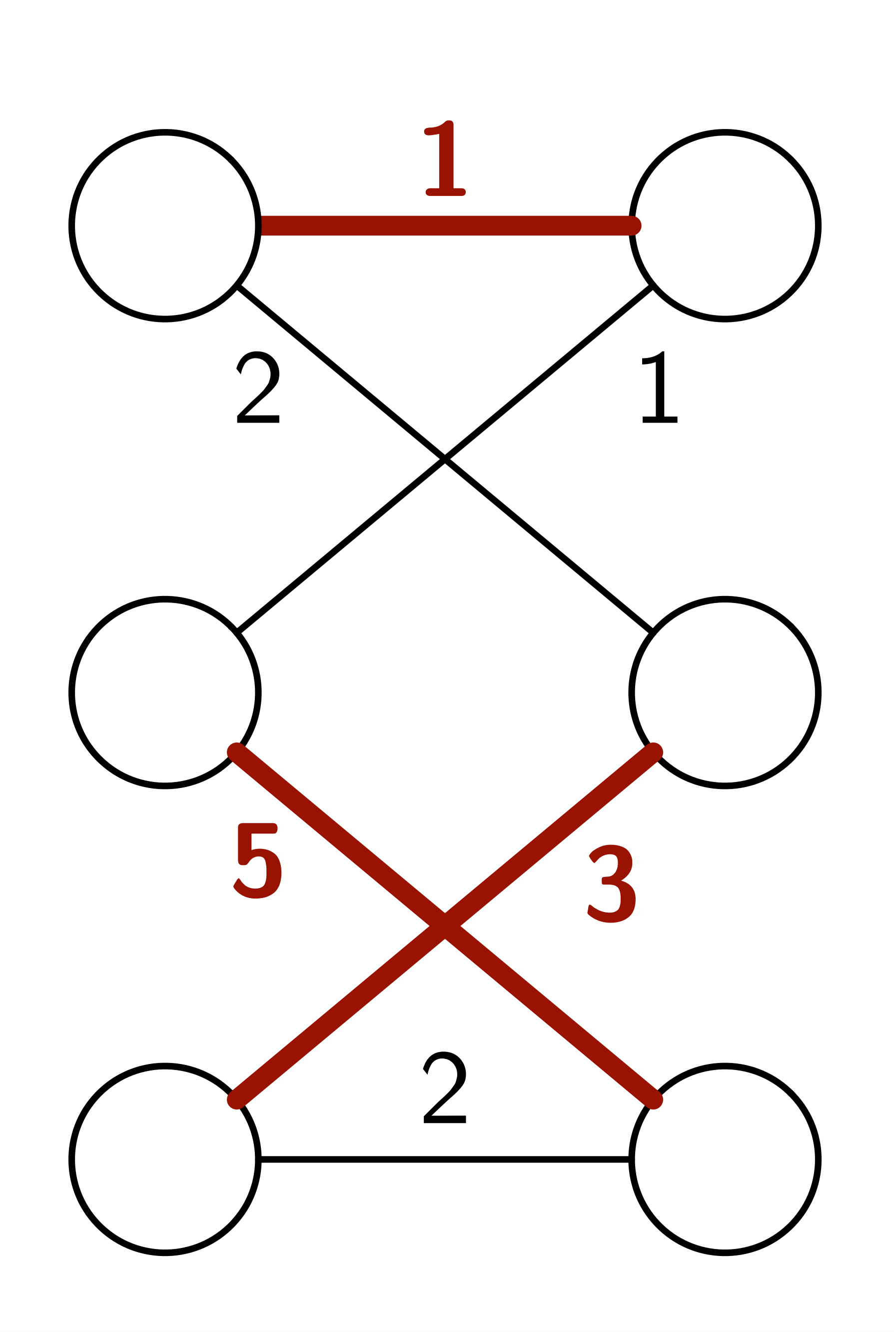
What if the edges in a matching also have costs associated?

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

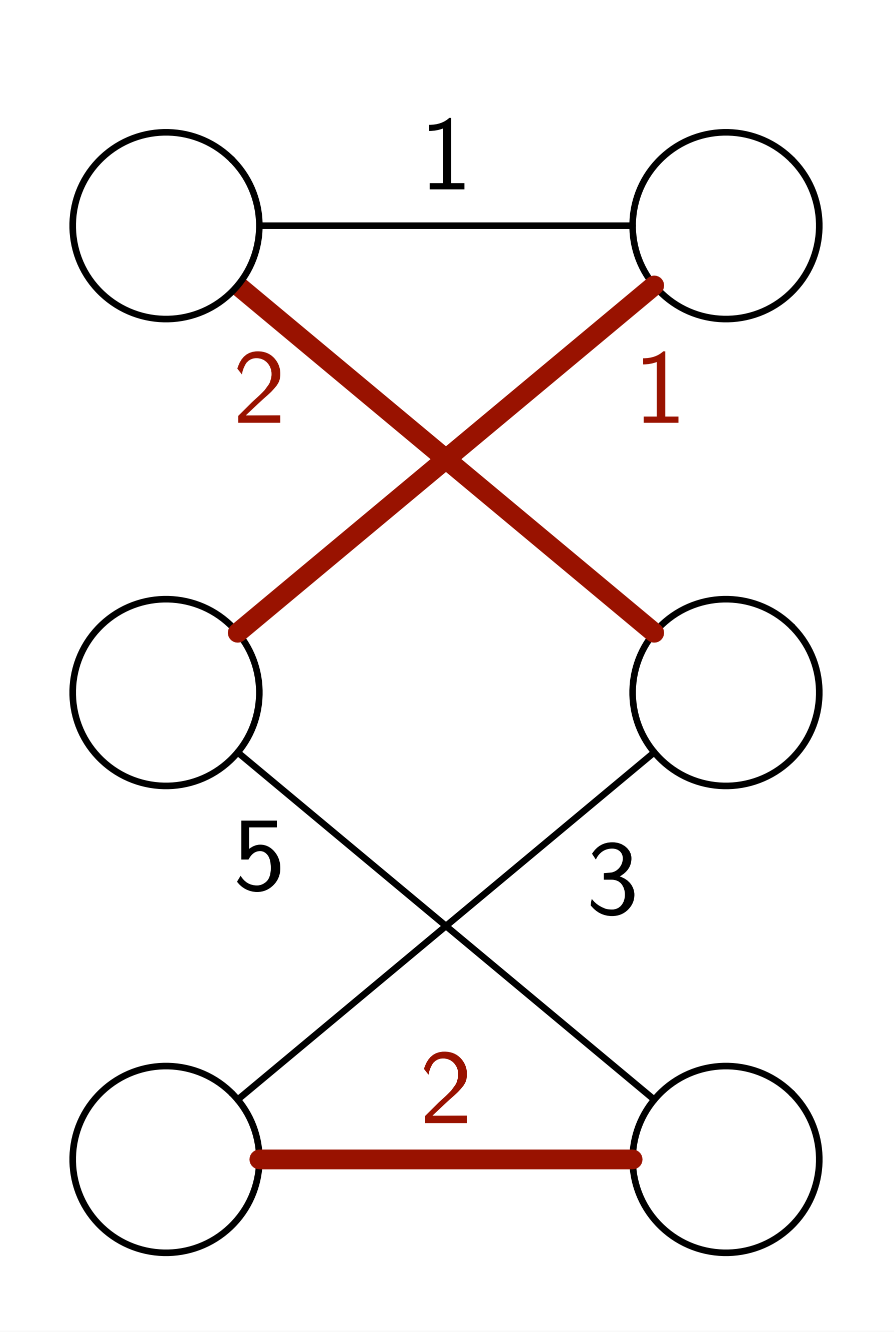
We search for the cheapest among all maximum matchings.

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

More General Model: Minimum Cost Maximum Flow

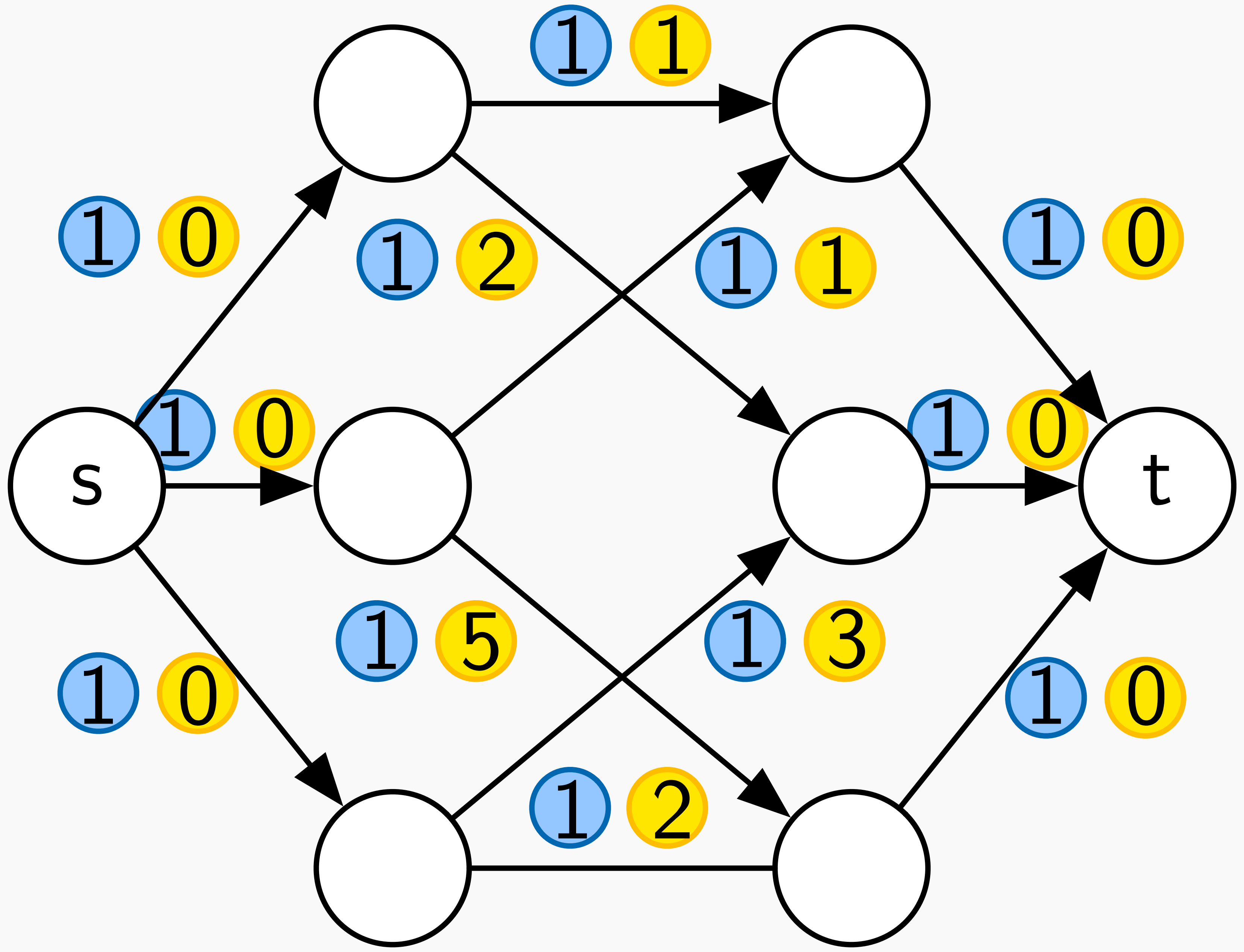
We extend the network flow problem by allowing edge costs that occur per unit of flow.

Input: A flow network consisting of

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

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.



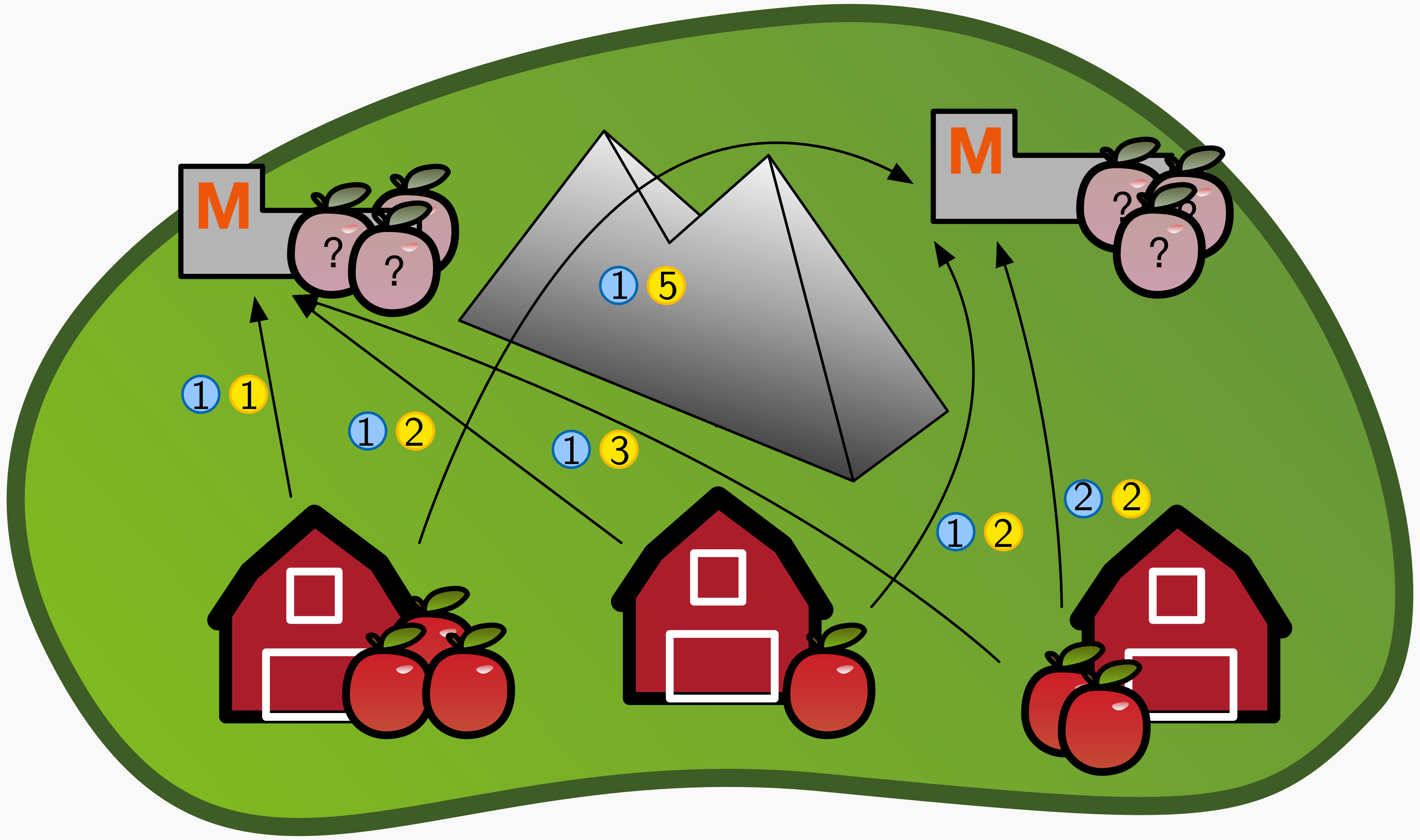
Legend: capacity vs cost

Example: Fruit Delivery

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

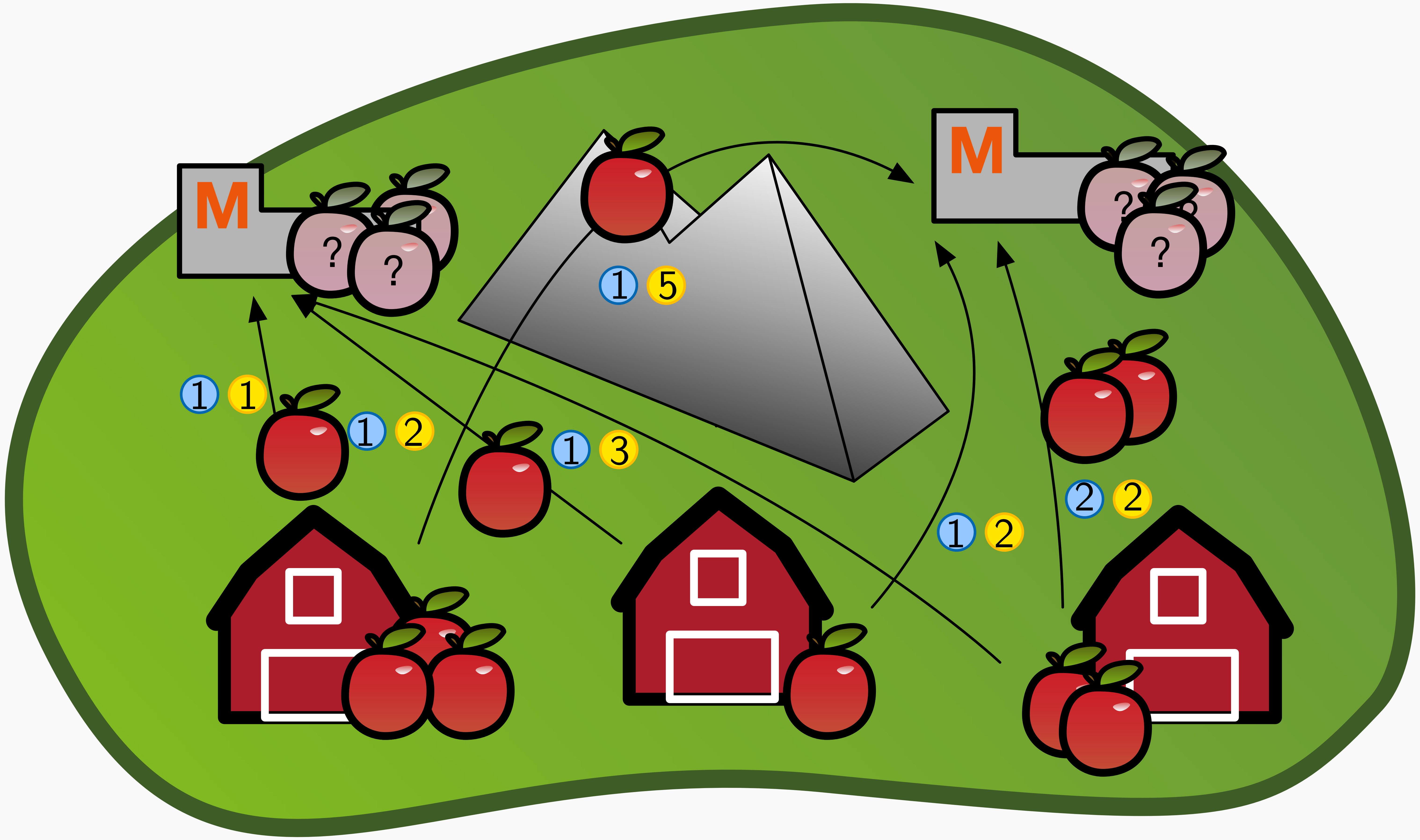
They know all important parameters;

- 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).

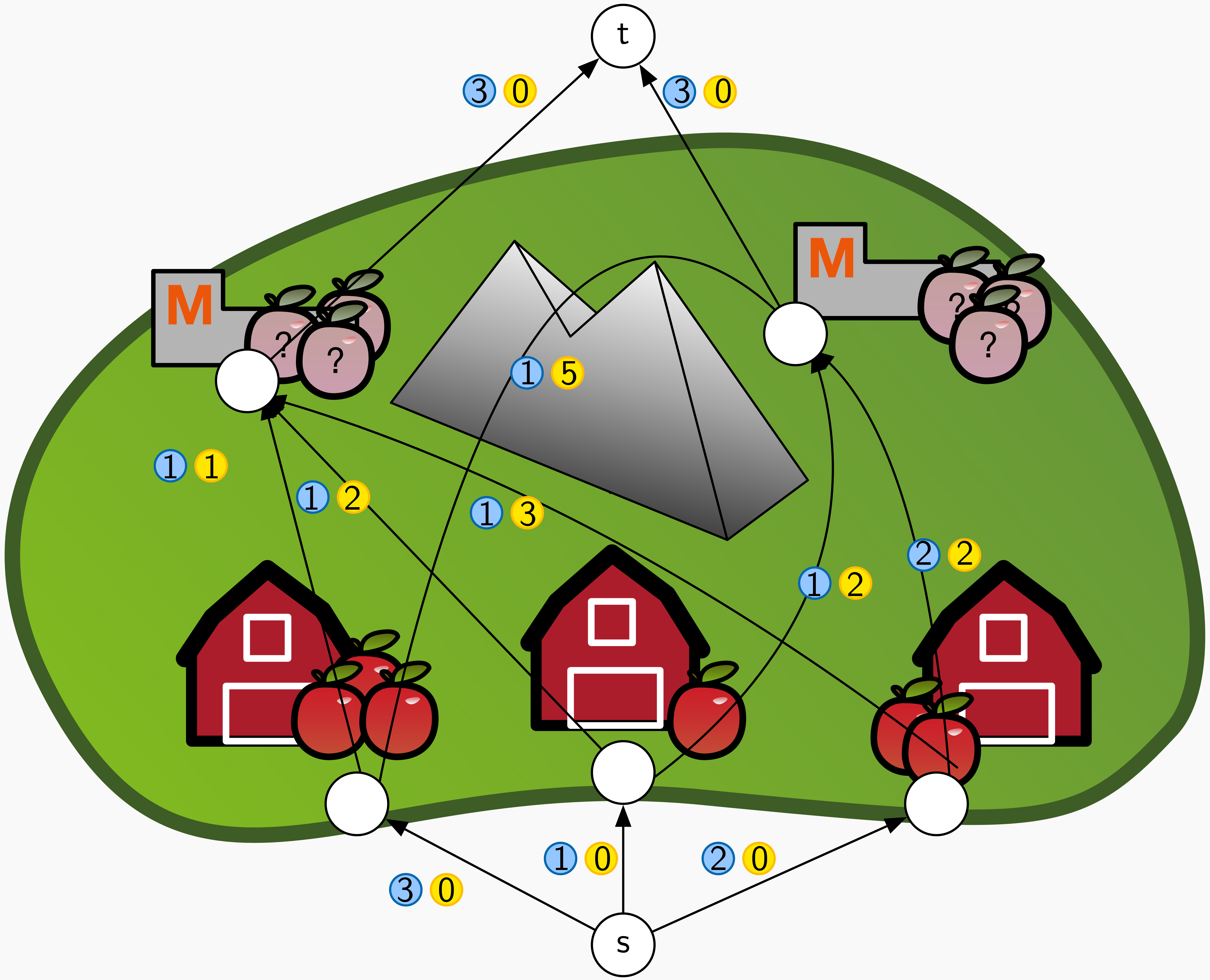
Example: Fruit Delivery



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

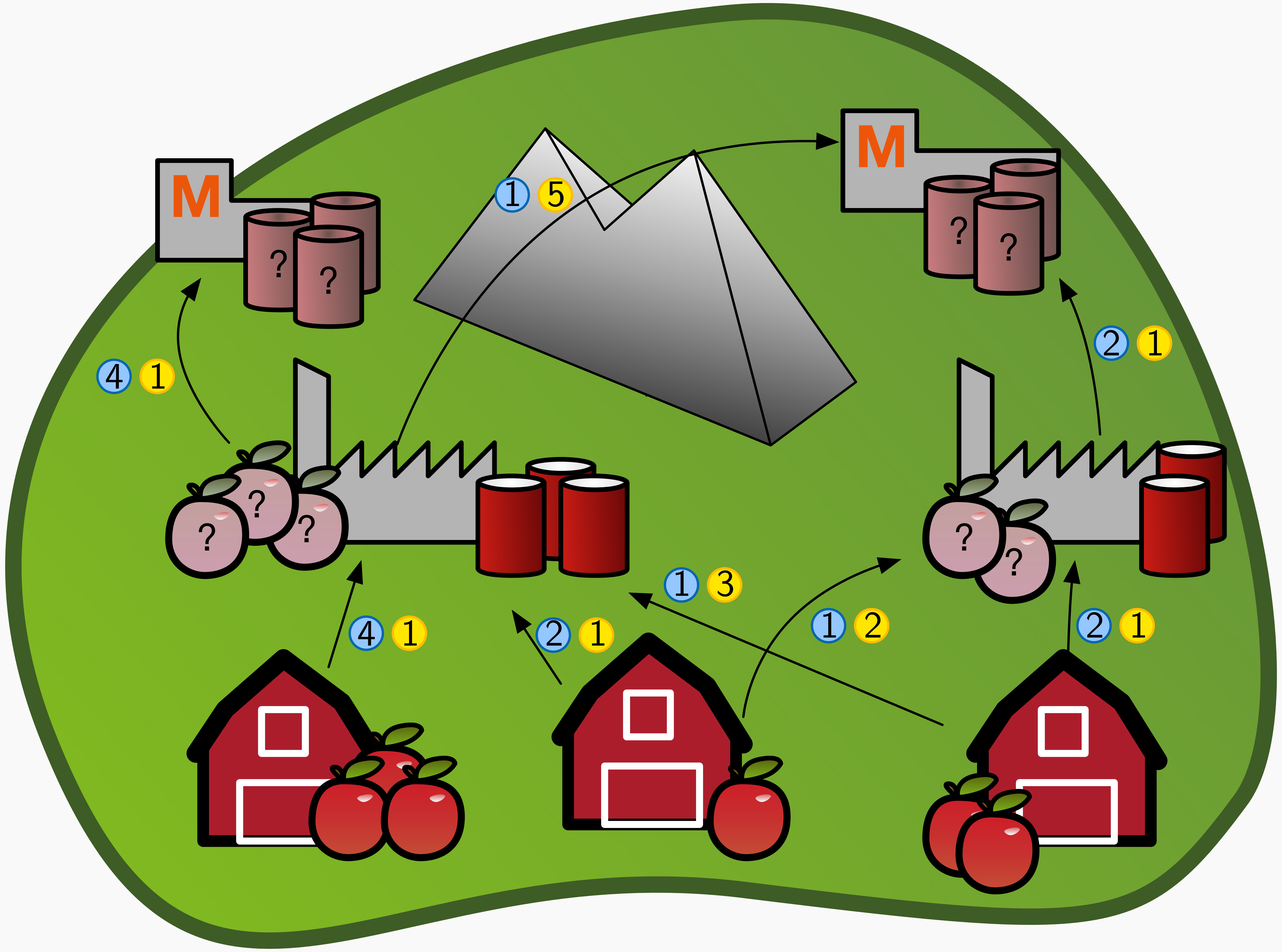
$$\text{Cost: } 1 \cdot 1 + 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 2 = 12$$

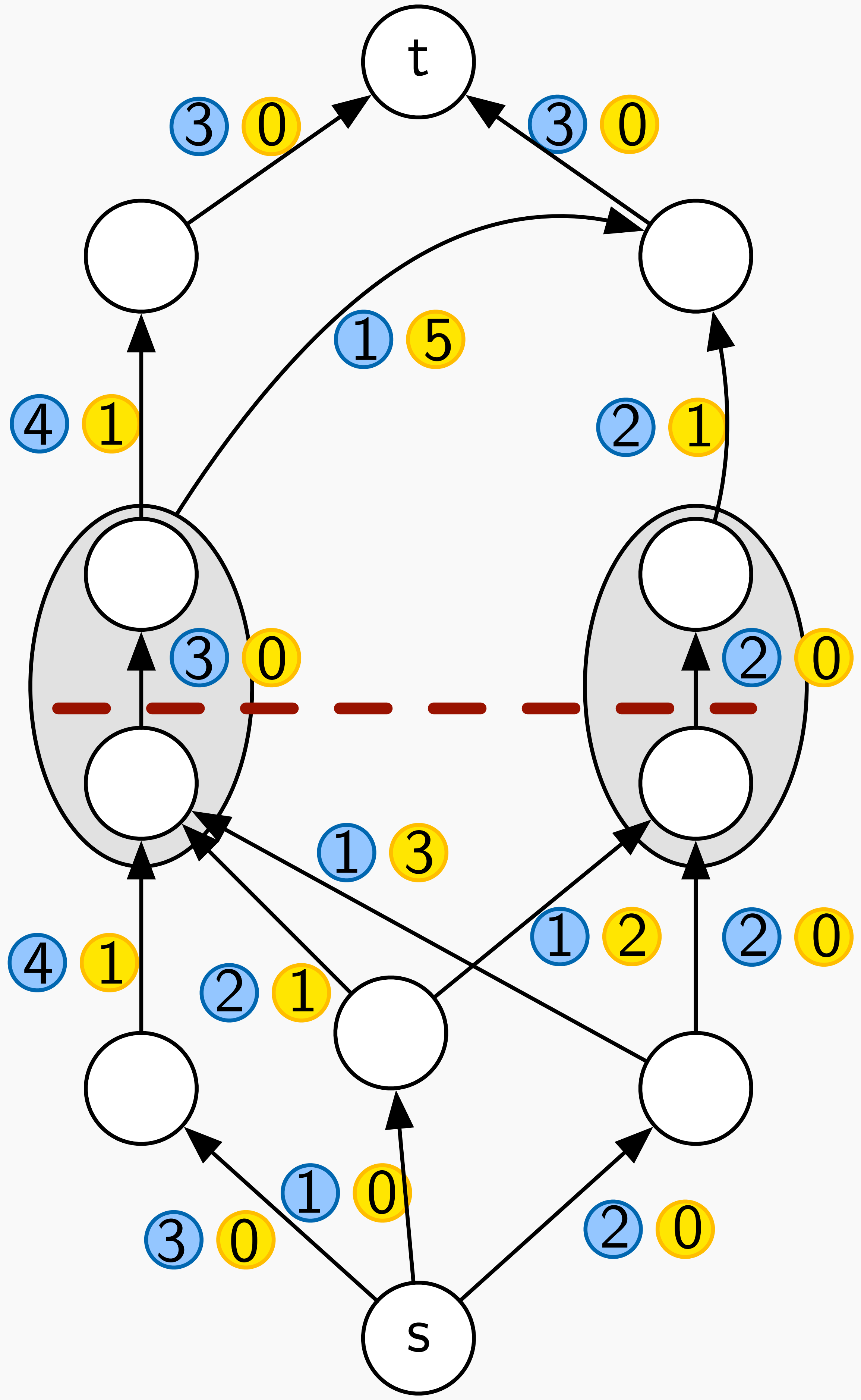
Example: Fruit Delivery



Extended Example: Canned Fruit Delivery

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



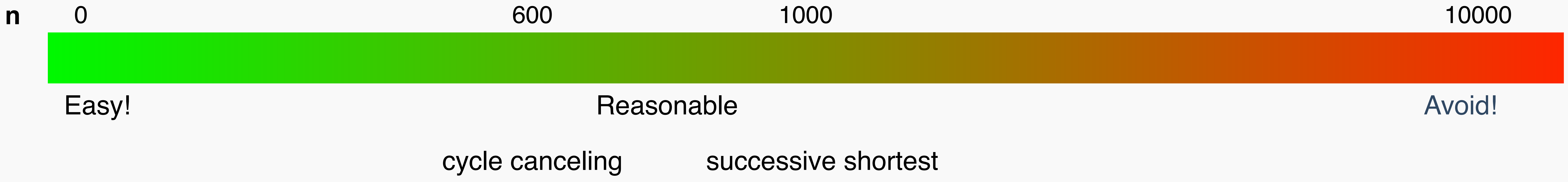


Min Cost Max Flow with BGL

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≈n, |C|, |f| << 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:

```
// Graph Type with nested interior edge properties for Cost Flow Algorithms
typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> traits;
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
    boost::property<boost::edge_capacity_t, long,
        boost::property<boost::edge_residual_capacity_t, long,
            boost::property<boost::edge_reverse_t, traits::edge_descriptor,
                boost::property <boost::edge_weight_t, long> > > > > graph; // new!

typedef boost::graph_traits<graph>::edge_descriptor            edge_desc;
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:

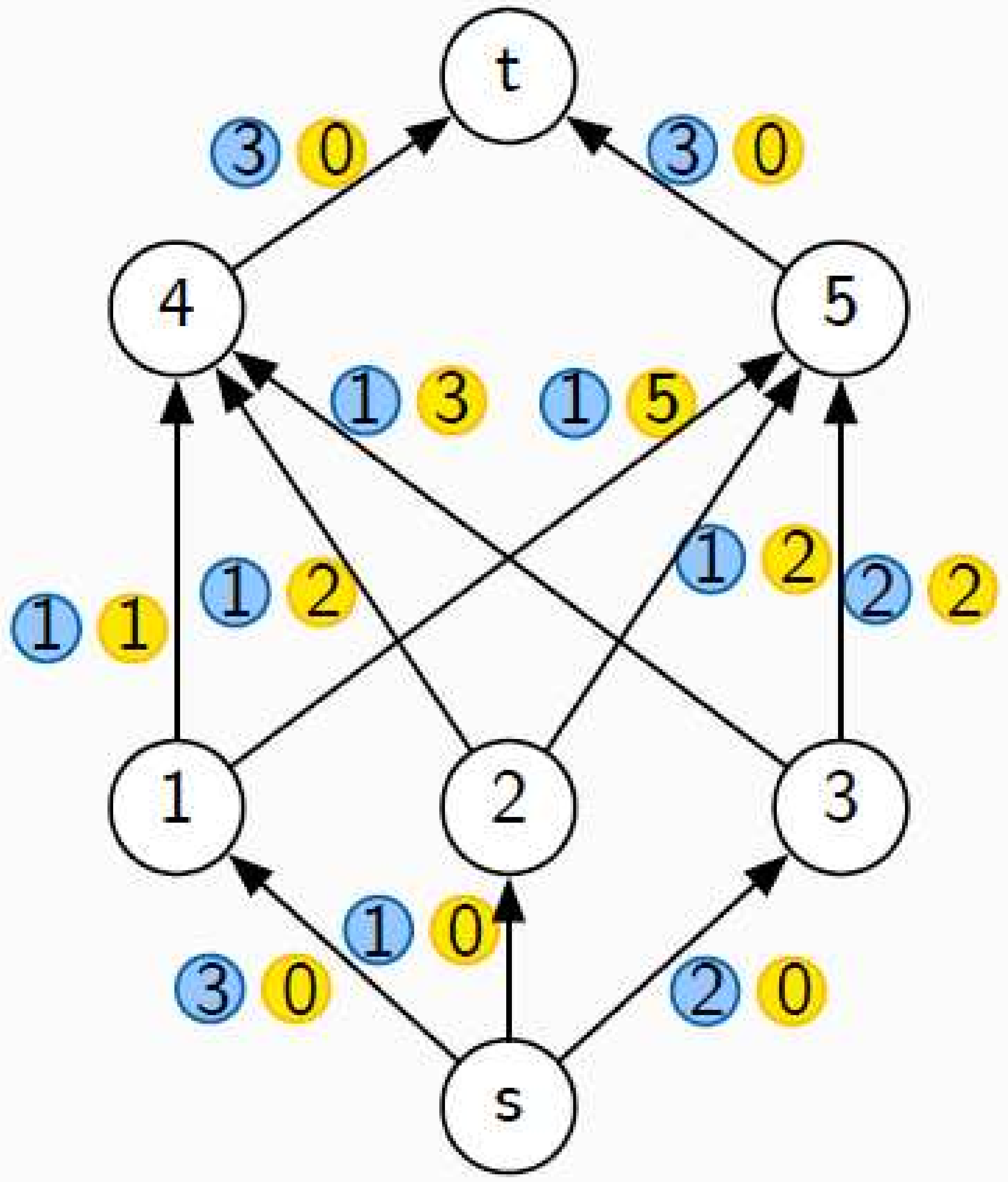
```
class edge_adder {
  graph &G;

 public:
  explicit edge_adder(graph &G) : G(G) {}
  void add_edge(int from, int to, long capacity, long cost) {
    auto c_map = boost::get(boost::edge_capacity, G);
    auto r_map = boost::get(boost::edge_reverse, G);
    auto w_map = boost::get(boost::edge_weight, G); // new!
    const edge_desc e = boost::add_edge(from, to, G).first;
    const edge_desc rev_e = boost::add_edge(to, from, G).first;
    c_map[e] = capacity;
    c_map[rev_e] = 0;
    r_map[e] = rev_e;
    r_map[rev_e] = e;
    w_map[e] = cost;    // new assign cost
    w_map[rev_e] = -cost;    // new negative cost
  }
};
```

Building the graph

```
const int N=7;
const int v_source = 0;
const int v_farm1 = 1;
const int v_farm2 = 2;
const int v_farm3 = 3;
const int v_shop1 = 4;
const int v_shop2 = 5;
const int v_target = 6;

// Create graph, edge adder class and propery maps
graph G(N);
edge_adder adder(G);
auto c_map = boost::get(boost::edge_capacity, G);
auto r_map = boost::get(boost::edge_reverse, G);
auto rc_map = boost::get(boost::edge_residual_capacity, G);
```



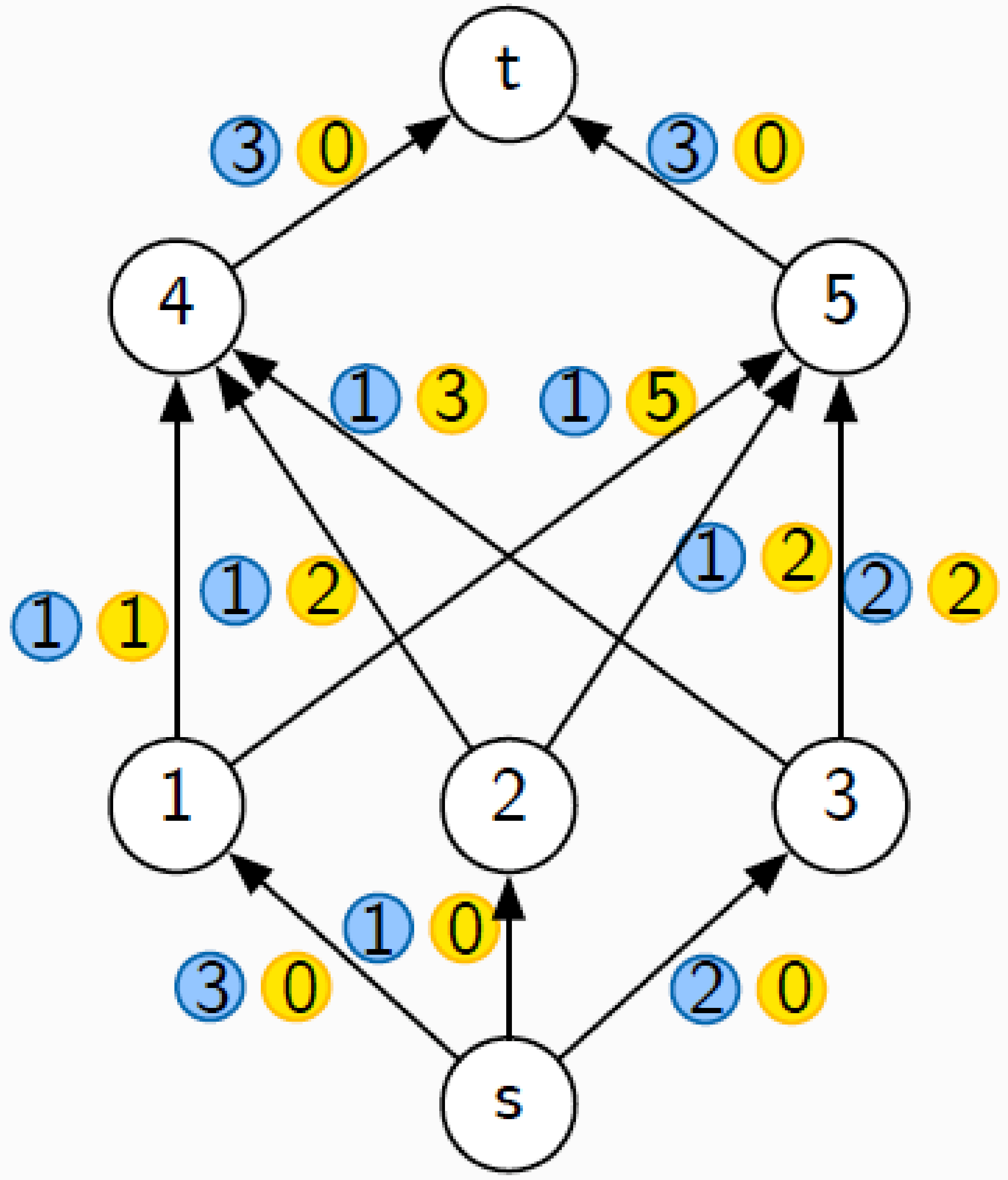
Min Cost Max Flow with BGL

Add the edges:

```
// Add the edges
adder.add_edge(v_source, v_farm1, 3, 0);
adder.add_edge(v_source, v_farm2, 1, 0);
adder.add_edge(v_source, v_farm3, 2, 0);

adder.add_edge(v_farm1, v_shop1, 1, 1);
adder.add_edge(v_farm1, v_shop2, 1, 5);
adder.add_edge(v_farm2, v_shop1, 1, 2);
adder.add_edge(v_farm2, v_shop2, 1, 2);
adder.add_edge(v_farm3, v_shop1, 1, 3);
adder.add_edge(v_farm3, v_shop2, 2, 2);

adder.add_edge(v_shop1, v_target, 3, 0);
adder.add_edge(v_shop2, v_target, 3, 0);
```



Min Cost Max Flow with BGL

Running the algorithm:

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

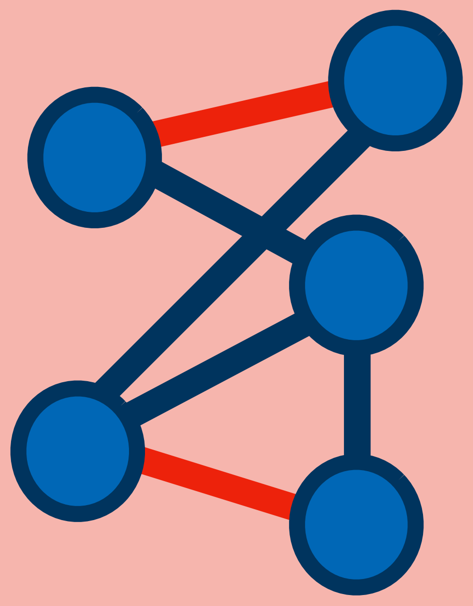
Min Cost Max Flow with BGL

Running the algorithm:

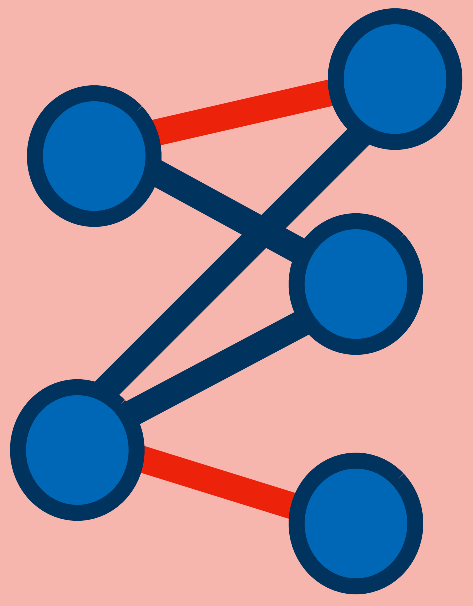
```
// Option 2: Min Cost Max Flow with successive_shortest_path_nonnegative_weights
boost::successive_shortest_path_nonnegative_weights(G, v_source, v_target);
int cost2 = boost::find_flow_cost(G);
std::cout << "cost " << cost2 << "\n"; // 12
// Iterate over all edges leaving the source to sum up the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

Problem Landscape

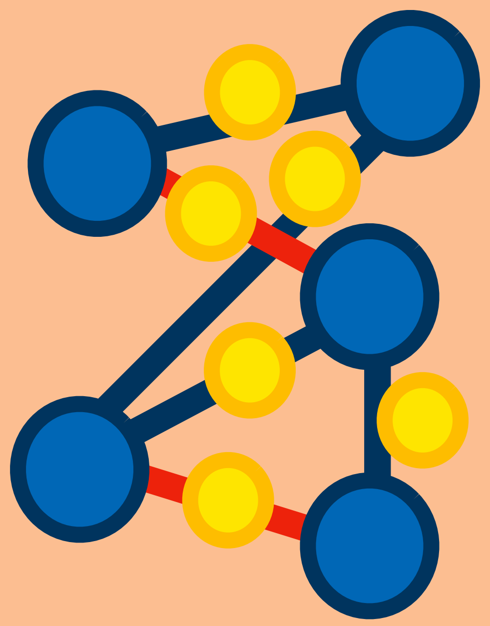
General Matching



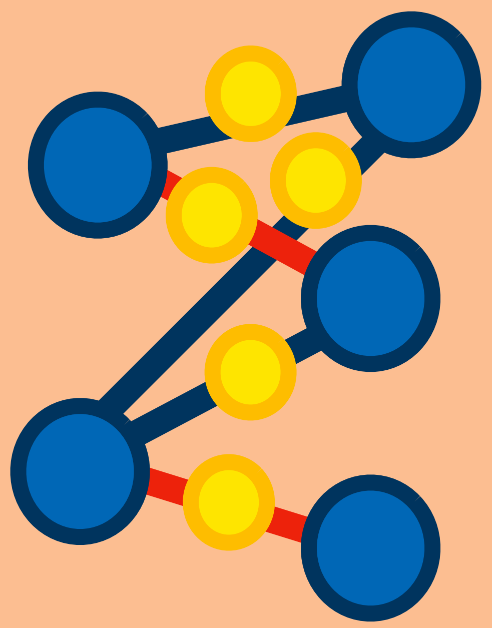
Bipartite Matching



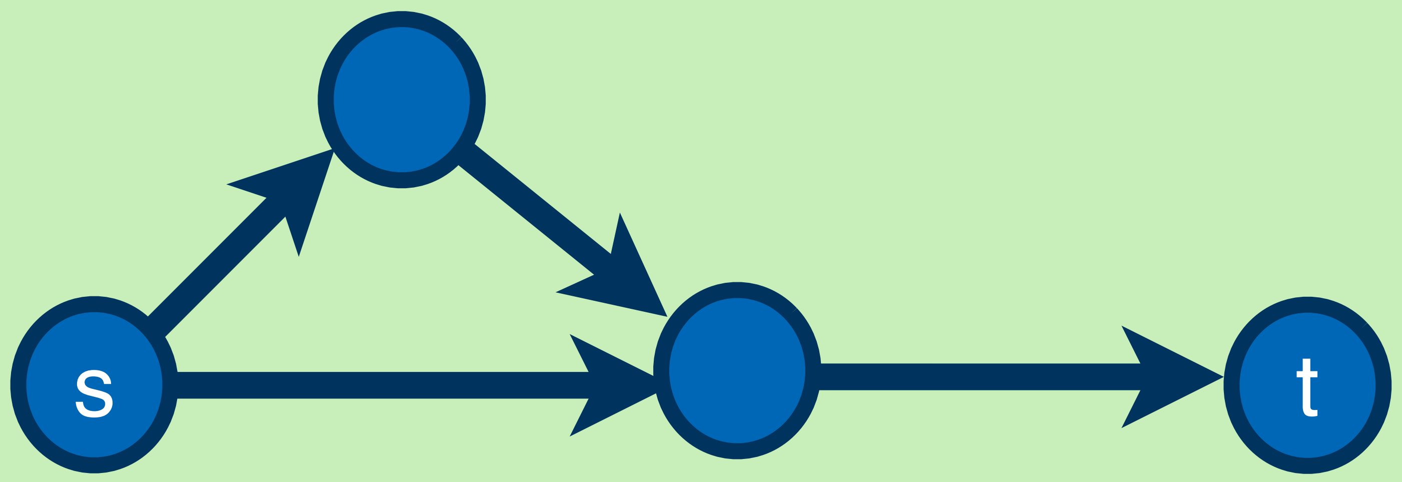
General Cost Matching



Bipartite Cost Matching

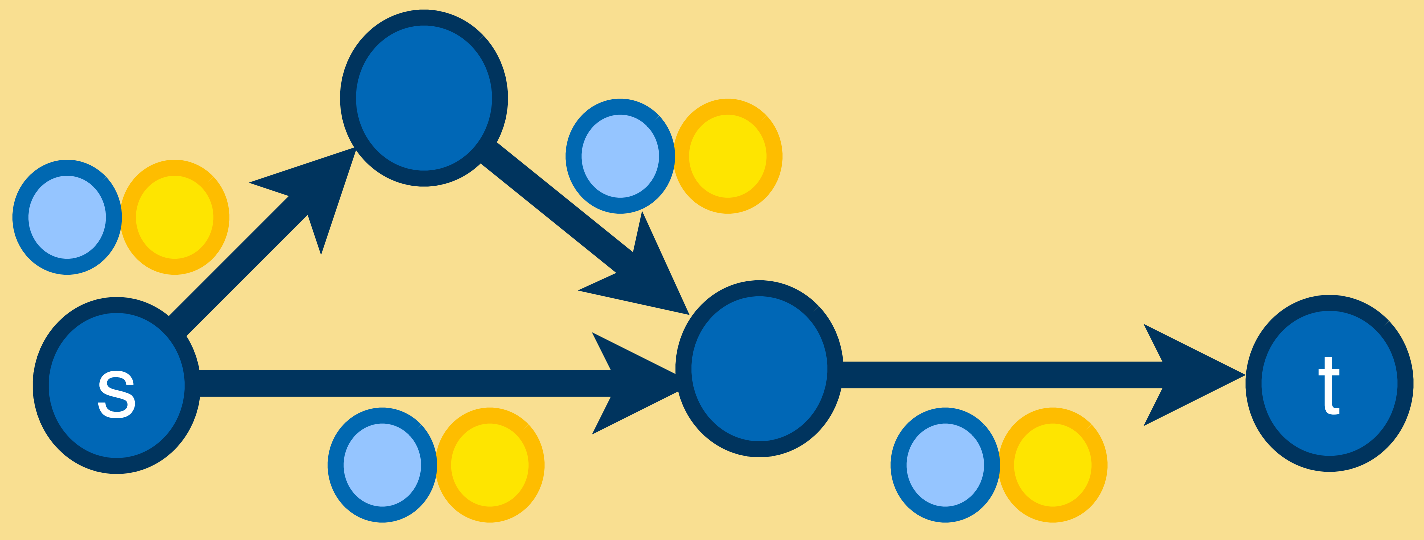


Edmonds Maximum Cardinality Matching



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