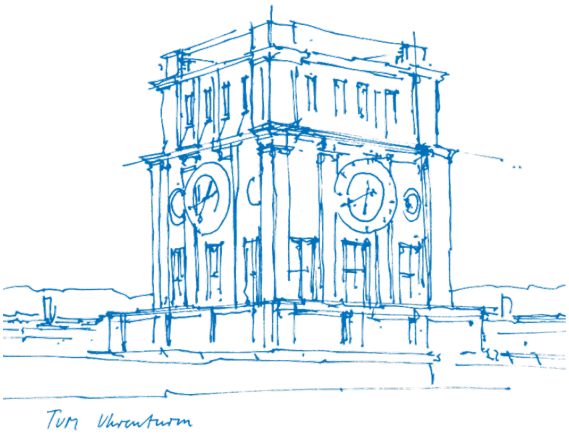
Please adjust group name in tumuser.sty.
Department of Electrical and Computer Engineering
Technical University of Munich

AACPP 26/27

Class 3: Graphs

Julia Spindler
Mateusz Gienieczko

November 12, 2026



Third problem

Third deadline – 26.11.2026, 12:00 PM



We want to find out what the maximum of gold is we can get, splitting up our hirelings.
As a DP problem:

$$f(0) = 0\,, \qquad f(i) = \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big), \qquad \text{answer } f(n)$$

We want to find out what the maximum of gold is we can get, splitting up our hirelings.
As a DP problem:

$$f(0) = 0, \qquad f(i) = \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big), \qquad \text{answer } f(n)$$

- 0 hirelings $\rightarrow$ 0 gold

RAI

We want to find out what the maximum of gold is we can get, splitting up our hirelings.
As a DP problem:

$$f(0) = 0, \qquad f(i) = \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big), \qquad \text{answer } f(n)$$

- 0 hirelings $\rightarrow$ 0 gold
- we want to maximize the amount of gold at hireling i using the past maxima we already computed

RAI

We want to find out what the maximum of gold is we can get, splitting up our hirelings.
As a DP problem:

$$f(0) = 0, \qquad f(i) = \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big), \qquad \text{answer } f(n)$$

- 0 hirelings $\rightarrow$ 0 gold
- we want to maximize the amount of gold at hireling i using the past maxima we already computed
- each hireling has power x_i and the amount of gold depends on the sum of the power of all of them in a raiding party

RAI - $O(n^2)$

```
def max_gold(a, b, c, x):
    n = len(x)
    S = [0] * (n + 1)                # prefix sums to avoid another loop
    for i in 1..n:
        S[i] = S[i - 1] + x[i - 1]

    f = [0] + [-INF] * n
    for i in 1..n:                   # last party ends at i
        for j in 0..i-1:             # last party is j+1 .. i
            s = S[i] - S[j]
            f[i] = max(f[i], f[j] + a * s * s + b * s + c)
    return f[n]
```

RAI - convex hull trick

$$f(i) = \max_{0 \le j < i} \Big(f(j) + g\big(\Sigma_{t=j+1}^{i} x_t\big) \Big)$$

RAI - convex hull trick

Prefix sums: $S_0 = 0, \quad S_i = x_1 + \cdots + x_i \quad \Rightarrow \quad \sum_{t=j+1}^{i} x_t = S_i - S_j$

$$
\begin{aligned}
f(i) &= \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + a(S_i - S_j)^2 + b(S_i - S_j) + c \Big)
\end{aligned}
$$

RAI - convex hull trick

Prefix sums: $S_0 = 0, \quad S_i = x_1 + \cdots + x_i \quad \Rightarrow \quad \sum_{t=j+1}^{i} x_t = S_i - S_j$

$$
\begin{aligned}
f(i) &= \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + a(S_i - S_j)^2 + b(S_i - S_j) + c \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + aS_i^2 - 2aS_iS_j + aS_j^2 + bS_i - bS_j + c \Big)
\end{aligned}
$$

RAI - convex hull trick

Prefix sums: $S_0 = 0, \quad S_i = x_1 + \cdots + x_i \quad \Rightarrow \quad \sum_{t=j+1}^{i} x_t = S_i - S_j$

$$
\begin{aligned}
f(i) &= \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + a(S_i - S_j)^2 + b(S_i - S_j) + c \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + aS_i^2 - 2aS_iS_j + aS_j^2 + bS_i - bS_j + c \Big) \\
&= aS_i^2 + bS_i + c \; + \; \max_{0 \le j < i} \Big(-2aS_j \cdot S_i \; + \; f(j) + aS_j^2 - bS_j \Big)
\end{aligned}
$$

RAI - convex hull trick

Prefix sums: $S_0 = 0, \quad S_i = x_1 + \cdots + x_i \quad \Rightarrow \quad \sum_{t=j+1}^{i} x_t = S_i - S_j$

$$
\begin{aligned}
f(i) &= \max_{0 \le j < i} \Big(f(j) + g\big(\textstyle\sum_{t=j+1}^{i} x_t\big) \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + a(S_i - S_j)^2 + b(S_i - S_j) + c \Big) \\
&= \max_{0 \le j < i} \Big(f(j) + aS_i^2 - 2aS_iS_j + aS_j^2 + bS_i - bS_j + c \Big) \\
&= \underbrace{aS_i^2 + bS_i + c}_{\text{depends only on } i} + \max_{0 \le j < i} \Big(\underbrace{-2aS_j}_{m_j} \cdot S_i + \underbrace{f(j) + aS_j^2 - bS_j}_{k_j} \Big)
\end{aligned}
$$

RAI - $O(n)$

```
S = prefix sums of x
f[0] = 0
dq = [ line(m=0, k=0) ]                          # j = 0

def useless(l1, l2, l3):
    return (l3.k - l1.k) * (l1.m - l2.m) <= (l2.k - l1.k) * (l1.m - l3.m)

for i in 1..n:
    q = S[i]
    while len(dq) >= 2 and dq[1](q) >= dq[0](q):
        dq.pop_front()                           # front line is outdated
    f[i] = aq + bq + c + dq[0](q)

    new = line(m = -2aq,  k = f[i] + aq - bq)
    while len(dq) >= 2 and useless(dq[-2], dq[-1], new):
        dq.pop_back()                            # last line is never on top
    dq.push_back(new)
```

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.

Trees

A *tree* is a connected (undirected) graph without cycles.

Equivalent characterisations (any one implies the rest):

- connected graph on n vertices with $n - 1$ edges;
- exactly one path between each pair of vertices;
- removing any edge disconnects it, adding any edge creates a cycle.

Trees – Root, Parent, Child, Depth

Mathematically trees don't have to be rooted, but as a data structure they are.

Root – unique distinguished vertex.

Parent – unique node "up"; root has none.

Child – direct neighbour other than the parent.

Depth – root has depth 0, children of a node with depth d have depth $d + 1$.

Leaves – nodes with no children.

Trees – Ancestor, Descendant

Transitive closure of the parent relation is called *ancestor*.

Of the child relation – *descendant*.

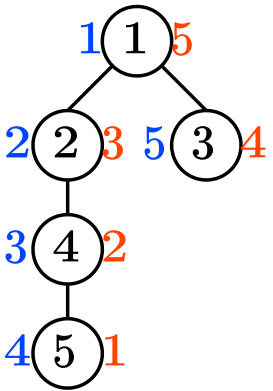
Trees – Forests

Disconnected trees are called a *forest* (or an acylic but disconnected graph is a forest).

Trees – DFS preorder and postorder

Preorder – give a number when entering.

Postorder – give a number when exiting.



Trees – Lowest Common Ancestor

$\mathrm{LCA}(u, v)$ is the deepest vertex that is an ancestor of both u and v (a vertex counts as its own ancestor).

Intuition: walk up from u and from v until the two walks meet – the meeting point is the LCA.

Trees – LCA with log-jumps

We can precompute jumps to speed this up.

Preprocessing: compute $\mathrm{depth}[v]$ and $\mathrm{jump}[v][k]$, the ancestor 2^k levels above v.

Trees – LCA with log-jumps

We can precompute jumps to speed this up.

Preprocessing: compute $\mathrm{depth}[v]$ and $\mathrm{jump}[v][k]$, the ancestor 2^k levels above v.

Query $\mathrm{LCA}(u, v)$, with $\mathrm{depth}[u] \geq \mathrm{depth}[v]$:

1. Lift u to the depth of v, using the jump table.
2. If $u = v$, we are done, v is the LCA
3. For k from high to low: if $\mathrm{jump}[u][k] \neq \mathrm{jump}[v][k]$, move both u and v up by 2^k.
4. The answer is $\mathrm{parent}[u]$.

Trees – Centroid Decomposition

Method for running divide-and-conquer algorithms on trees.

Idea – take a vertex and split the problem across its children.

Subproblems happen entirely in the subtrees, or involve the pivot vertex.

Effective if the pivot splits into “small” subtrees.

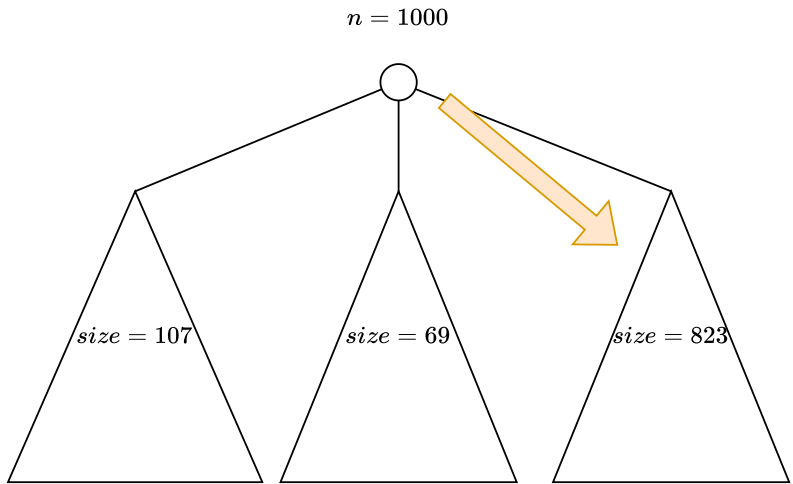
Trees – Centroid Decomposition

Centroid of a tree on n vertices is a vertex whose subtrees all have at most $\frac{n}{2}$ vertices.

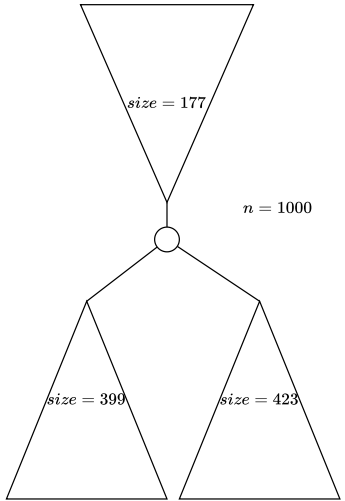
Provably there is exactly one or two such vertices.

Algorithm to find: root the tree arbitrarily and compute subtree sizes. Start at the root. While some child c of the current vertex has $\mathrm{size}[c] > \frac{n}{2}$, move to c. The vertex where we stop is a centroid.

Trees – Centroid Decomposition



Trees – Centroid Decomposition



DFS tree structure

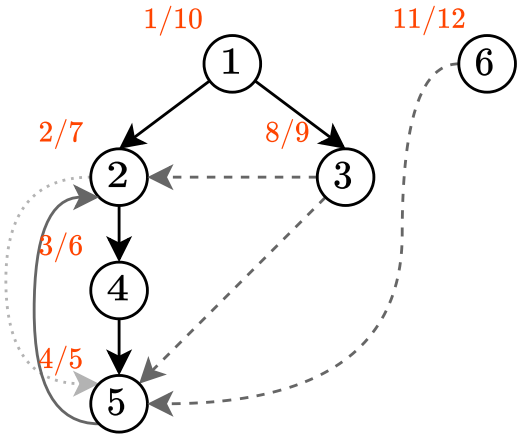
Run DFS on a directed graph and look at each edge $u \to v$:

Tree – DFS discovers v through this edge. Solid black.

Forward – v is a descendant of u, but not a tree edge. Dotted light grey.

Back – v is an ancestor of u (still active). Solid dark grey.

Cross – v is in another branch and already finished. 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.

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 either v and u have no other edges).

The other direction is not true.

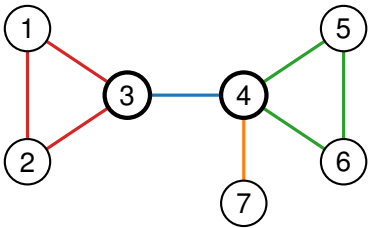
Graphs – biconnected components

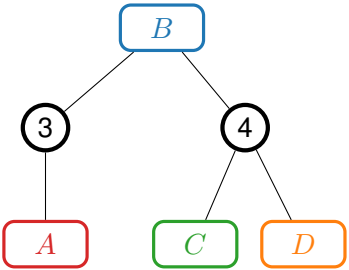
A *biconnected component* is a maximal subgraph that has no articulation points.

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





Graphs – biconnected components

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

Graphs – biconnected components

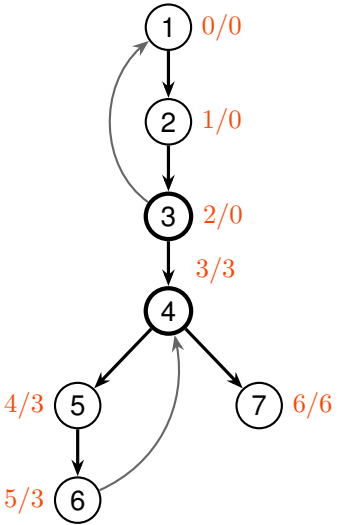
```
fn dfsAP(v, p) {
  low[v] = pre[v] = time
  dfs_deg[v] = 0
  time += 1
  for u in N[v] {
    if u == p { continue }
    if pre[u] is None
      dfs_deg[v] += 1
      dfsAP(u, v);
      if pre[v] <= low[u] { is_art[v] = true }
      low[v] = min(low[v], low[u]);
    else
      low[v] = min(low[v],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

DFS tree of the articulation point example

Tree edges – solid black.

Back edges – solid dark grey.

Orange labels: $\mathrm{pre}[v]/\mathrm{low}[v]$.



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

Some properties:

- any biconnected component can be decomposed to a cycle and then additional paths that connect vertices from the cycle or earlier paths;[1]
- if an articulation point belongs to two biconnected components, any path between two vertices in different components passes through that vertex;

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

Graphs – strongly connected components

In a *directed* graph, “reachable” is not symmetric: $u \to v$ does not give a way back from v to u.

Instead: u and v are in the same *strongly connected component* (SCC) if there is a directed path from u to v *and* one from v to u.

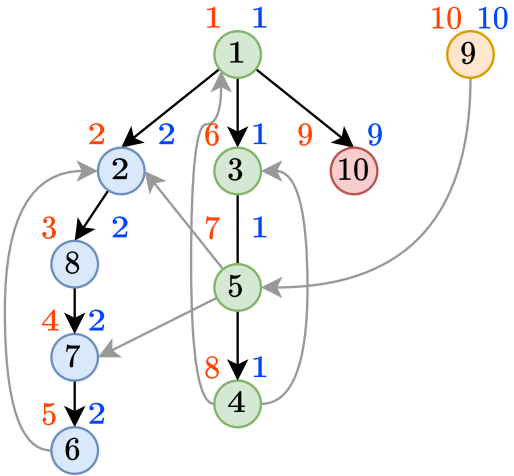
SCCs are the maximal such sets, and they partition the vertices.

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 – finding SCCs



Graphs – DAGs and DP

DAGs are very nice for DP.

Information flows along the edges – obvious “direction” of recurrence.