Graphs – Matchings

A *matching* in an undirected graph is a set of edges such that every vertex touches at most one edge.

We want to find the *maximal matching*.

Graphs – Matchings

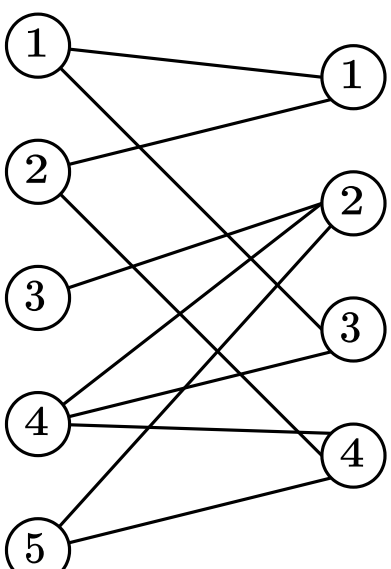
A *matching* in an undirected graph is a set of edges such that every vertex touches at most one edge.

We want to find the *maximal matching*.

In general graphs the best-known algorithms are very complicated.

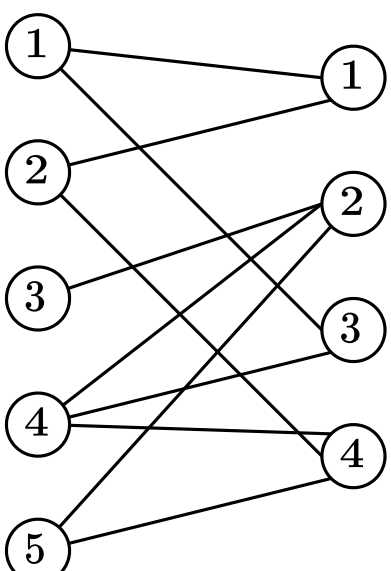
Graphs – Bipartite graphs

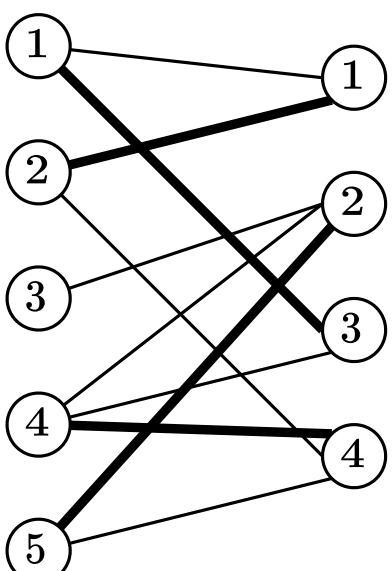
A *bipartite graph* is a graph in which vertices can be separated into two disjoint sets A and B where no edges connect vertices in the same set.



Graphs – Bipartite graphs

A *bipartite graph* is a graph in which vertices can be separated into two disjoint sets A and B where no edges connect vertices in the same set.





In bipartite graphs a maximal matching can be found easily.

Graphs – Matching in bipartite graphs

Greedy algorithm doesn’t work.

Graphs – Augmenting paths

Key concept – *augmenting paths*.

Assume we have a candidate matching M (valid but not necessarily maximal).

An augmenting path is a path that starts with an unmatched vertex, alternates between edges in and out of M, and ends in an unmatched vertex.

Graphs – Hopcroft-Karp

1. Start with $M = \emptyset$
2. Find all augmenting paths.
   - If none, matching is maximal, return;
3. Extend the matching with the symmetric difference of augmenting paths.
4. Go to 2.

The inner loop takes $\mathcal{O}(m)$ time. It can be proven there are at most $\mathcal{O}\left(\sqrt{n}\right)$ iterations for a runtime of $\mathcal{O}\left(m\sqrt{n}\right)$.

Proof sketch: Each phase increases the length of the shortest augmenting path. After $\mathcal{O}\left(\sqrt{n}\right)$ phases the length is at least $\mathcal{O}\left(\sqrt{n}\right)$. There can be at most $\mathcal{O}\left(\sqrt{n}\right)$ new paths to contribute to the matching, bounding the remaining phases.

Graphs – Hopcroft-Karp

```
size = 0
while find_paths()
  for v in A
    if M[v] is None
      if extend_match(v)
        size += 1
```

Graphs – Hopcroft-Karp

```
fn find_paths()
  q = new Queue
  dist.fill(None)
  for v in A
    if M[v] is None
      dist[v] = 0
      q.push(v)
```

```
while v = q.pop()
for u in N[v]
  if M[u] is None
    return true
  else if D[M[u]] is None
    D[M[u]] = D[v] + 1
    q.push(M[u])
```

Graphs – Hopcroft-Karp

```
fn extend_match(v)
  for u in N[v]
    if M[u] is None or
       (D[M[u]] = D[v] + 1 and extend_match(M[u]))
       M[u] = v
       M[v] = u
       return true
  D[v] = None
  return v
```

Graphs – Vertex cover

Vertex cover is a set of vertices such that all edges in the graph touch one of the vertices in the cover.

We want a minimal such set.

Graphs – Vertex cover

Vertex cover is a set of vertices such that all edges in the graph touch one of the vertices in the cover.

We want a minimal such set.

In general, finding a minimal vertex cover is NP-complete.

Graphs – Kőnig's theorem

In bipartite graphs we can get a cover based on the matching.

Theorem (Kőnig): *In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.*

A constructive proof of this theorem gives a method for obtaining the cover from a matching (see XAP last year).

Graphs – Flow networks

Two distinguished vertices – source s and sink t.

Push units of *flow* from s to t while respecting edge capacities.

Graphs – Flow networks

Formally – find a function $f : E \to \mathbb{Z}$[1] assigning flow values to all edges. Define:

$f_{\text{out}} : V \to \mathbb{Z}$ as $f(v) = \sum_{(v,u)\in E} f(v,u)$; and
$f_{\text{in}} : V \to \mathbb{Z}$ as $f(v) = \sum_{(u,v)\in E} f(u,v)$.

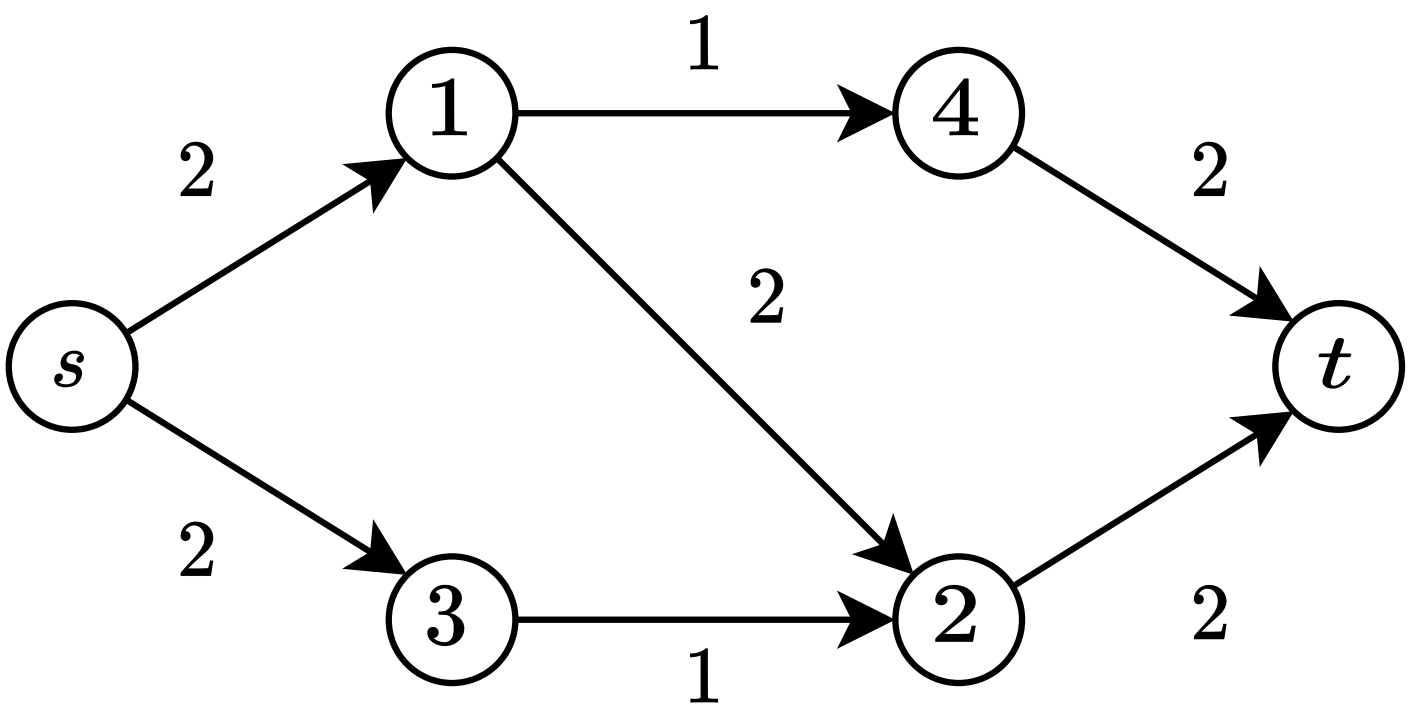
The following restrictions must be respected:

- For all v, u: $f(v,u) = -f(u,v)$;
- For all v, u: $f(v,u) \leq c(v,u)$;
- for all v other than s, t: $f_{\text{out}}(v) = f_{\text{in}}(v)$;

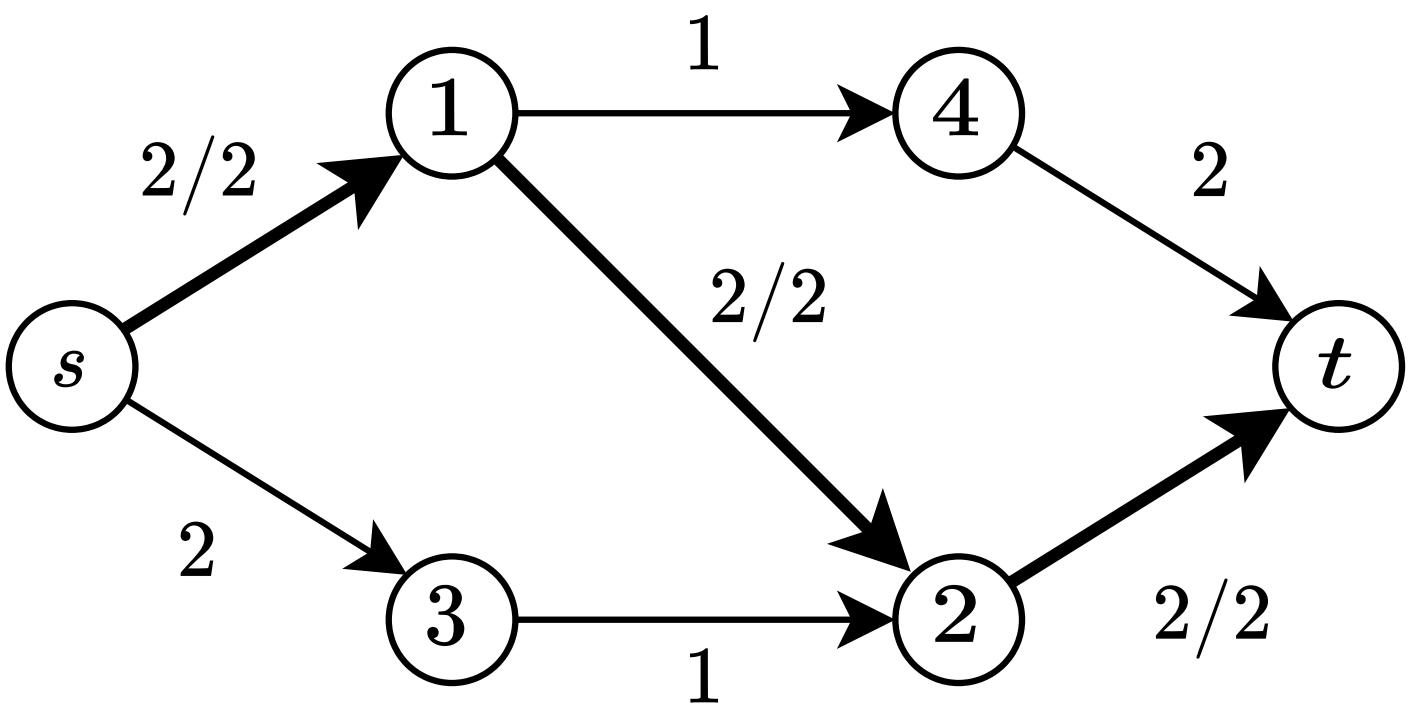
value of the flow is defined as $|f| := f_{\text{out}}(s) - f_{\text{in}}(s)$.

[1] Also works for $\mathbb{Q}$, for $\mathbb{R}$ it's much harder to solve.

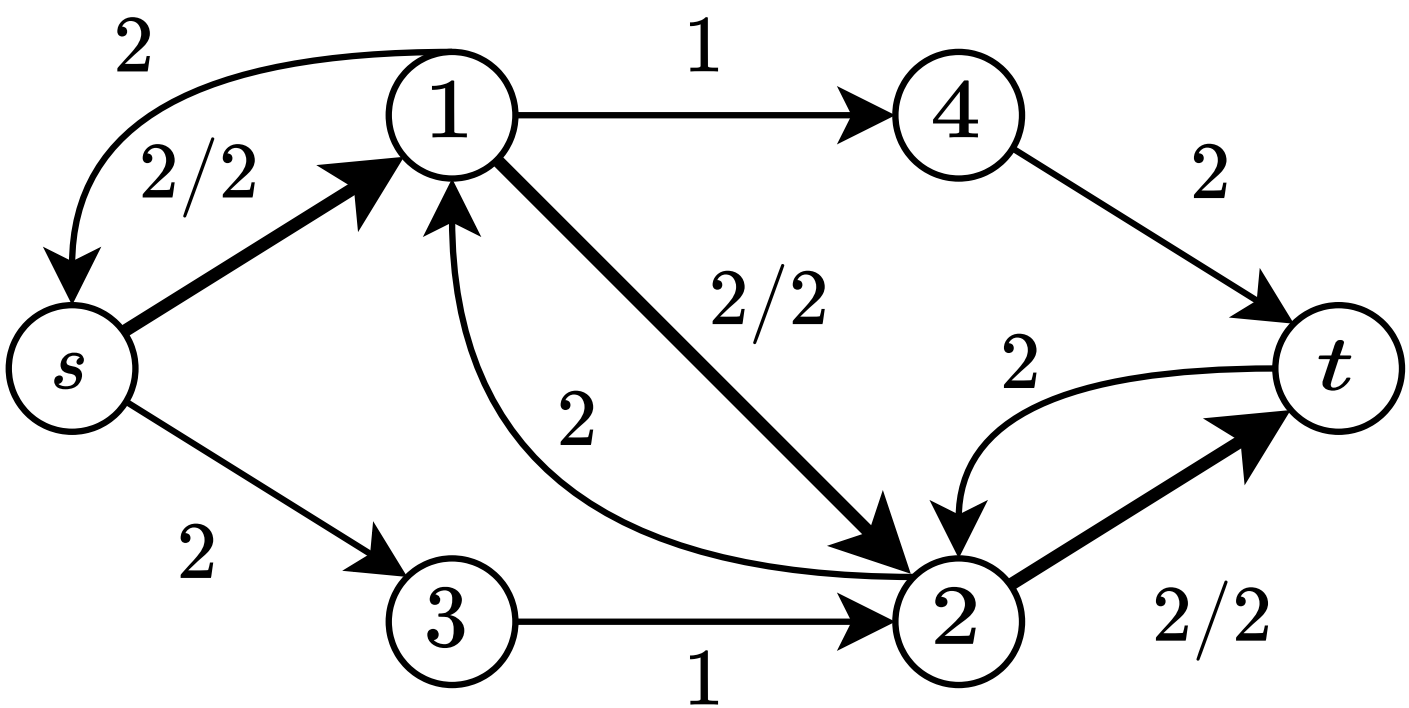
Graphs – Flow networks



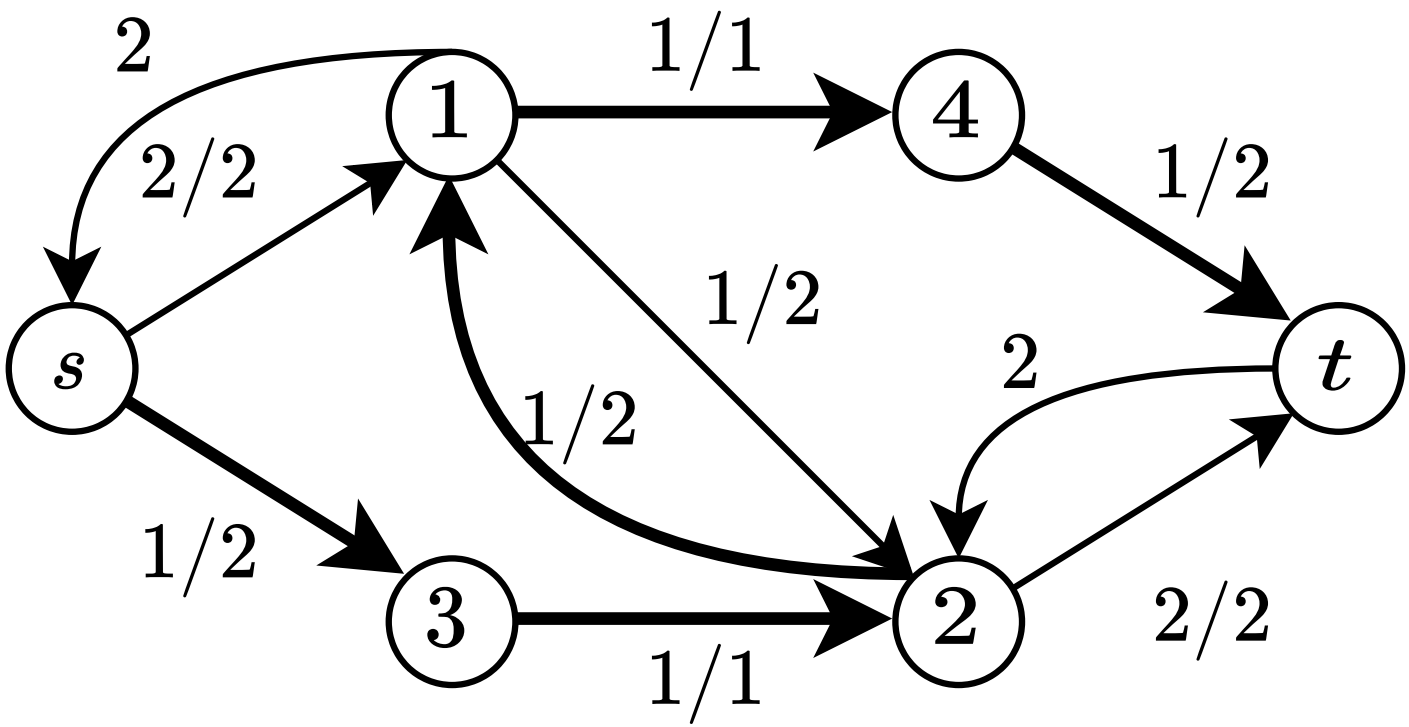
Graphs – Flow networks



Graphs – Flow networks



Graphs – Flow networks



Graphs – Max-flow min-cut theorem

An s,t-*cut* in a graph is a set of edges that disconnects s from t.

The **max-flow min-cut theorem** states that if we treat the graph as a flow network then the maximum flow value is equal to the minimum weight (capacity) of any s,t-cut.

In particular, if all edges have capacity 1 then we get the minimal cut wrt. number of edges.

Graphs – Max-flow min-cut theorem

An s,t-*cut* in a graph is a set of edges that disconnects s from t.

The **max-flow min-cut theorem** states that if we treat the graph as a flow network then the maximum flow value is equal to the minimum weight (capacity) of any s,t-cut.

In particular, if all edges have capacity 1 then we get the minimal cut wrt. number of edges.

This is a central theorem for solving max-flow problems.

Graphs – Ford-Fulkerson

A template for greedy algorithms computing the max flow.

Core idea: *residual network* $G_f = \left(V, E_f\right)$.

The residual network is the flow network with some flow value applied.

Take $c_f(v,u) = c(v,u) - f(v,u)$ and $E_f = \{(v,u) \mid c(v,u) > 0\}$.

Graphs – Ford-Fulkerson

The idea is that any path from s to t in the residual network is an *augmenting path*.

Given such a path we can push flow equal to minimum of capacities on the path and increase the overall flow.

The bound on any implementation of this method on $\mathbb{Z}$ capacity values is $\mathcal{O}(m \cdot |f|)$ – finding a path is $\mathcal{O}(m)$ and each path increases the flow.

Graphs – Edmonds-Karp

Always choose the shortest path from s to t with a BFS.

$\mathcal{O}(nm^2)$.

Graphs – Dinitz

Also known as “Dinic” but Dinitz is correct.

Runs in $\mathcal{O}(n^2 m)$ and even faster in special cases.

Idea – find a *blocking flow*, i.e. a *maximal* set of augmenting paths in the residual network.

More precisely:

- construct a DAG such that going forward always brings us closer to the sink;
- saturate the DAG with a series of DFS pushes.

Graphs – Dinitz

First construct a layer graph from the residual network.

Run a BFS from source to find distances to each vertex. If the sink is not reachable then we already have maximum flow.

The layer graph has only the edges that increase the distance by 1, i.e. (v, u) is kept iff $D[v] + 1 = D[u]$.

Graphs – Dinitz

Now, repeat:

- run a DFS search from source to find any path to sink in the layer graph;
- push a flow through that path.

Crucial optimization: **delete any useless edges from the layer graph**.

An edge is useless if:

- during the DFS it doesn't lead us to the sink;
- or we push flow through it and decrease its capacity to 0.

Once no more paths exist, go back to step one and construct a new layer graph.

Graphs – Dinitz running time

The layer graph construction takes $\mathcal{O}(m)$ time.

Crucial lemma: *Applying a blocking flow increases the distance from s to t in the residual network.*

From this we know the main loop iterates at most n times.

Graphs – Dinitz running time

Assume for a moment all capacities in the original network are 1 (this is a common occurrence).

Every time we use an edge when pushing a flow it gets deleted, therefore, the body of the loop cannot take more than $\mathcal{O}(m)$ time.

There is a stricter bound – after k DFS runs all paths are of length at least k. So further iterations can increase the flow by at most $\frac{m}{k}$. Therefore, the total iteration count is bounded by $\max_k k + \frac{m}{k}$. This is bounded by $\mathcal{O}\left(\sqrt{m}\right)$, but also $\mathcal{O}\left(n^{\frac{2}{3}}\right)$ if we take $m = O(n^2)$.

Together we have bounds $\mathcal{O}\left(m^{\frac{3}{2}}\right)$ and $\mathcal{O}\left(mn^{\frac{2}{3}}\right)$ in unit networks.

Graphs – Dinitz running time

In general, since each push deletes at least one edge, there can be only $\mathcal{O}(m)$ pushes per blocking flow. Moreover, each DFS takes at most $\mathcal{O}(n)$ amortized time – each edge either gets traversed or deleted; deletion happens at most once; traversal brings us closer to the sink and distance is bounded by n.

We have $\mathcal{O}(n)$ iterations and $\mathcal{O}(nm)$ time for each for a total of $\mathcal{O}(n^2m)$.

Graphs – Dinitz running time

In general, since each push deletes at least one edge, there can be only $\mathcal{O}(m)$ pushes per blocking flow. Moreover, each DFS takes at most $\mathcal{O}(n)$ amortized time – each edge either gets traversed or deleted; deletion happens at most once; traversal brings us closer to the sink and distance is bounded by n.

We have $\mathcal{O}(n)$ iterations and $\mathcal{O}(nm)$ time for each for a total of $\mathcal{O}(n^2m)$.

A more rigorous analysis uses amortisation.

This reasoning is important, since in many specific networks the time can be bounded further.

Graphs – Max flow min cost

We can give edges weights and ask for a minimum-cost flow.

The cost is computed as $\sum_{(v,u),f(v,u)>0} w(v,u) \cdot f(v,u)$.

If there are no negative-cycles then plugging Bellman-Ford into Ford-Fulkerson gives an $\mathcal{O}(n^2m^2)$ solution.

[2] This can be further brought down to $\mathcal{O}(n^2\sqrt{m})$

[3] CF blog on Dinitz and max-flow min-cost: https://codeforces.com/blog/entry/104960.

Graphs – Max flow min cost

We can give edges weights and ask for a minimum-cost flow.

The cost is computed as $\sum_{(v,u),f(v,u)>0} w(v,u) \cdot f(v,u)$.

If there are no negative-cycles then plugging Bellman-Ford into Ford-Fulkerson gives an $\mathcal{O}(n^2m^2)$ solution.

In practice, **push-relabel** is used. A basic implementation gives $\mathcal{O}(n^2m)$[4].

We skip the discussion on this.[5]

[4] This can be further brought down to $\mathcal{O}(n^2\sqrt{m})$

[5] CF blog on Dinitz and max-flow min-cost: https://codeforces.com/blog/entry/104960.