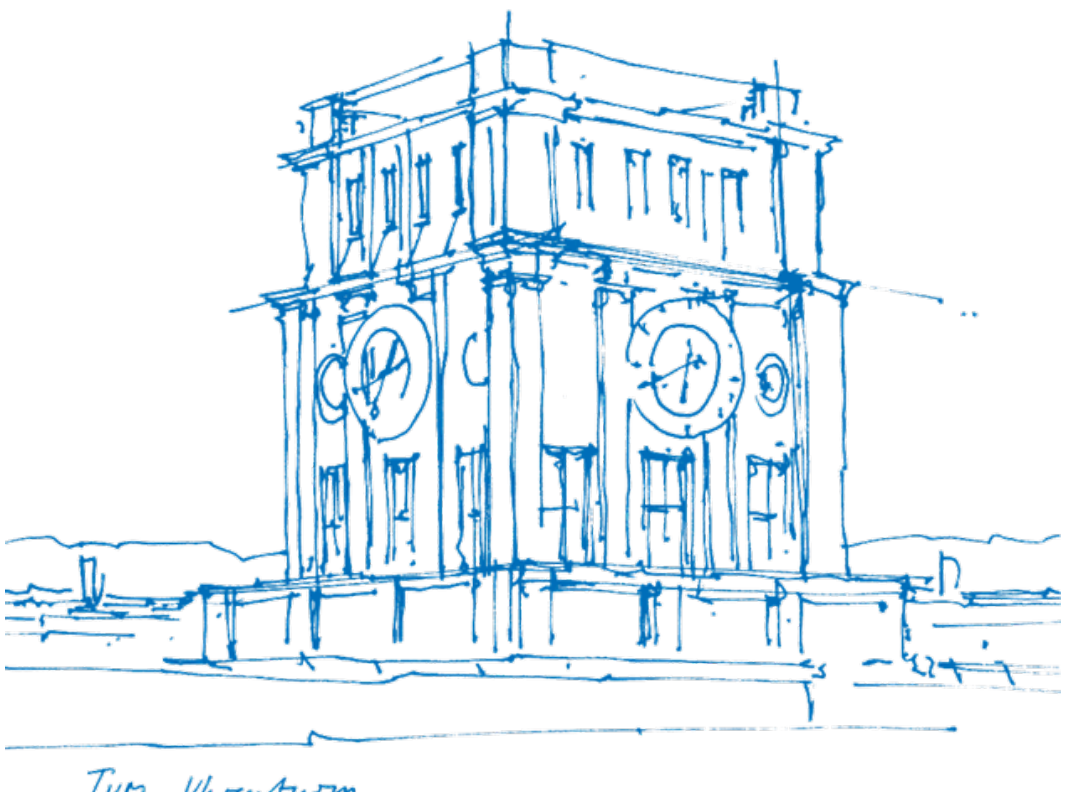
School of Computation, Information and Technology
Technical University of Munich

AACPP 2025

Week 5: Graphs



Mateusz Gienieczko, **Mykola Morozov**

School of Computation, Information and Technology
Technical University of Munich

2025.06.24

Third round – survey

Fourth round

Deadline – **17**.06.2025, 10:00 AM.

10.06 is Pfingstferien

TUV – Tunnel Vision

We are given a tree with one distinguished edge.

TUV – Tunnel Vision

We are given a tree with one distinguished edge.

We have g queries of the form – we put m_g toys in each leaf and propagate them through the tree. Propagation follows the rules:

- assume m toys enter; if $m = 0$ propagation ends;
- consider edges other than the one we came from; let there be e such edges;
- if $e = 0$ propagation ends; otherwise, send $\left\lfloor \frac{m}{e} \right\rfloor$ toys through each edge.

TUV – Tunnel Vision

We are given a tree with one distinguished edge.

We have g queries of the form – we put m_g toys in each leaf and propagate them through the tree. Propagation follows the rules:

- assume m toys enter; if $m = 0$ propagation ends;
- consider edges other than the one we came from; let there be e such edges;
- if $e = 0$ propagation ends; otherwise, send $\left\lfloor \frac{m}{e} \right\rfloor$ toys through each edge.

We want to answer how many groups of exactly k toys will travel through the marked edge after all queries.

TUV – Tunnel Vision

Straightforward brute force – simulate the propagation.

For each query start at each leaf and propagate the information, e.g. using DFS.

$\mathcal{O}(n^2 g)$

TUV – Tunnel Vision

Let's try standardising the direction of propagation.

We're only interested in reaching the marked edge.

Split the marked edge by introducing a virtual vertex 0 and root the tree in it.

Now we start all queries at the leaves and always propagate up, as we're only interested in groups that reach 0.

TUV – Tunnel Vision

Second observation is that we can compress the tree.

Vertices that have degree exactly two are boring. We can contract them.

After we do that, the toy group always gets at least two times smaller when being pushed up the tree.

Naive brute-force from before now gets $\mathcal{O}(ng \log M)$, where $M = \max_{1 \leq i \leq g} m_i$.

TUV – Tunnel Vision

To get a faster solution consider what is the size of the package in a child of the root in order to count.

If the child has e edges then the minimum number of toys is $k(e-1)$.

The maximum is $k(e-1)+e-2$.

We can DFS from the root to calculate these intervals in the leaves.

TUV – Tunnel Vision

Now we have g groups and $\mathcal{O}(n)$ intervals $[x, y]$ and for each group we want to count in how many intervals it falls.

We can do it the other way around – for each interval count how many groups it catches.

This is easy to do with a binary-search.

We have $\mathcal{O}(n)$ for DFS, $\mathcal{O}(g \log g)$ to sort the groups, and $\mathcal{O}(n \log g)$ for searches, for a total of $\mathcal{O}((n + g) \log g)$.

FAL – Falling Cats

Here we are given a graph where each vertex has at most 4 edges.

There can be multiedges.

We remove edges from the graph one by one and need to answer, for each vertex, at which point it becomes disconnected from 1.

FAL – Falling Cats

Simplest idea – remove an edge, for each remaining vertex run any algorithm that tries to reach 1.

Definitely works, but takes $\mathcal{O}(mn^2)$.

FAL – Falling Cats

Better – after each edge run a search from 1 to find any reachable vertices.

Down to $\mathcal{O}(mn)$.

FAL – Falling Cats

Core idea – reverse time.

It's easier to detect when something gets connected to 1 than disconnected.

How? Find-union while maintaining time of connection to 1.

FAL – Falling Cats

1. Remove all m edges.
2. Use remaining edges to Union. Vertices connected to 1 never fall and get assigned -1.
3. For each removed edge in reverse order use it to Union with correct timestamp.

FAL – Falling Cats

How to maintain timestamps in Find-Union?

Simplest idea – use the worse Find-Union that maintains lists of vertices and merges shorter lists to larger lists, but also always merge to 1.

Whenever a vertex is moved to 1's component give it the current timestamp.

$\mathcal{O}(n \log n)$.

FAL – Falling Cats

We can augment the standard path-compressing Find-Union as well.

Note that we only need to add the timestamp to the root of the tree being Unioned.

During path compression we have to copy the first timestamp on the path to root.

$\mathcal{O}(n\alpha(n))$

FAL – Falling Cats

```
Find(x) {
  if x != P[x] {
      let p = Find(P[x])
      if Ans[x] == GROUND {
          Ans[x] = Ans[P[x]]
      }
      P[x] = p
  }
  return P[x]
}
```

Recall the plan

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

Graphs

Widespread data structure.

$G = (V, E)$

Network of vertices and edges between them.

Surprisingly many things can be modelled as a graph.

Usually no *multiedges* (E is a set).

Most of the time no *self-loops* ((v, v)).

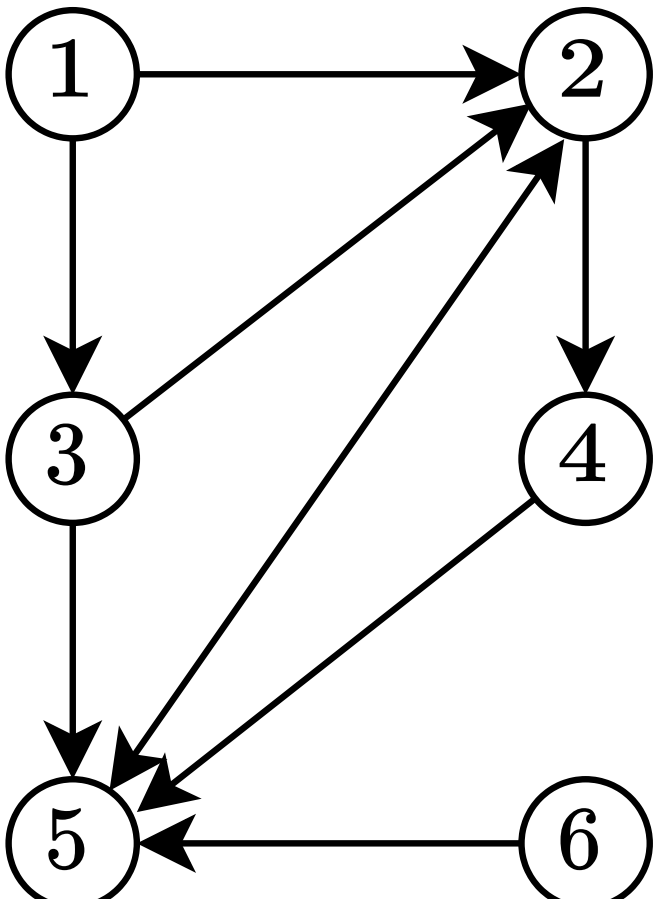
Graphs

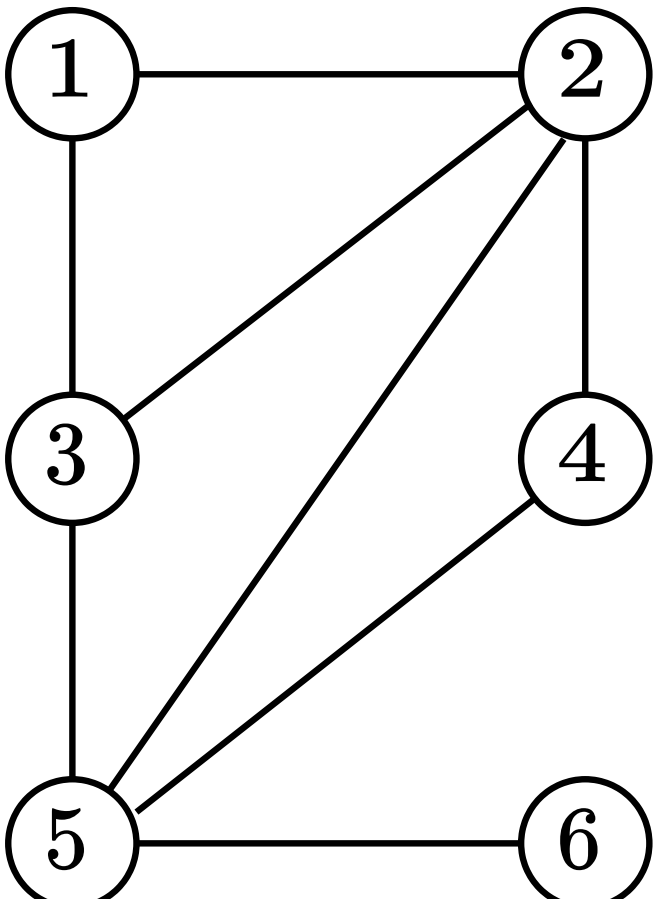
Multitude of different interpretations:

- travelling between vertices using edges, e.g. a map of cities and connections between them, road network where vertices are intersections, etc.
- edges are dependencies between vertices, e.g. this task has to be completed before this one;
- social network, e.g. this person follows this person.

Literally any binary relation can be a graph if you're brave enough.

Graphs – directed/undirected





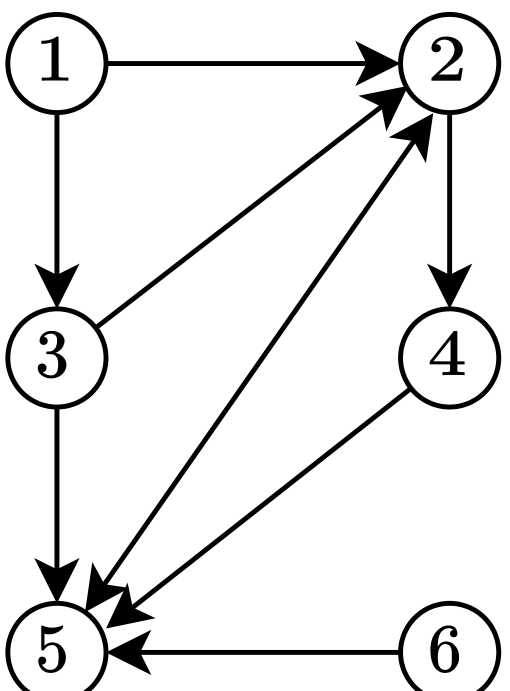
Graphs – directed/undirected

In practice, undirected simply means that $(u, v) \in E \Leftrightarrow (v, u) \in E$.

Graphs – input representation

As lists of children.

```
6 9
1 2
1 3
2 4
3 2
3 5
4 5
6 5
2 5
5 2
```



Graphs – in-memory representation

Lists of lists.

N[v] is a list of all neighbours of v.

In C++ a `vector<vector<int>>`.

In Rust a `Vec<Vec<u32>>`.

Graphs – in-memory representation

Adjacency matrix

N$[v][u]$ is 1 if and only if there is an edge (v, u).

In C++ a `vector<vector<bool>>` (bitset-optimised).

In Rust a `Vec<Vec<bool>>` (not optimised).

Graphs – paths, cycles

Path – sequence of vertices $v_1 \ldots v_n$ where there are edges between v_i, v_{i+1} **and none of the vertices repeat**.

Cycle – sequence of vertices $v_1 \ldots v_n$ where there are edges between v_i, v_{i+1} where $v_1 = v_n$ and **none of the other vertices repeat**.

Graphs – connected components

In an undirected graphs two vertices are in the same connected components when there exists a path between them.

We will assume a graph is connected (contains exactly one connected component) unless stated otherwise.

Most algorithms that work on a connected graph also work on a general one – run it on each component independently.

Graphs – DFS

The idea is the same as in a tree – visit each of the neighbours.

Difference is we have to keep track of visited vertices.

```
fn DFS(v)
  pre_process(v)
  for u in N[v]
    if not visited[u]
      visited[u] = true
      DFS(u)
  post_process(v)
```

Finding components

Run DFS once from a vertex – it will visit all vertices in that component.

Repeat it until all components are found.

```
c = 1
for v in V
  if C[v] == 0
    C[v] = c
    DFS(v)
    c += 1
```

```
fn DFS(v)
  for u in N[v]
    if C[u] == 0
      C[u] = C[v]
      DFS(u)
```

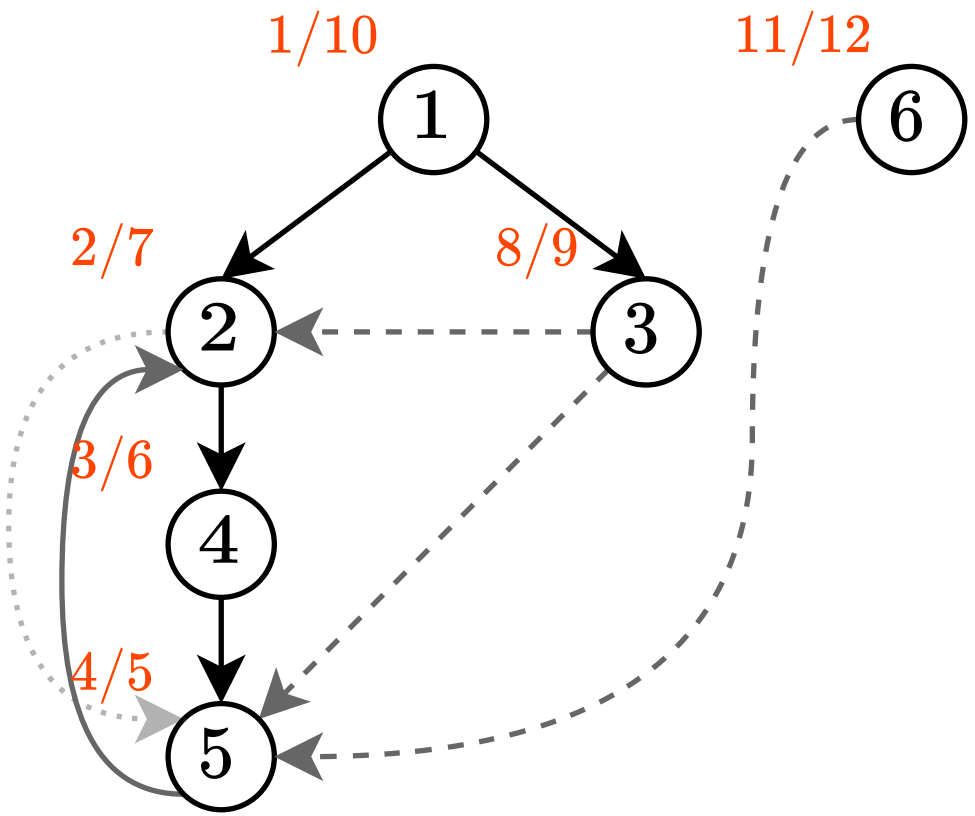
DFS tree structure

Tree edges – solid black.

Forward edges – dotted light grey.

Back edges – solid dark grey.

Cross edges – dashed dark grey.



Detecting cycles

Cycles are always back edges.

In a directed graph – any edge back to an *active* vertex.

In an undirected graph – any edge back to any visited vertex.

BFS – shortest paths

```
fn BFS(from)
  queue = Queue::new()
  queue.push(from)
  distance[from] = 0
  while v = queue.pop()
    for child in C[v]
      if distance[child] is None
        distance[child] = distance[v] + 1
        queue.push(child)
```

Graphs – weights

Edges can have *weights*.

These can have different meanings – cost of travel, time, distance…

Graphs – Dijkstra

BFS works when all weights are equal.

Dijkstra gives shortest paths **when all weights are non-negative**.

Find all paths from a given source s.

Graphs – Dijkstra

Algorithm is greedy.

Take the closest unprocessed vertex and try to extend paths from there.

We find the closest vertex with a priority queue.

Graphs – Dijkstra

```
fn Dijkstra(s)
  q = new MinHeap
  D[s] = 0
  q.push((0, s))
  while (d, v) = q.pop()
    if D[v] != d { continue; }
    for (u, cost) in N[v]
      new_d = d + cost
      if new_d < D[u]
        D[u] = new_d
        q.push((new_d, u))
```

Graphs – Dijkstra

In theory, Dijkstra can be implemented better with a heap that allows a DecreaseKey operation.

We get $\mathcal{O}(m \log n)$ time and $\mathcal{O}(n)$ memory then. With Fibonacci heap that has a fast DecreaseKey one can get $\mathcal{O}(m + n \log n)$.

The implementation we showed before is $\mathcal{O}(m \log m)$ time[1] and $\mathcal{O}(m)$ memory worst case since we can push the same vertex multiple times, but that's the best you can get using a standard library heap.

[1] But that's also equal to $\mathcal{O}(m \log n)$ since $m = \mathcal{O}(n^2)$ and $\log n^2 = 2 \log n = \mathcal{O}(\log n)$

Graphs – negative weights

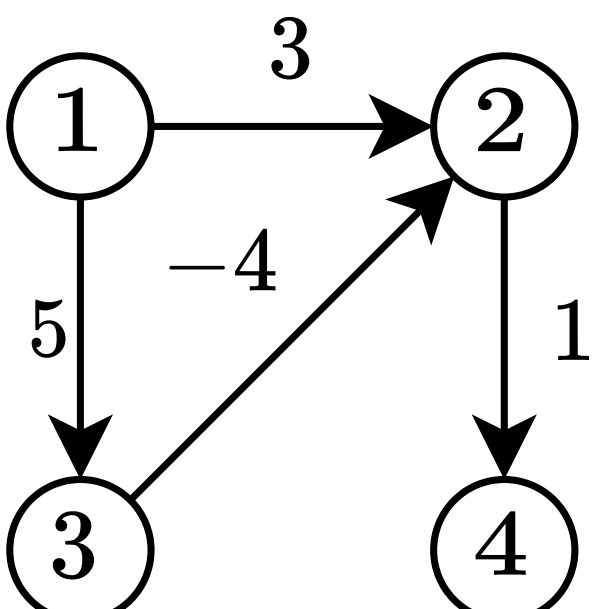
Dijkstra *does not* work with negative weights.

Correctness relies on the greedy property – once a vertex is popped from the queue its distance is already optimal and can never decrease.

Graphs – negative weights

Dijkstra *does not* work with negative weights.

Correctness relies on the greedy property – once a vertex is popped from the queue its distance is already optimal and can never decrease.



Graphs – Bellman-Ford

In presence of negative edges – Bellman-Ford.

Very simple idea – just relax all edges $n - 1$ times.

```
fn BellmanFord(s)
  D[s] = 0
  repeat n - 1 times
    for (v, u, d) in E
      new_d = D[v] + d
      if new_d < D[u]
        D[u] = new_d
```

Obvious $\mathcal{O}(nm)$ running time.

Graphs – Bellman-Ford

Why does this work?

Shortest path cannot have more than $n - 1$ edges.

Unless there is a negative-weight cycle in which case the result is garbage.

```
for (v, u, d) in E
  new_d = D[v] + d
  if new_d < D[u]
    raise "Negative cycle detected"
```

Graphs – Floyd-Warshall

DP for shortest paths between **all pairs** of vertices in $\mathcal{O}(n^3)$.

We skip it for time, but it's simple enough you can figure it out as an exercise.

Graphs – Minimal Spanning Tree

A *spanning tree* is a tree using edges from the graph that connects all vertices.

Always exists when graph is connected.

We are interested in the lowest total weight tree.

Graphs – Kruskal

Idea – greedily take the cheapest edge that connects two components.

```
fn Kruskal() {
  fu = new FindUnion
  mst = []
  edges.sort_by_weight()
  for (v, u, d) in E
    if fu.Find(v) != fu.Find(u)
       mst.push((v, u, d))
       fu.Union(v, u)
}
```

$\mathcal{O}(m \log m)$ to sort and $\mathcal{O}(m + n\alpha(n))$ for selection.

Graphs – Prim

Idea – maintain a subtree and expand it by picking the cheapest edge that goes to an unconnected vertex.

Graphs – Prim

```
fn Prim()
  queue = new MinHeap
  W[1] = 0
  visited[1] = true
  queue.push((0, 1))
  while (w, v) = queue.pop()
    visited[v] = true
    if W[v] != w { continue }
    for (c, u) in N[v]
      if not visited[u] and c < W[u]
        W[u] = c
        P[u] = v
        queue.push((c, u))
```

Graphs – Prim

The above implementation is the same as Kruskal, $\mathcal{O}(m \log m)$.

However, in theory using Fibonacci heaps like in Dijkstra gives $\mathcal{O}(m + n \log n)$.

Graphs – bridges and articulation points

A *bridge* is any edge in a connected graph whose removal would disconnect it.

An *articulation point* is any vertex whose removal would do so.

If (v, u) is a bridge then v and u are articulation points (unless v and u have no other edges).

The other direction is not true.

Graphs – biconnected components

A *biconnected component* is a maximal subgraph that has no articulation points.

Any graph can be decomposed into a tree of biconnected components.

Easy to find articulation points and bridges in such a tree.

Graphs – biconnected components

```
fn ArticulationPoints() {
  time = 0;
  for u in V
    if pre[u] is None
      ArtDFS(u, u)
      is_art[u] = dfs_deg[u] > 1
}
```

Graphs – biconnected components

```
fn ArtDFS(v, p) {
  low[v] = pre[v] = time
  dfs_deg[v] = 0
  time += 1
  for u in N[v] {
    if u == p { continue }
    if pre[u] is None
      dfs_deg[v] += 1
      dfsAP(u, v);
      if pre[v] <= low[u] { is_art[v] = true }
      low[v] = min(low[v], low[u]);
    else
```

```
        low[v] = min(low[v], pre[u])
}
```

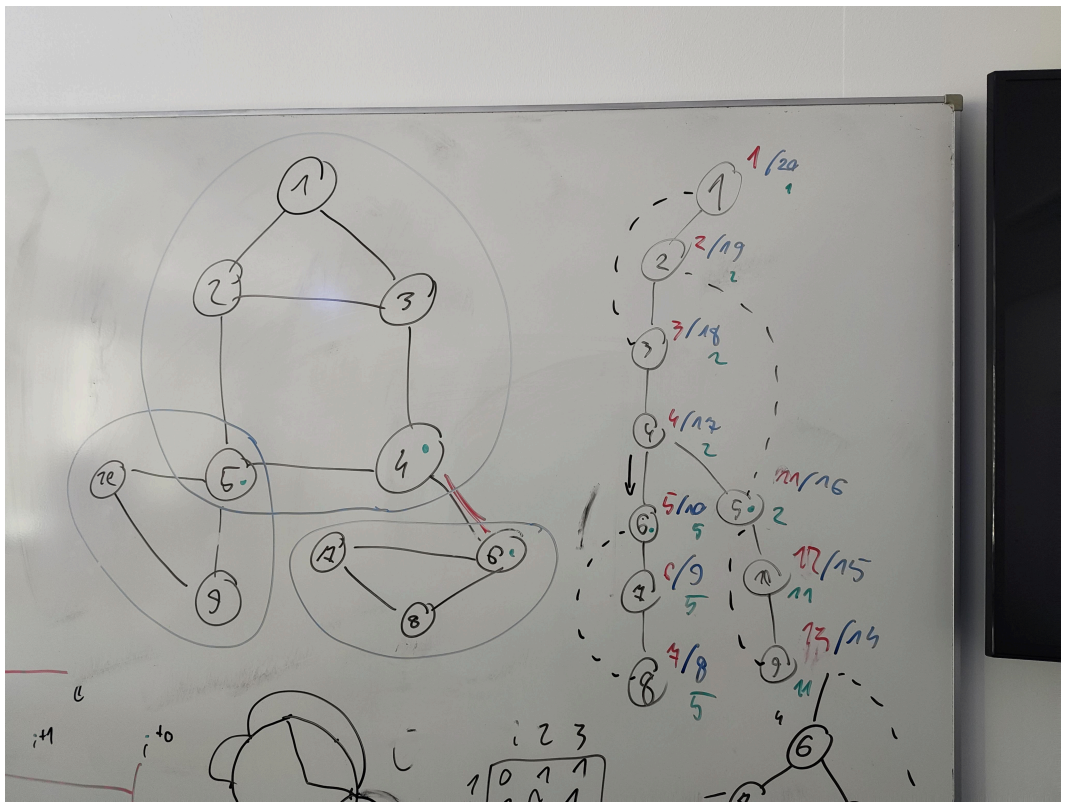
Graphs – biconnected components

How to find bridges?

One line change – the condition is `pre[v] < low[u]` for (v, u) to be a bridge.

The same algorithm works with depth instead of preorder times.

Graphs – biconnected components



Graphs – biconnected components

Some properties:

- any biconnected component can be decomposed to a cycle and then additional paths that connect vertices from the cycle or earlier paths;[2]
- if an articulation point belongs to two biconnected components, any path between two vertices in different components passes through that vertex;

[2] This is called an *ear decomposition*. See: https://en.wikipedia.org/wiki/Ear_decomposition

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).

These are important. We'll talk about them next time 😉 .

See you in two weeks

PAW and CAT: 17.06.2025, 10:00 AM

Good luck!