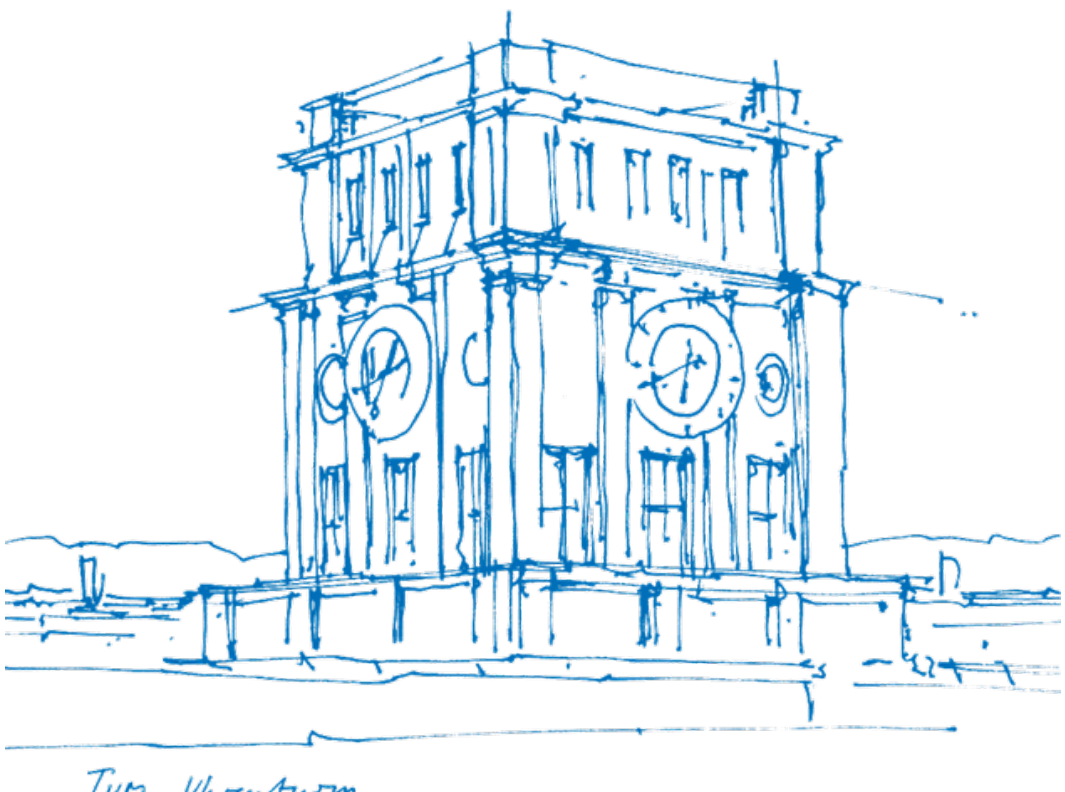
School of Computation, Information and Technology
Technical University of Munich

AACPP 2025

Week 6: Graphs 2 – Advanced Boogaloo



Mateusz Gienieczko, **Mykola Morozov**

School of Computation, Information and Technology
Technical University of Munich

2025.06.17

Fourth round – survey

Fifth round

Deadline – 24.06.2025, 10:00 AM.

CAT – Catventure

We have a large 2D grid with n points on it.

We can travel along the grid lines. Taking a turn *forces* us to switch between *run* and *stroll*, while arriving at a point *allows* us to change the mode.

Objective: get from point 1 to n while walking the least.

CAT – Catventure

Brute force – treat the entire grid as a graph, for each coordinate keep the best cost of getting there from each of the four directions.

Run a BFS on it (all edges have length 1).

$\mathcal{O}(XY \log XY)$ where X is the max x_i and Y is max y_i.

CAT – Catventure

Simplifying observations:

- always move between points;
- it always makes sense to move “towards” the target in both dimensions.

In fact, the following is true:

The best achievable cost to travel between i and j is $\min\big(|x_i - x_j|, |y_i - y_j|\big)$.

Better brute-force – create a complete graph on n vertices with edge costs defined as above and run a Dijkstra.

$\mathcal{O}\big(n^2 \log n\big)$

CAT – Catventure

Can we make the graph sparser?

Assume we have a path with an optimal cost in which we travel from i to j, $x_i < x_j$, and we choose to stroll horizontally (so the cost is $x_j - x_i$). If there exists a k such that $x_i < x_k < x_j$ then we can always first travel to k and then to j, and the cost remains the same.

This is also true for $x_i > x_j$ and the Y dimension with vertical strolls.

CAT – Catventure

Create a sparse (planar, in fact) graph where each node is connected to at most 4 other nodes – the closest to the left, right, up, and down.

By our above reasoning for any shortest path in the complete graph there exists a path of the same length in this simplified graph.

Running Dijkstra on this gives us $\mathcal{O}(n \log n)$.

PAW – Park Walks

This was the hardest task by far.

The task describes an undirected graph and directly asks about *cycles*:

"*routes that start and end at the same spot, and neither go through any other spot more than once nor use any path more than once*"

We have to decide:

- if all cycles in the graph are of the same length;
- the number of such cycles.

PAW – Park Walks

Aside: the task asks for the solution mod $10^9 + 7$.

This is used when the actual number is so large it won't fit into usual integers. It doesn't have "real life" sense.

PAW – Park Walks

As we know, the total number of cycles can be exponential, so exhaustively checking all cycles only gets 2 points for $n \leq 18$.

The first step towards a solution is focusing on biconnected components.

PAW – Park Walks

Any cycle in a graph is fully contained within a single biconnected component.

Assume we can solve the task for a single BCC. Then we can solve it individually for each BCC in the graph, assert cycle lengths are equal, and return the sum of counts.

We know how to find BCCs in $\mathcal{O}(n + m)$ from the class.

PAW – Park Walks

A biconnected component that is just a single cycle of length l obviously satisfies the length requirement and contributes $2l$ to the result (we can start at any node and go in either direction).

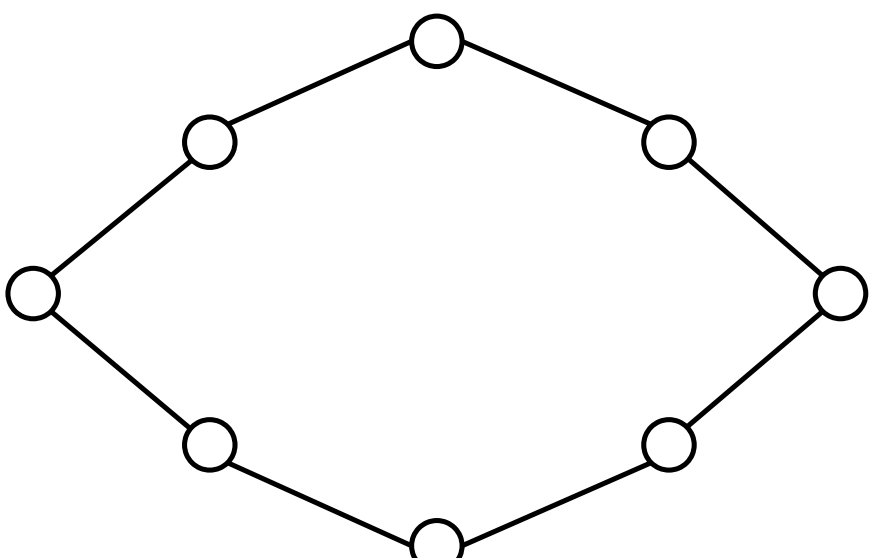
What can we add to a cycle like that to get more routes but not violate the length restriction?

PAW – Park Walks

What can we add to a cycle like that to get more routes but not violate the length restriction?

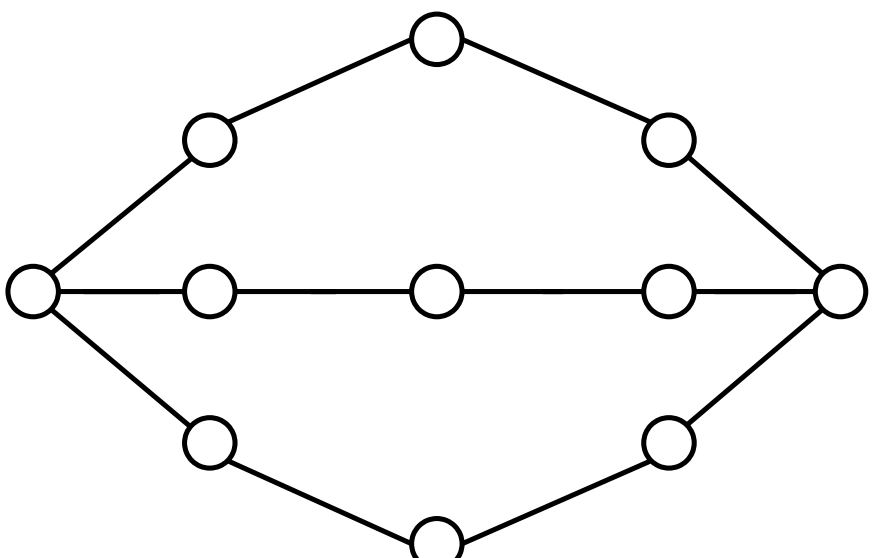
PAW – Park Walks

What can we add to a cycle like that to get more routes but not violate the length restriction?



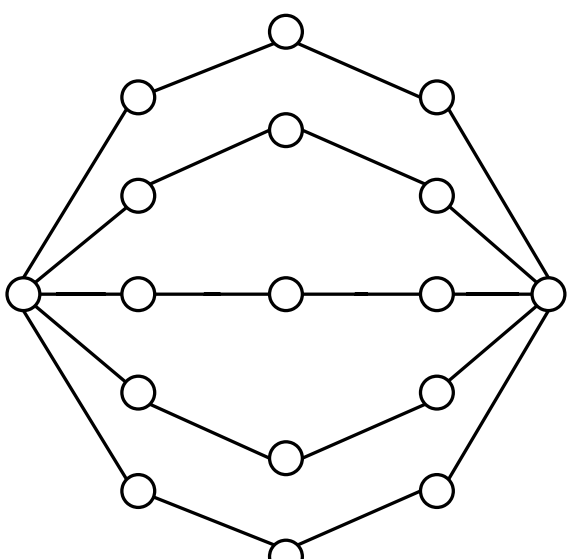
PAW – Park Walks

What can we add to a cycle like that to get more routes but not violate the length restriction?



PAW – Park Walks

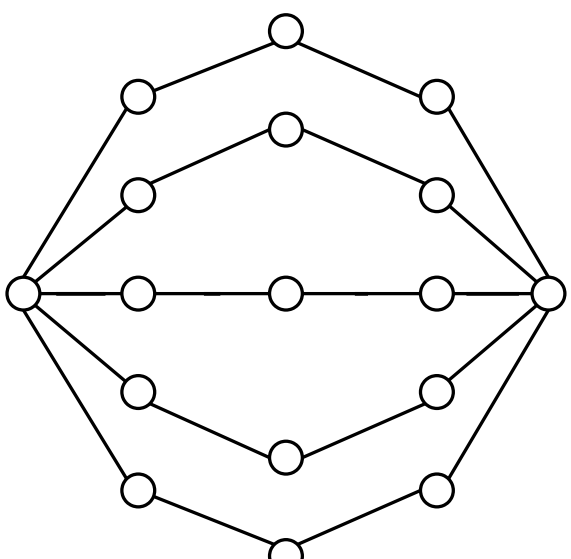
What can we add to a cycle like that to get more routes but not violate the length restriction?



PAW – Park Walks

We'll call k paths of (edge) length l arranged as that an (k, l)-onion.

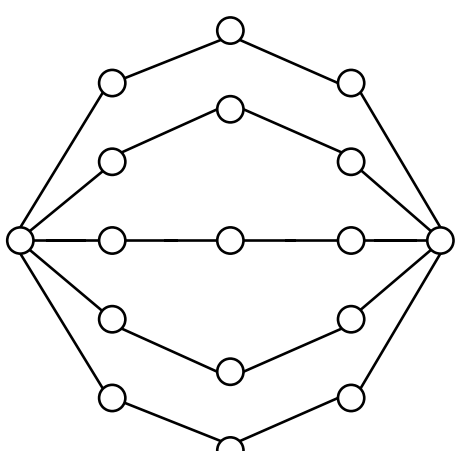
So this is a $5, 4$ onion.



PAW – Park Walks

In an (k, l)-onion:

- there are $\binom{k}{2}$ cycles (choose any two distinct paths);
- all cycles are of length $2l$;
- this adds $\binom{k}{2} \cdot 2l$ to the result (start at any point, go in either direction).



PAW – Park Walks

Onions are the only shape a biconnected component that satisfies our requirements can have.

Recall any biconnected component can be decomposed into a main cycle and then a number of paths (“ear decomposition”).

If we have a (k, l)-onion then any path that we can add that wouldn’t violate the requirements turns it into a $(k + 1, l)$-onion.

PAW – Park Walks

Plan now is:

1. Divide graph into BCCs.
2. For each BCC decide if it is an onion
   - if no BCC is even a cycle then the result is `FUR-LORN`;
   - if some are not onions then the result is `MEOW-NO`;
   - if there are two BCCs with different onion lengths then `MEOW-NO`.
3. Sum the number of cycles over all BCCs.

PAW – Park Walks

How to check if a BCC is an onion?

If it is a cycle then it's a $(2, l)$-onion and we're done.

Otherwise, there should be exactly two vertices that have degree > 2. Call them v and u.

All paths connecting v and u have to have length l.

Multiple ways to check this, e.g. run a BFS from v and assert that

- $D[u] = l$
- $\forall_x . D[x] \leq l$

Everything we've done is $\mathcal{O}(n + m)$.

Recall the plan

- Greedy and dynamic programming (DP)
- Trees
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- **Ways to run DP on graphs (Toposort)** ← *we are here*
- **Advanced graph algorithms (Matchings, flows)** ← *we are here*
- Binary Search Trees
- Number theory
- String algorithms (KMP, tries, suffix tables)
- Some problems can't* even be solved efficiently (NP-completeness)

Graphs – strongly connected components

In a *directed* graph a connected component is not really coherent.

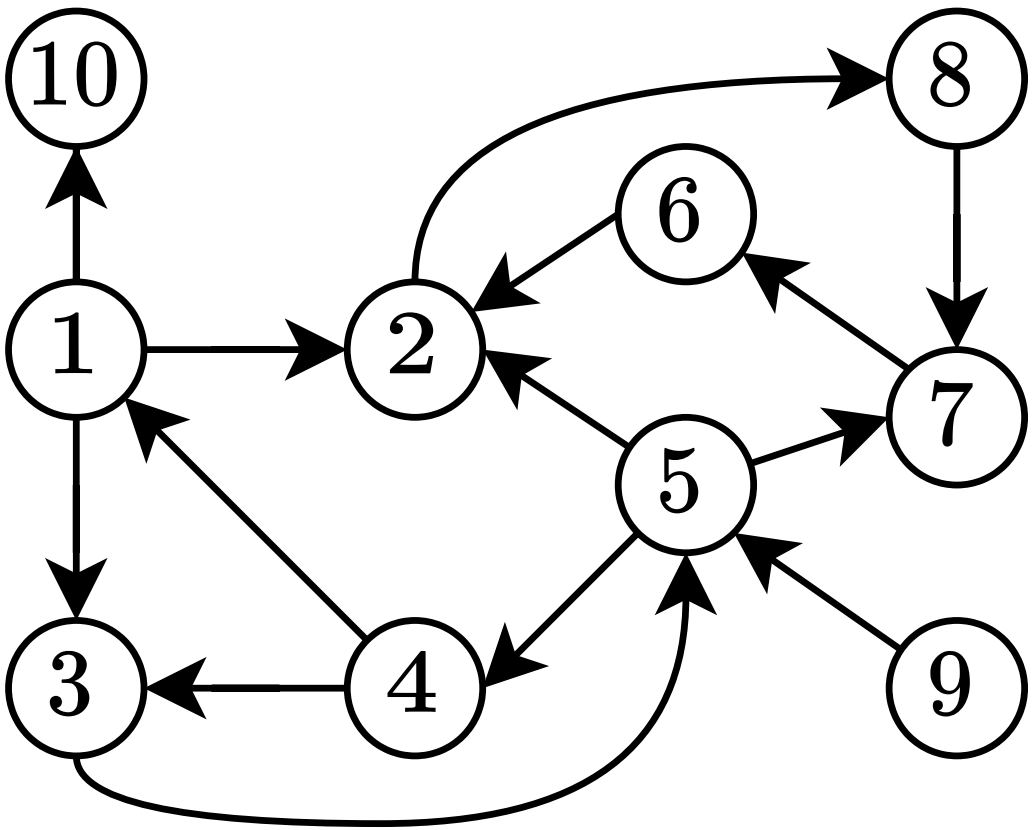
We define *strongly connected components* as maximal sets of vertices where there exists a pair between each pair of vertices (bidirectional).

Graphs – strongly connected components

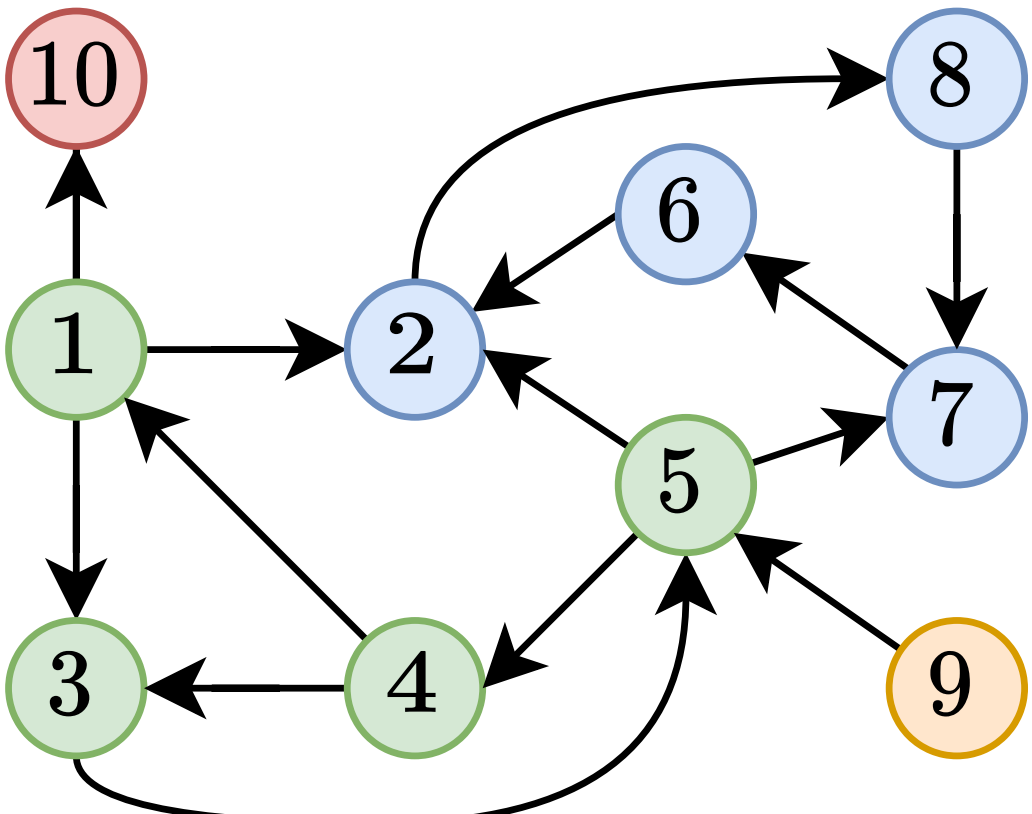
Any directed graph can be decomposed into a graph of its SCCs.

An SCC graph is always a DAG (Directed Acyclic Graph).

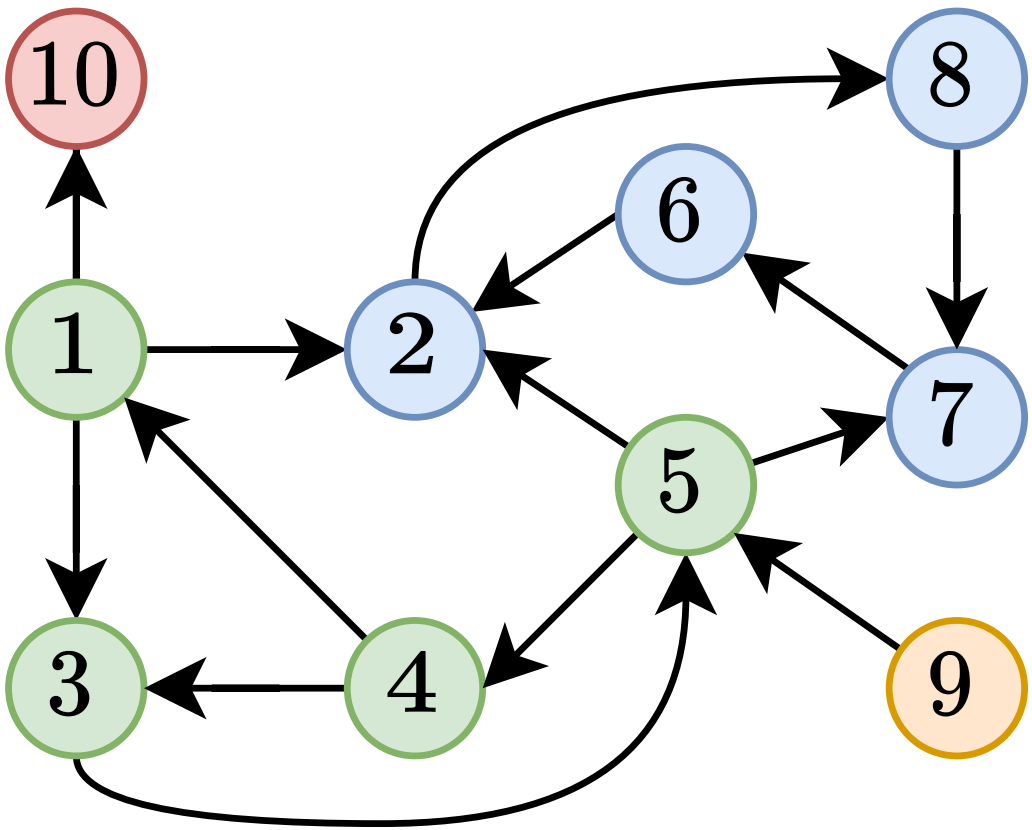
Graphs – strongly connected components

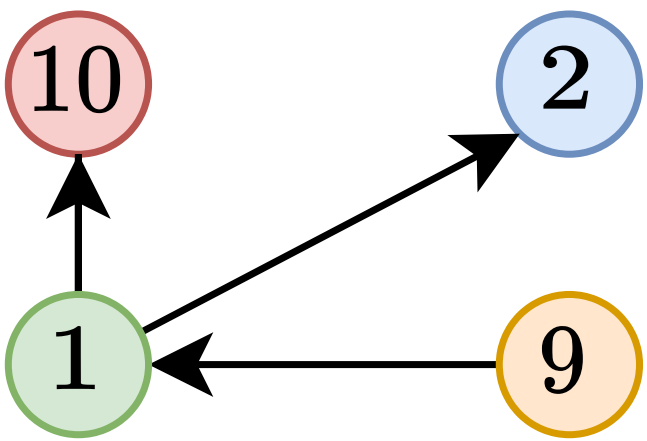


Graphs – strongly connected components



Graphs – strongly connected components





Graphs – finding SCCs

Surprise – algorithm by Tarjan.

Run a DFS and keep an additional stack and, for each vertex, a *lowlink*.

The lowlink is the earliest (in preorder) vertex on the stack reachable from the current DFS root.

If the lowlink is the same as our preorder it means there are no backwards edges in the entire subtree, so we must be in a distinct SCC from our ancestors.

Graphs – finding SCCs

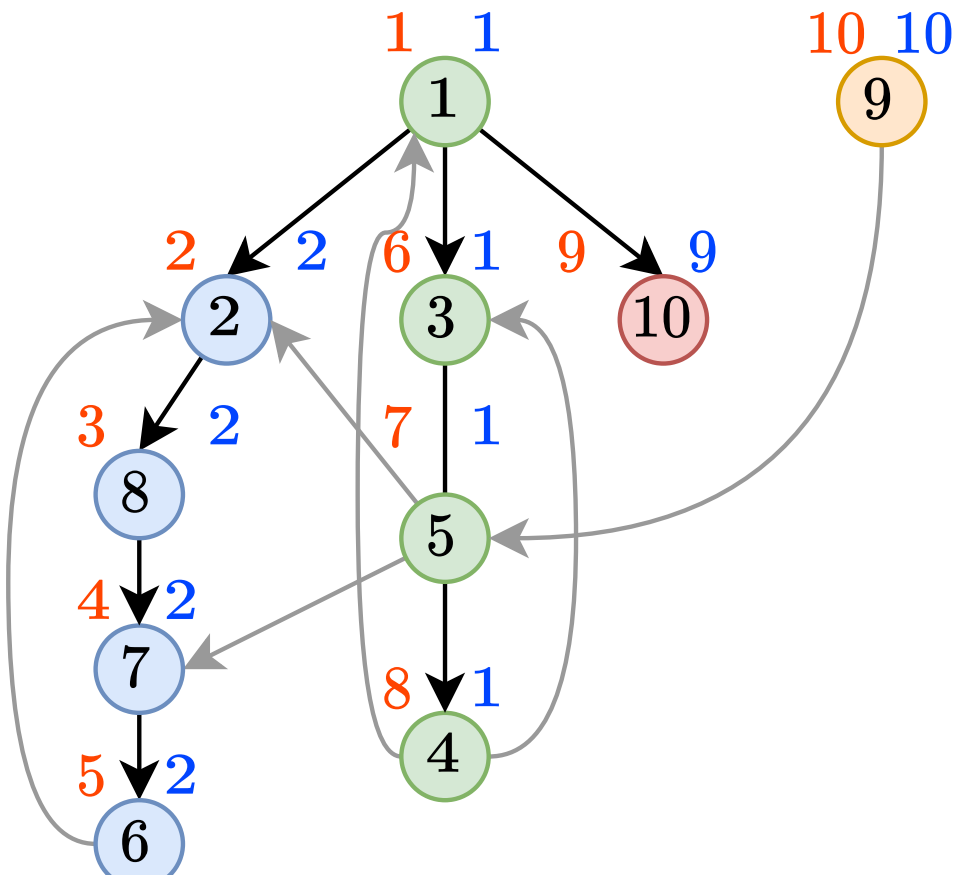
```
for v in V
  if preorder[v] is None
    stack = new Stack
    DFS(v)
```

Graphs – finding SCCs

```
fn DFS(v)
  pre[v] = time
  lowlink[v] = time
  time += 1
  stack.push(v)
  onStack[v] = true
  for u in N[v]
    if preorder[u] is None
      DFS(u)
      lowlink[v] = min(lowlink[v], lowlink[u])
    else if onStack[u]
      lowlink[v] = min(lowlink[v], pre[u])
```

```
//after the loop
if lowlink[v] = pre[v]
  //v is a root of an SCC
  while (w = stack.pop())
    onStack[w] = false
    SCC[v].push(w)
    if v == w { break; }
```

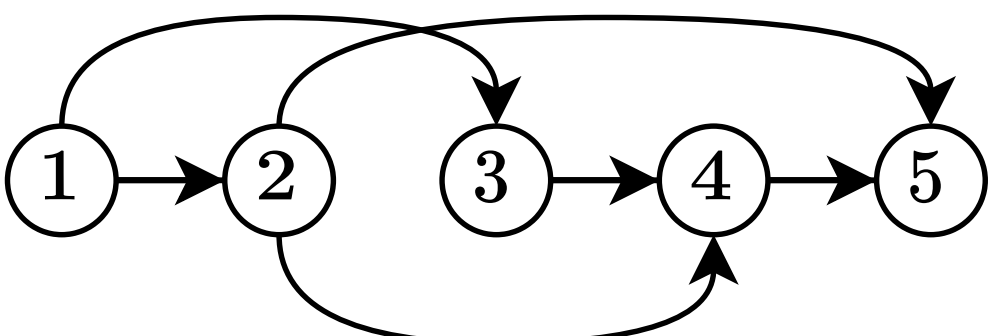
Graphs – finding SCCs



Graphs – topological order

On a DAG, a *topological order* of vertices is an order such that if v is before u then there are no edges (u, v).

Intuitively, “no going backwards”.



Tarjan’s SCC has a nice property – the SCCs are output in an order that is a *reverse* topological order of the graph’s SCC DAG.

Graphs – topological sort

Any DAG can be topologically sorted in linear time.

One algorithm – you're not gonna believe it –

Graphs – topological sort

Any DAG can be topologically sorted in linear time.

One algorithm – you're not gonna believe it – is by Tarjan.

Idea: postorder is a reverse topological order.

Graphs – topological sort

```
topoorder = new List
for v in V
  if visited[v] = false
    DFS(v)
topoorder.reverse()
```

```
fn DFS()
  visited[v] = true
  for u in N[v]
    if visited[u] = false
      DFS(u)
  topoorder.push(v)
```

Graphs – DP on a DAG

DAGs are nice because dataflow has a set direction.

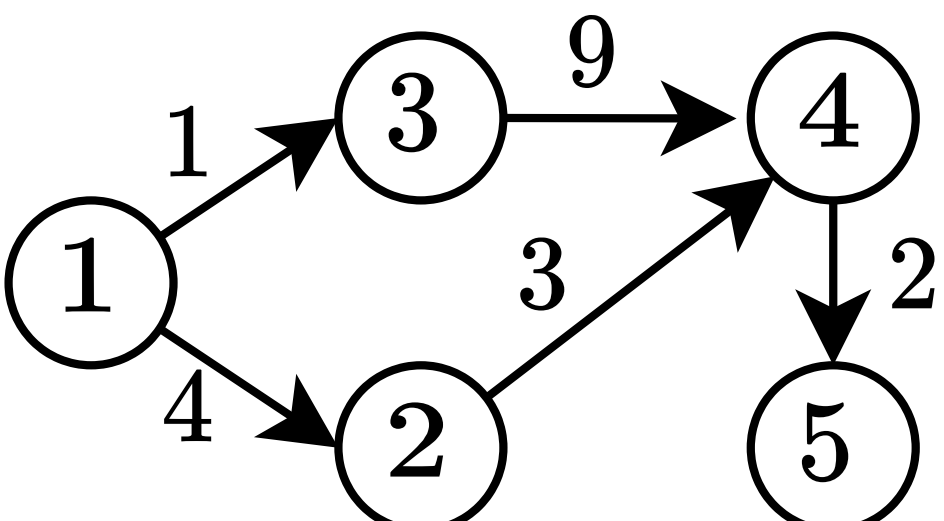
This allows us to run dynamic programming – calculate the result in a given node based on its inputs, as long as we go along the topo order.

Graphs – DP on a DAG

Dexter's house has n platforms, connected with m unidirectional paths of varying lengths. After leaving a platform there is no way to come back to the same platform. Dexter wants to plan a route from platform 1 to n of shortest length. Additionally, he can perform k long-distance jumps that always count as length 1.

```
5 5 1
1 2 4
1 3 1
2 4 3
3 4 9
4 5 2

4
```

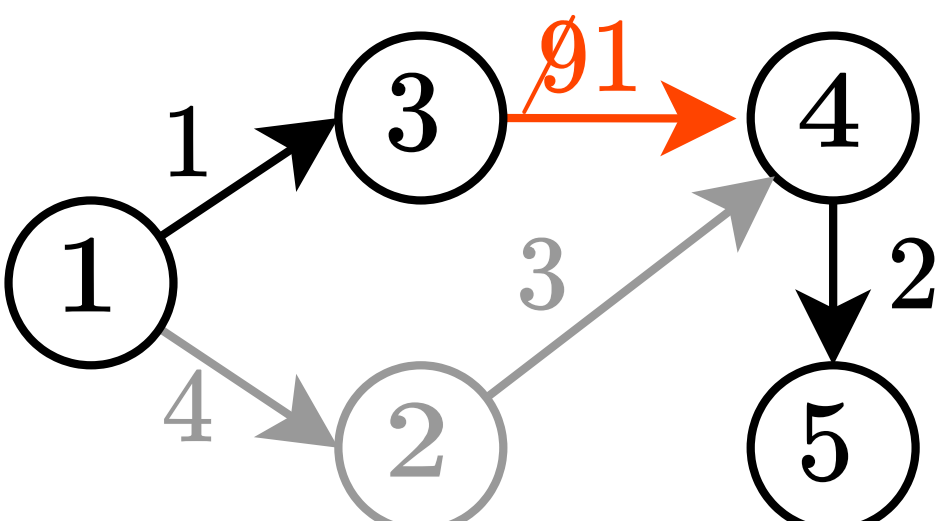


Graphs – DP on a DAG

Dexter's house has n platforms, connected with m unidirectional paths of varying lengths. After leaving a platform there is no way to come back to the same platform. Dexter wants to plan a route from platform 1 to n of shortest length. Additionally, he can perform k long-distance jumps that always count as length 1.

```
5 5 1
1 2 4
1 3 1
2 4 3
3 4 9
4 5 2

4
```



Graphs – DP on a DAG

Topo-sort the graph.

$\mathrm{DP}[v][x]$ – best time to reach v when using x jumps.

$\mathrm{DP}[1][0] = 0$

In topo order:

$$\mathrm{DP}[v][x] = \min_{(u,v,d)\in E} \min(\mathrm{DP}[u][x] + d, \mathrm{DP}[u][x-1] + 1)$$

Return $\min_x \mathrm{DP}[n][x]$

All in time $\mathcal{O}(k(n+m))$

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

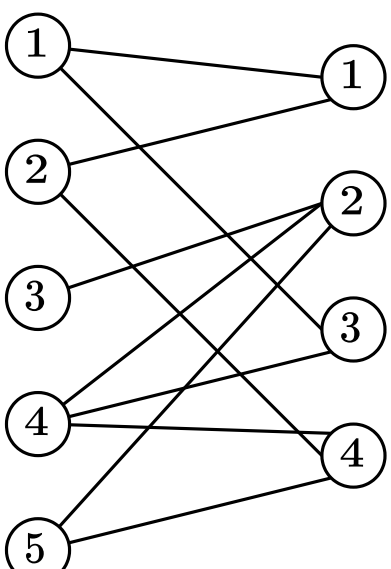
A *matching* in an undirected graph is a set of edges such that every vertex touches at most one edge.

We want to find the *maximal matching*.

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

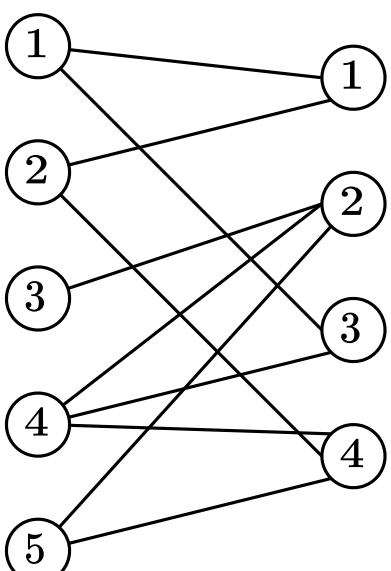
Graphs – Bipartite graphs

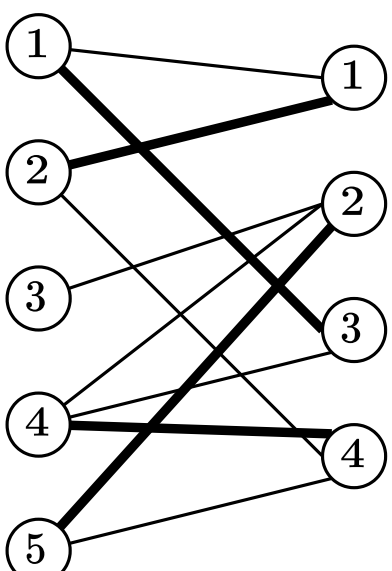
A *bipartite graph* is a graph in which vertices can be separated into two disjoint sets A and B where no edges connect vertices in the same set.



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 $f(v) = \sum_{(u,v)\in E} f(u,v)$.

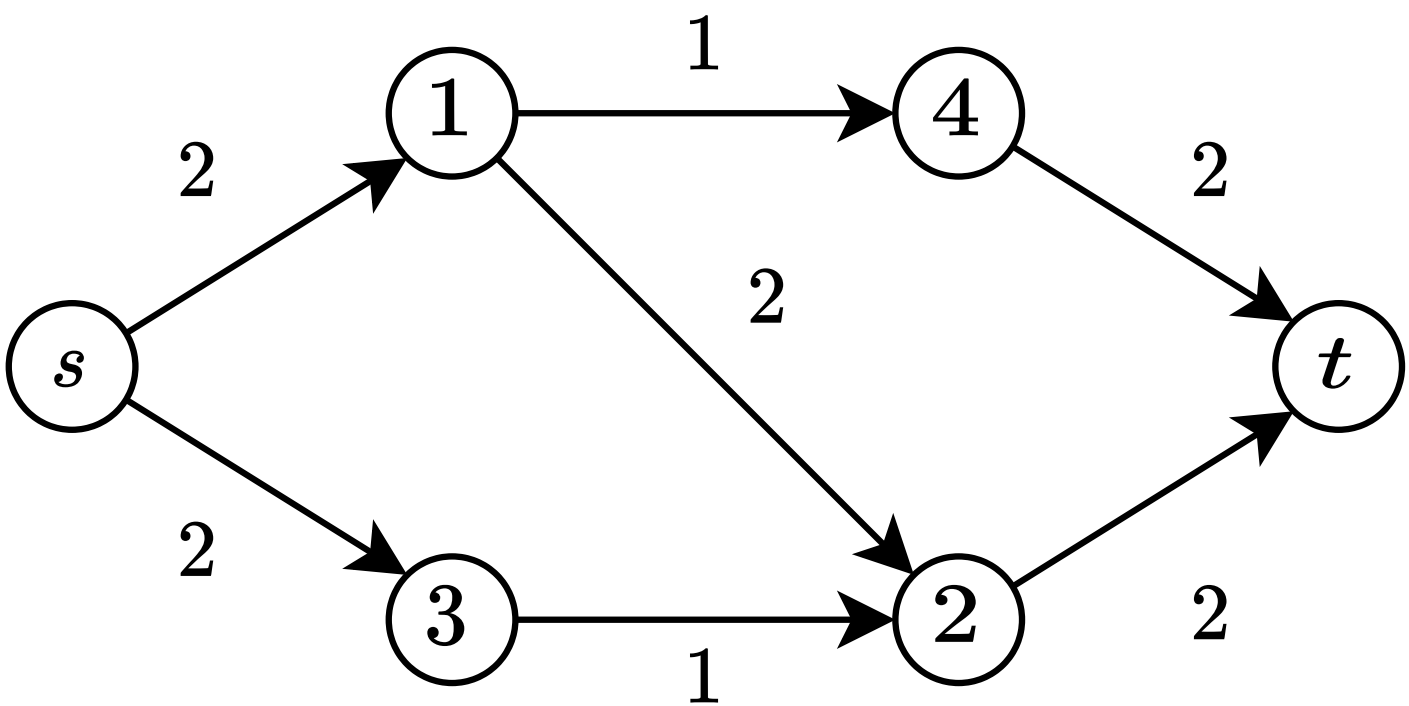
The following restrictions must be respected:

- 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: $f_{\text{out}}(v) = f_{\text{in}}(v)$;

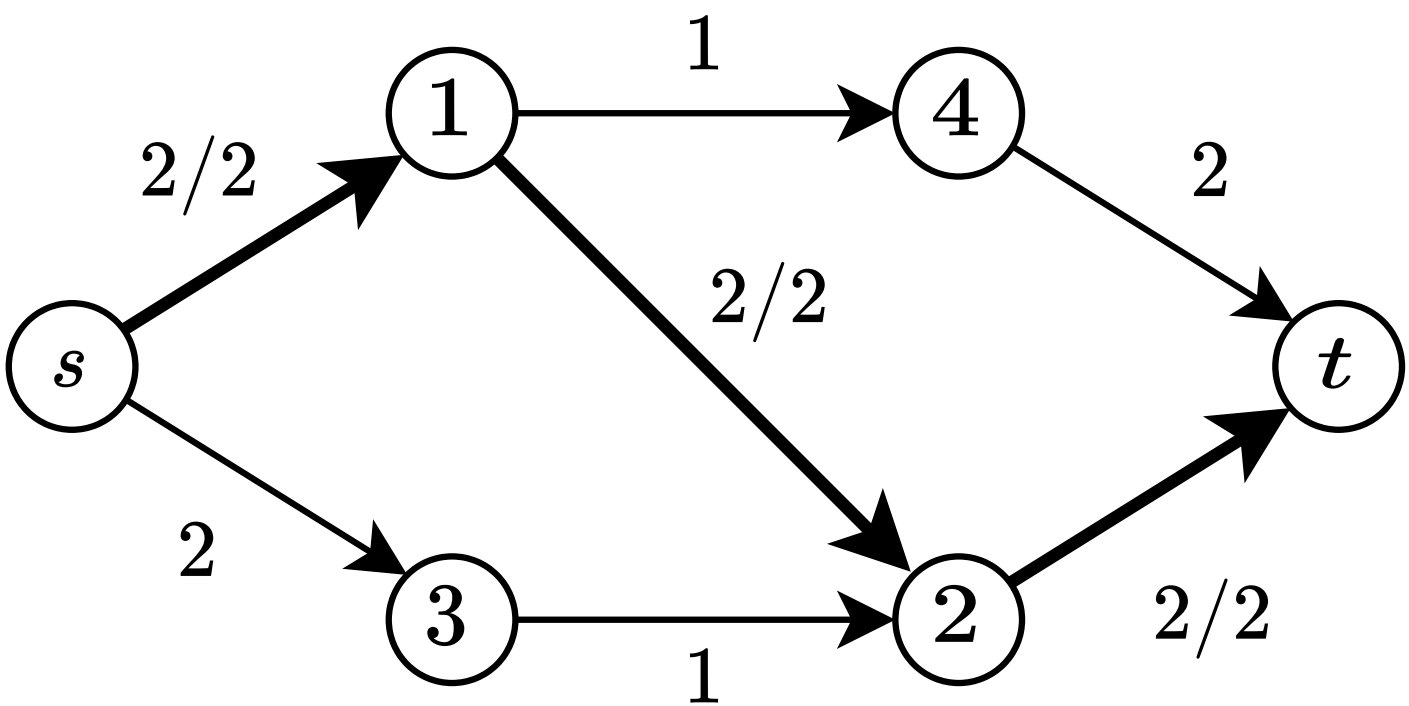
value of the flow is defined as $|f| := f_{\text{out}}(s) - f_{\text{in}}(s)$.

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

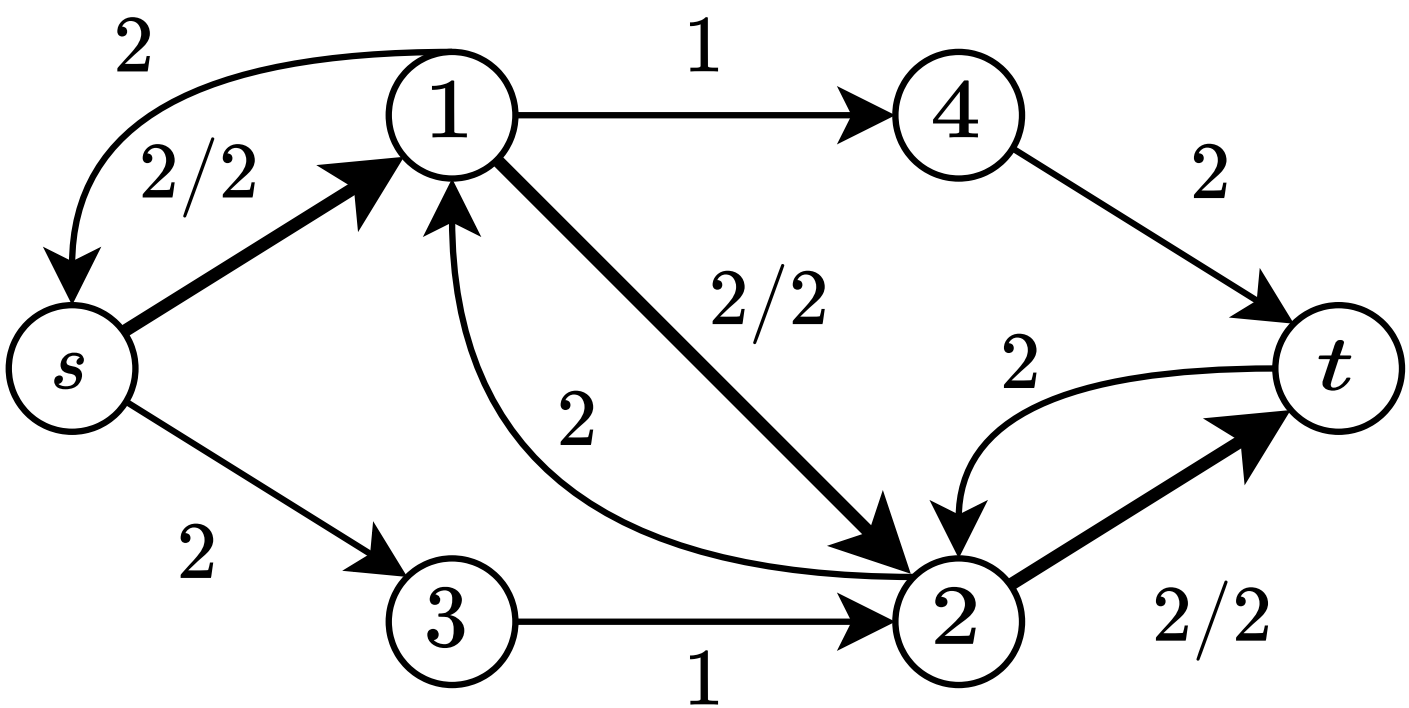
Graphs – Flow networks



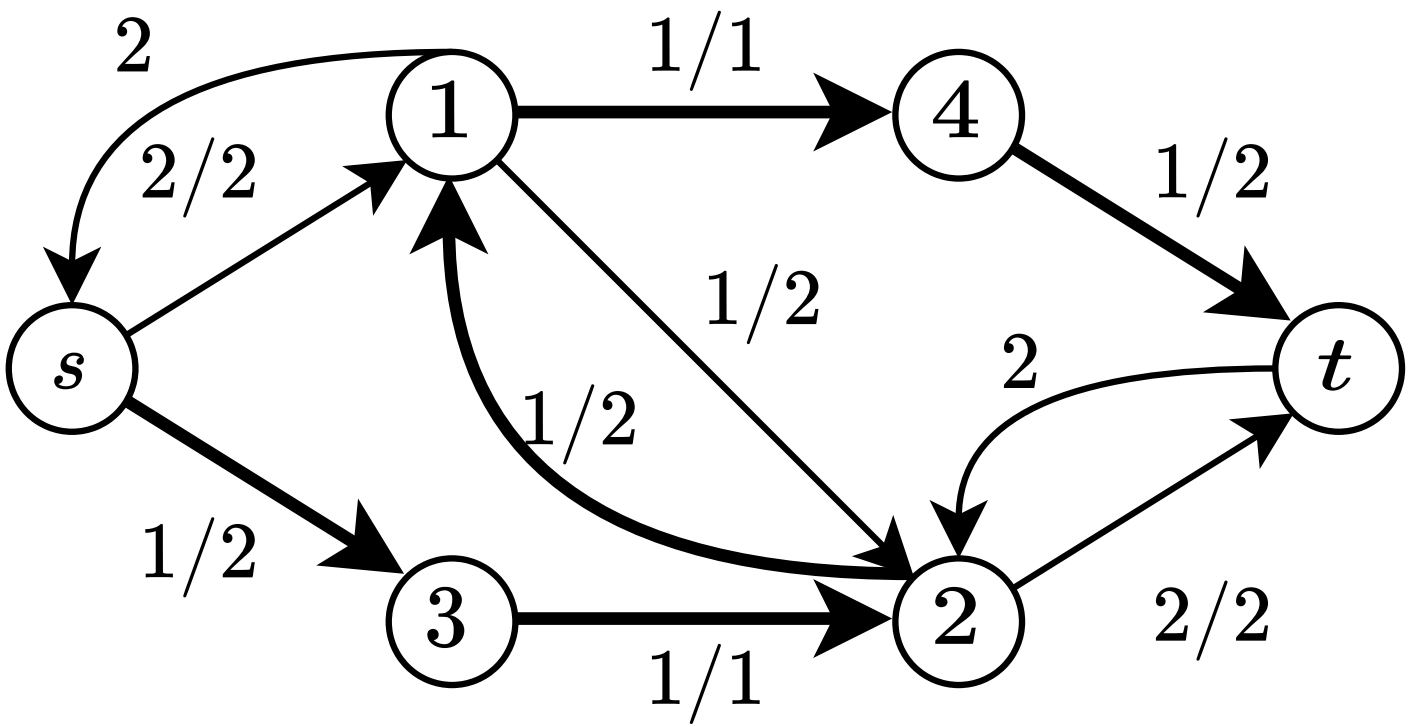
Graphs – Flow networks



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.

See you next week

GRR and LUC: 24.06.2025,
10:00 AM

Good luck!