Graphs

- **(Undirected) Graph**: $G = (V, E)$

V = vertices = networks units

E = edges

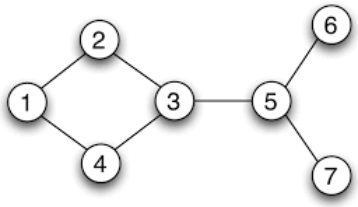
- **Directed Graph**: $\overrightarrow{G} = (V, A)$

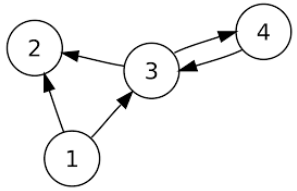
A = arcs

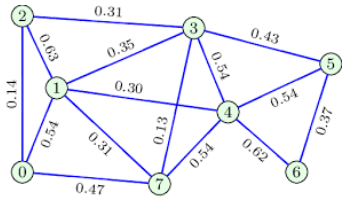
- **Weighted Graph**: $G = (V, E, w)$

$w : E \to \mathbb{R}$ is a weight function (length, cost)

There may be loops, multiple edges (= Multigraphs), vertex-weights, etc.







Terminology

- If $e = uv$ is an edge, then vertices u and v are **adjacent**, and they are **incident** to edge e.

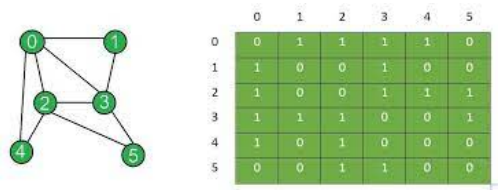
- The **degree** $d(v)$ of a vertex v equals its number of incident edges (loops are counted twice!).

- The (open) **neighbourhood** of a vertex v is the set $N(v)$ of all its adjacent vertices. The **closed neighbourhood** is defined as $N[v] = N(v) \cup \{v\}$.

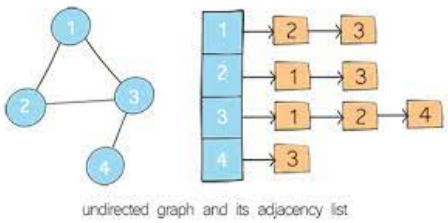
- The neighbourhood of a set M equals $N(M) = \bigcup\{N(v) \setminus M \mid v \in M\}$. The closed neighbourhood is defined as $N[M] = N(M) \cup \{M\}$.

Graph representations

- **Adjacency matrix**



- **Adjacency list**



Comparison

Throughout ALL this semester:

- **n = number of vertices;**
- **and m = number of edges.**

- Adjacency matrix requires $\mathcal{O}(n^2)$ space

- Adjacency list requires $\mathcal{O}(n + m)$ space, which is optimal.

Remark 1: $m \leq \binom{n}{2} = \mathcal{O}(n^2)$, therefore adjacency list is *never worse* than adjacency matrix.

Remark 2: if there is no isolated vertex, $m \geq n/2$, and therefore we can simplify $\mathcal{O}(n + m) = \mathcal{O}(m)$.

Basic operations on graphs

- Enumerate all edges
- Enumerate all neighbours of a vertex
- Output the degree of a vertex
- Decide whether two vertices are adjacent
- Addition/Removal of edges/vertices.
- **Contraction** of an edge $e = uv$: replace u, v by some new vertex x whose neighbourhood equals $N(u) \cup N(v) \setminus \{u, v\}$.

With Adjacency matrix

- Enumerate all edges: scan the whole matrix in $\mathcal{O}(n^2)$ time.
- Enumerate all neighbours of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.
- Output the degree of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.

  Better: $\mathcal{O}(1)$ time if we store the degree in a separate variable.
- Decide whether two vertices are adjacent: Look at the corresponding cell in the matrix in $\mathcal{O}(1)$ time.
- Addition/Removal of edges: Modify the two corresponding cells in $\mathcal{O}(1)$ time.

With Adjacency matrix (cont'd)

- Addition/Removal of a vertex: Reallocate the actualized matrix in $\mathcal{O}(n^2)$ time.

  Better: use doubling arrays, which support addition/removal of elements at front/back of the vector in amortized $\mathcal{O}(1)$ time.

  $\rightarrow$ Addition of a new vertex requires inserting a new bottom line: in amortized $\mathcal{O}(n)$ time, and a new cell at the end of each previous line: in amortized $n \times \mathcal{O}(1) = \mathcal{O}(n)$ time.

  $\rightarrow$ For vertex removal: swap the corresponding line and column with the *last* line and column of the matrix. It can be done in $\mathcal{O}(n)$ time. Then, delete the last line and column in amortized $\mathcal{O}(n)$ time.
  Requires reference updates for the vertices.
- **Contraction** of an edge $e = uv$: Reduces to two vertex removals, one vertex addition, and up to $\mathcal{O}(n)$ edge additions.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n + m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.
- Enumerate all neighbours of a vertex v: scan the corresponding list in $\mathcal{O}(d(v))$ time.
- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.
- Addition of a new vertex: insertion of a new empty list at the end in amortized $\mathcal{O}(1)$ time.
- Removal of an isolated vertex v: Switch v with the last vertex, then remove the last vertex, in amortized $\mathcal{O}(1)$ time.

Adjacency Testing

For static graphs (no vertex/edge modifications allowed), we may replace adjacency list by adjacency vectors.

If adjacency vectors are **sorted**, then we can test whether u, v are adjacency in $\mathcal{O}(\log \min\{d(u), d(v)\})$ time, using *binary search*.

Sorting all adjacency lists can be done naively in $\mathcal{O}(n + m \log n)$ time.

If vertices are labeled $0, 1, \ldots, n-1$, then we can sort adjacency lists in $\mathcal{O}(n + m)$ time and space:

- Create a new graph G' with no edges.
- For $i = 0 \ldots n-1$, traverse the adjacency list of vertex i in the original graph G. For every adjacent vertex j, add i to j's adjacency list in the copy G'.

Reminder: Hash Table

Store a collection of pairs `(key,value)`.

- Three operations:
  - **int** `lookup(e);` Returns the value associated to some key e (if it is present in the table)
  - **void** `insert(e, v);` Adds a new pair `(e,v)` (if the key e is not already present in the table)
  - **void** `delete(e);` Deletes a pair `(key,value)` – given its key e.

Each operation runs in expected $\mathcal{O}(1)$ time.

Applications to Adjacency Testing

We store every edge in a Hash table. To every edge uv, we associate pointers its positions in the respective adjacency lists of u and v.

- Adjacency testing: Lookup in the table in expected $\mathcal{O}(1)$ time.
- Addition of an edge uv: Insertion of u (v, resp.) at the bottom of the adjacency list of v (u, resp.). Insertion in expected $\mathcal{O}(1)$ time in the Hash table, along with pointers to the bottom positions in the respective adjacency lists of u and v.
- Removal of an edge uv: Lookup in the Hash-table in order to find the positions of u, v in the respective adjacency lists of v, u. Then, removal in both adjacency lists and in the table.
- Removal of a vertex: removal of every incident edge + removal of an isolated vertex.

Tentative schedule for this semester

- Graph searches.
- Cographs.
- Modular decomposition.
- Partitive families and their applications to Graph decomposition.
- Chordal graphs.
- Clique-separator decomposition.
- Tree decompositions.
- Parameterized complexity (mostly, Treewidth).
- Courcelle's theorem and relatives.
- Perfect graphs and Linear Programming.
- Planar graphs.
- Dynamic graph algorithms.

Advanced Graph Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Graph search

Given a graph $G = (V, E)$, a (connected) graph search with start vertex v_{n-1} consists of a total ordering $v_{n-1}, v_{n-2}, \ldots, v_0$ of the vertices, under the following rule:

- for every $1 \leq i \leq n-1$, let $V_i = \bigcup_{j=i}^{n-1} N(v_j) \setminus \{v_i, v_{i+1}, \ldots, v_{n-1}\}$. If $V_i \neq \emptyset$, then $v_{i-1} \in V_i$.

$\rightarrow$ "Choose any neighbour of an already visited vertex as the next vertex to be visited."

Today's objectives:

- Review of classical graph searches (DFS, BFS, and more).
- Implementation
- Properties

Graph search vs. Spanning forests

Observation: in any graph search, all vertices in a same connected component must be consecutive.

Consequences:

- *We can list all connected components*: a new component starts each time we visit a vertex v_{i-1} with no neighbours in $\{v_i, v_{i+1}, \ldots, v_{n-1}\}$.
- We can construct a *spanning tree* for every connected component: if a vertex v_i is in the same connected component as v_{i-1}, choose any of its neighbours in $\{v_i, v_{i+1}, \ldots, v_{n-1}\}$ as its father node.

Complexity: Time to execute the search $+ \mathcal{O}(n+m)$.

Algorithmic applications

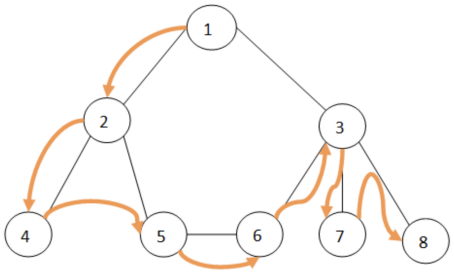
In order to solve some fundamental graph problems, we only need to compute a spanning tree for each connected component.
Examples:

- Acyclicity: test whether all edges of the graph belong to one of the spanning trees.
- Bipartite: compute the unique bipartition of every spanning tree and check whether it is also valid for the whole graph.
- 2-edge connectivity: for any edge between a vertex v and its parent u, let us denote by T_v the spanning subtree rooted at v. Then, uv is a cut-edge (i.e., it disconnects the graph) if and only if there is no edge xy such that $x \in V(T_v),\ y \notin V(T_v)$.

  $\rightarrow$ Keep track, for each xy not in the spanning forest, of the least common ancestor of x, y.

DFS

Pick the **most recently visited** vertex with at least one neighbour unvisited. Then, go to an arbitrary unvisited neighbour of this vertex.



Equivalently: either continue the search to any neighbour of the current vertex (if possible) or backtrack to the father node in the search tree.

Implementation

At any moment during the execution of the algorithm, we **keep in a stack** the path from the start vertex v_{n-1} to the current vertex v_i.

`S := {}`

`S.push(`v_{n-1}`)`

`Visit` v_{n-1}

while `!S.empty():`
    u `:= S.top()` *//current vertex*
    if `there exists some` $v \in N(u)$ `unvisited:`
        `S.push(`v`)`
        `Visit` v
    else: `S.pop()`

Complexity: $\mathcal{O}(n+m)$

Palm tree

Definition (Palm tree)

A spanning tree output by DFS.

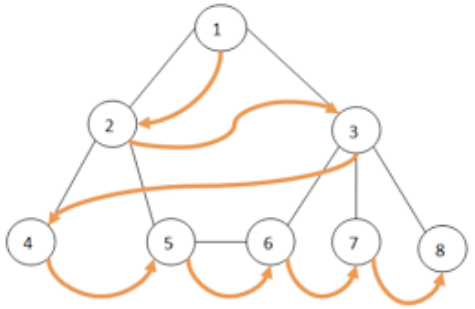
Characterization: A spanning tree of a connected graph G is a palm tree if and only if every *backward edge* xy (i.e., not in the spanning tree) satisfies that x is an ancestor of y, or y is an ancestor of x.

Consequence: one can decide in $\mathcal{O}(n+m)$ time whether a spanning tree is a palm tree.

Application: simpler algorithm to compute all cut-edges in $\mathcal{O}(n+m)$ time (and other related problems such as strongly connected components, etc.).

BFS

Pick the **least recently visited** vertex with at least one neighbour unvisited. Then, go to an arbitrary unvisited neighbour of this vertex.



Implementation

We **keep in a queue** the next vertices to be visited, in order.

```
Q := {}
Q.enqueue(v_{n-1})
visited[v_{n-1}] := True
while !Q.empty():
    u := Q.dequeue() //current vertex
    Visit u
    for all v ∈ N(u):
        if !visited[v]:
            Q.enqueue(v)
            visited[v] := True
```

Complexity: $\mathcal{O}(n+m)$

Shortest path tree

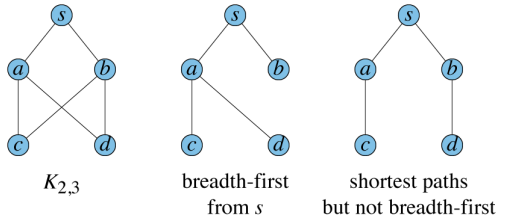
If G is connected, then we can define the distance $d_G(u,v)$ = minimum number of edges on a uv-path.

Definition (Shortest-path tree)

Rooted spanning tree (T,r) such that $d_G(r,v) = d_T(r,v)$ for every $v \in V(G)$.

Property: every output of a BFS is a shortest-path tree.

The converse is false:



$K_{2,3}$ | breadth-first from s | shortest paths but not breadth-first

Recognition of BFS trees

Let (T,r) be a shortest-path tree of G, and let xy be a backward edge (of $E(G) \setminus E(T)$). Edge xy is either:

- horizontal: $d(x,r) = d(y,r)$
- vertical: $d(x,r) = d(y,r) - 1$.

Observation: only vertical edges matter.

Manber's property: if $z = \texttt{lca}(x,y)$ and a_x, a_y are the respective ancestors of x, y such that $d(a_x,r) = d(a_y,r) = d(z,r)+1$, then we must visit a_y before a_x in the BFS.

$\rightarrow$ Add an arc (a_x, a_y). For every level, the resulting directed graph must be a DAG! Any topological ordering indicates in which order we should visit vertices of this level, with the additional properties that all children of a same node must be visited consecutively.

Layering search

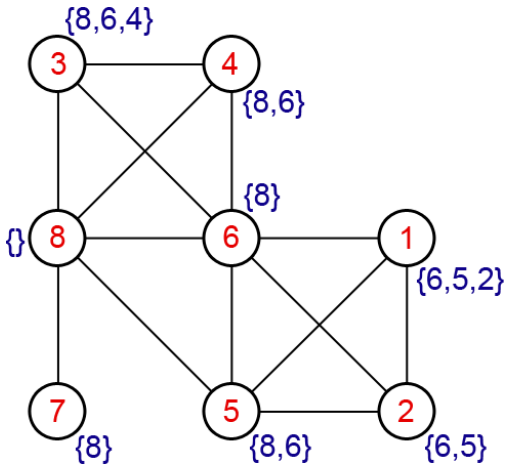
A weakening of BFS: Pick a **closest to the root visited** vertex with at least one neighbour unvisited. Then, go to an arbitrary unvisited neighbour of this vertex.

Proposition

A spanning tree is a shortest-path tree if and only if it is the output of a layering search.

LexBFS

The visited neighbours of each vertex are ordered by decreasing label. At every step, the next vertex to be visited must have a label which is **lexicographically** maximum.



Partition Refinement

• Data Structure that maintains an ordered collection of pairwise disjoint sets, subject to the following basic operations:

- `init(V)`: initialize the structure with one set, equal to V.
- `refine(S)`: for each set X such that $X \cap S \neq \emptyset$ and $X \setminus S \neq \emptyset$, we replace X by the two consecutive new sets $X \cap S$ and $X \setminus S$.

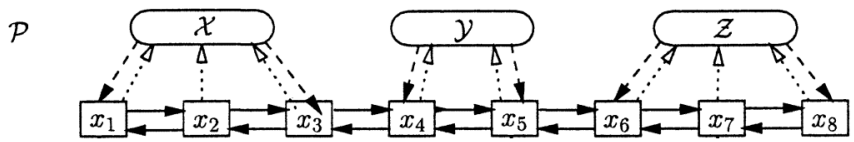
• Operation `init(V)` is in worst-case $\mathcal{O}(|V|)$. Each operation `refine(S)` is in worst-case $\mathcal{O}(|S|)$. Note that these are optimal runtimes!

Implementation

• Elements in V are maintained in a doubly-linked list $\mathcal{L}$, such that all elements in a same set X are consecutive.

• Each set X of the partition is represented by a structure with two fields: pointers to its first and last elements in $\mathcal{L}$.

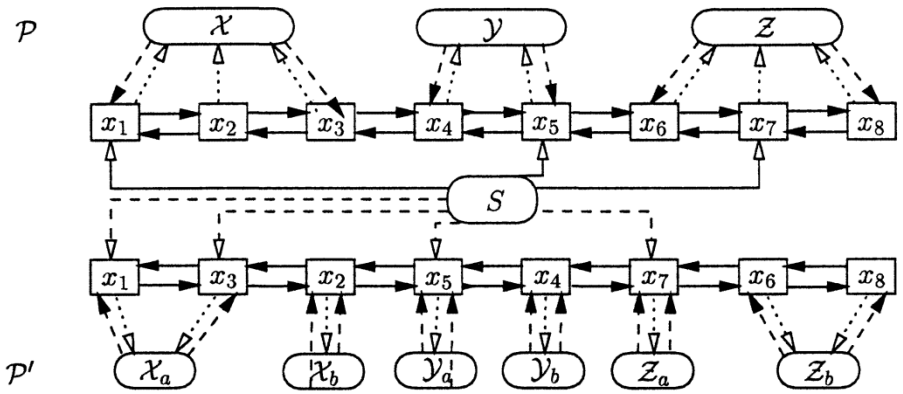
• Each node in the list $\mathcal{L}$ stores a pointer to the set X which contains it.



Refinement

- To each set X, "lazily" associate an empty list $L[X]$ (we effectively create the list only if it needs to be accessed at some point during the execution of the algorithm).
- For each $s \in S$, access to its set X and add a pointer to s in $L[X]$. Put a pointer to X in an auxiliary list $\mathcal{H}$ (the sets of $\mathcal{H}$ are those intersecting S).
- For each set X in $\mathcal{H}$, if $L[X] \neq X$, then:
  - Update the first and last element of X as its first and last elements not in S (forward/backward search in $\mathcal{L}$).
  - Remove all elements in $L[X]$ from $\mathcal{L}$;
  - Reinsert $L[X]$ immediately before the first element of X (or immediately after the last element of X);
  - Create a new set from $L[X]$.

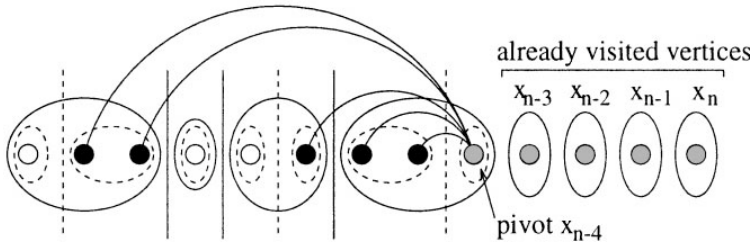
Refinement: illustration



Application to LexBFS

Traverse backward the list of all vertices – with the start vertex v_{n-1} at the end.

Repeatedly visit the next vertex v_i and refine the unvisited vertices according to $N(v_i)$.



Complexity: $\mathcal{O}(n+m)$

LexBFS-tree

Theorem (Beisegel et al., 2019)

Deciding whether a spanning tree is the output of a LexBFS is NP-complete.

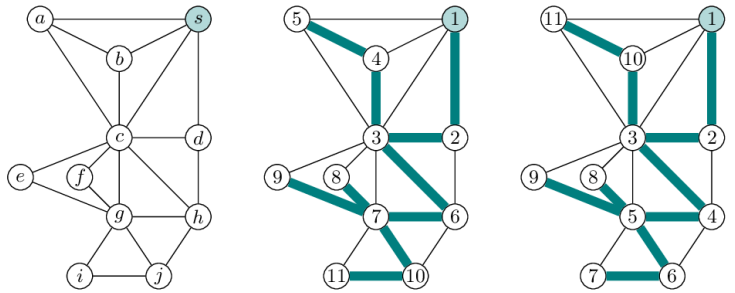
Tightly related to the following end-vertex problem:

- Given a vertex v in a graph G, does there exist a graph search where we visit v last?

The end-vertex problem is NP-complete for LexBFS, but also for DFS, BFS, etc.

LexDFS

The vertices are labeled from 1 to n (before they were labeled from $n-1$ to 0). As before, the visited neighbours of each vertex are ordered by decreasing label. At every step, the next vertex to be visited must have a label which is lexicographically maximum.



Open: existence of an $\mathcal{O}(n+m)$-time implementation? Spinrad claimed an $\mathcal{O}(n+m\log\log n)$-time implementation.

Implementation in $\mathcal{O}(n + m \log n)$ time

1) Traverse forward the list of all vertices – with the start vertex v_1 at the front.

2) After i steps, unvisited vertices are grouped by decreasing labels $X_{i,1}, X_{i,2}, \ldots, X_{i,q_i}$ such that $lab(X_{i,1}) \geq lab(X_{i,2}) \geq \ldots \geq lab(X_{i,q_i})$.

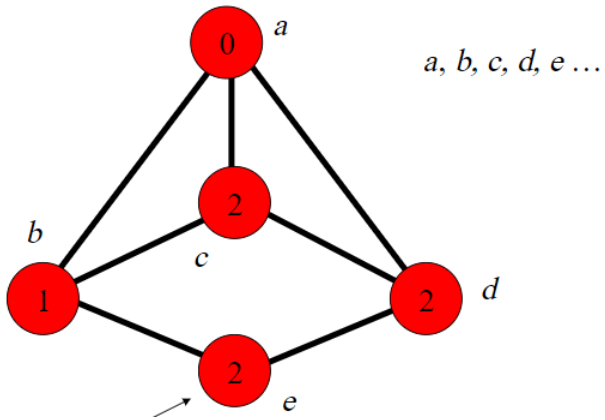
3) We refine: $X_{i,1} \cap N(v_i), X_{i,1} \setminus N(v_i), X_{i,2} \cap N(v_i), X_{i,2} \setminus N(v_i), \ldots, X_{i,q_i} \cap N(v_i), X_{i,q_i} \setminus N(v_i)$.

4) The intersections with $N(v_i)$ are pushed to the left: $X_{i,1} \cap N(v_i), X_{i,2} \cap N(v_i), \ldots, X_{i,q_i} \cap N(v_i), X_{i,1} \setminus N(v_i), X_{i,2} \setminus N(v_i), \ldots, , X_{i,q_i} \setminus N(v_i)$.

$\rightarrow$ **This has to be done while respecting the group orders**. If all groups $X_{i,j}$ are labeled by decreasing integers $\ell_{i,1} > \ell_{i,2} > \ell_{i,3} > \ldots > \ell_{i,q_i}$ then it can be done in $\mathcal{O}(d(v_i) \log n)$ time by sorting.

MCS

At every step, the next vertex to be visited must have a **maximum number of visited neighbours**.



In general, a MCS is neither a BFS nor a DFS. However, it has common properties with both LexDFS and LexBFS.

Implementation

A **frequency queue** stores a collection of repeated elements, ordered by their number of repetitions.

Two operations:

- insertion of a new element (if the element was already present, its number of repetitions increases).
- output and removal of any element with maximum number of repetitions.

Implementation: we store a list of lists $[L[i_0], L[i_1], \ldots, L[i_q]]$ such that all elements repeated exactly i_j times are stored in list $L[i_j]$, and $i_0 < i_1 < \ldots < i_q$. A pointer to the position of each element in $L[i_j]$ is stored in a separate Hash table.

$\rightarrow$ Every operation in expected $\mathcal{O}(1)$ time.

Application to MCS

FQ := empty frequency queue.

Insert v_{n-1} in FQ.

While FQ is nonempty:

1) Output a vertex v_i with maximum frequency (removed from FQ).

2) Visit v_i

3) Insert all neighbours of v_i in FQ (if a neighbour was already present, increase its number of repetitions).

Complexity: in expected $\mathcal{O}(n + m)$ time.

MNS

Every unvisited vertex stores the set of all its visited neighbours. At every step, the next vertex to be visited must have a set of visited neighbours which is **inclusion-wise maximal**.

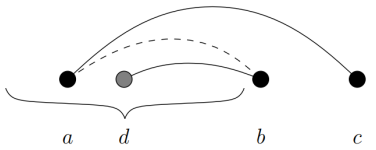
Far-reaching generalization of:

- LexBFS
- LexDFS
- MCS

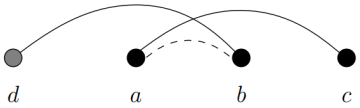
Common properties to all these graph searches can be explained/proved only once by proving them directly for MNS.

Four-point characterizations

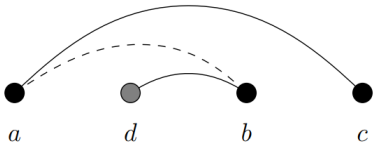
- Generic search:



- BFS:

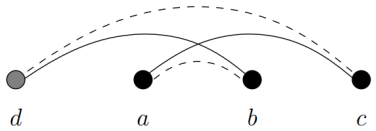


- DFS:

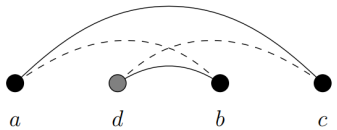


Four-point characterizations

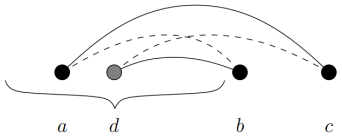
- LexBFS:



- LexDFS:



- MNS:



Consequence: Polynomial-time recognition of search orderings.

What about MCS?

No 4-point characterization known.

(P_3) If $a < b < \{c_1, \ldots, c_k\}$, $c_1, \ldots, c_k$ pairwise distinct vertices, and $ac_i \in E$ and $bc_i \notin E$, $i = 1, \ldots, k$, then there are pairwise distinct vertices, $d_1, \ldots, d_k$ such that $b < d_i$, $d_i b \in E$ and $d_i a \notin E$, $i = 1, \ldots, k$.

Recognition of MCS order in $\mathcal{O}(n + m)$ time: scan the ordering and, for every vertex not yet visited, keep track of its number of visited neighbours.

Multi-sweep

- At some points during the execution of some graph search, there may be several vertices which could be visited next.
- A tie-break rule consists in an additional rule in order to decide *unambiguously* which vertex must be next visited.
- For instance, we may use the ordering obtained from a previous search.
- This is a powerful approach for many problems (*e.g.*, recognition of graph classes).
- Application to the recognition of (Lex)DFS/BFS orders: just execute the algorithm and use the order to be checked for tie-break rules.

$\rightarrow$ equivalent point of view: in the partition refinement data structures, vertices are sorted according to the order to be verified.

Weighted graphs

If all edge-weights are positive, we can use Dijkstra's algorithm as a replacement for BFS:

`H := empty heap`

`Insert every` $v \in V$ `in H with infinite value.`

`H[`v_{n-1}`] := 0 //`*start vertex*

while `H is nonempty:`

  (v_i, d_i) `:= H.extract_min()`

  for all $u \in N(v_i)$ `such that` u `is in the heap:`

  if d_i `+` $w(v_i, u)$ `< H[u]:  H.decrease_key(u,`d_i `+` $w(v_i, u)$`)`

We need to perform on the heap: n insertions, n deletions, and $\mathcal{O}(m)$ decrease key operations.

$\rightarrow \mathcal{O}(n \log n + m)$ time by using Fibonacci heaps.

Optimality

Being given a vertex `v` with n elements, we construct a star with $n+1$ nodes, whose center is numbered $n+1$ and whose leaves are numbered $0, 1, \ldots, n$.

- For every i such that $0 \leq i \leq n$, the edge between i and the center $n+1$ has weight $v[i]$.

Proposition: If we follow the order in which Dijkstra's algorithm visits the nodes, then we can sort vector `v`.

Consequence: $\Omega(n \log n)$-time lower bound.

Advanced Graph Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Hereditary graph classes

Definition

A class of graphs $\mathcal{G}$ is called hereditary if every induced subgraph of a graph in $\mathcal{G}$ also belongs to the class.

Example:

- Forests
- Bipartite graphs

Property: every hereditary graph class can be characterized as the $\mathcal{H}$-free graphs (a.k.a., all graphs excluding every graph in $\mathcal{H}$ as an induced subgraph), for some possibly infinite family $\mathcal{H}$.

Hereditary graph classes

Definition

A class of graphs $\mathcal{G}$ is called hereditary if every induced subgraph of a graph in $\mathcal{G}$ also belongs to the class.

Example:

- Forests – *Cycle-free* –
- Bipartite graphs – *Odd-cycle-free* –

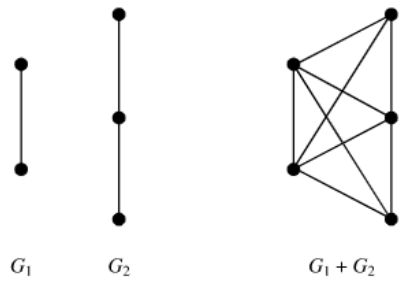
Property: every hereditary graph class can be characterized as the $\mathcal{H}$-free graphs (a.k.a., all graphs excluding every graph in $\mathcal{H}$ as an induced subgraph), for some possibly infinite family $\mathcal{H}$.

P_k-free graphs

- P_1-free graphs: the empty graph...
- P_2-free graphs: edgeless graphs...
- P_3-free graphs: cluster graphs (every connected component must be a clique)
- P_4-free graphs: **cographs** (complement-reducible graphs)

Easy properties of cographs

- If G is connected, then $diam(G) \leq 2$.
- The complement $\overline{G}$ of a cograph G is also a cograph.
- The disjoint union of two cographs G_1, G_2 is also a cograph.
- The **join** of two cographs is also a cograph.



Connectivity

Theorem

If G is a cograph, then either G or $\overline{G}$ is disconnected.

Proof by contradiction. Suppose both $G, \overline{G}$ are connected.

Let $v \in V(G)$ be an arbitrary vertex. We partition the vertex set $V(G)$ in $\{v\}$, $A = N(v)$, $B = V(G) \setminus N[v]$.

(1) Both A, B are nonempty.

(2) If $u, w \in B$ are adjacent, then $N(u) \cap A = N(w) \cap A$.

(3) If $u, w \in B$, then $N(u) \cap A, N(w) \cap A$ are comparable for inclusion.

(4) If $x, y \in A$, then $N(x) \cap B, N(y) \cap B$ are comparable for inclusion. If x, y are nonadjacent, then $N(x) \cap B = N(y) \cap B$.

Connectivity cont'd

Theorem

If G is a cograph, then either G or $\overline{G}$ is disconnected.

(5) There exists a $x \in A$ such that $B \subseteq N(x)$.

There exists a $u \in B$ such that $A \subseteq N(u)$.

$\Longrightarrow$ Every vertex is adjacent to one of u, x.

$\Longrightarrow$ $diam(\overline{G}) \geq 3$. A contradiction.

Decomposition theorem

Every cograph must be either:

- Reduced to one vertex
- The disjoint union of two cographs.
- The join of two cographs.

Remark: these three cases are mutually exclusive.

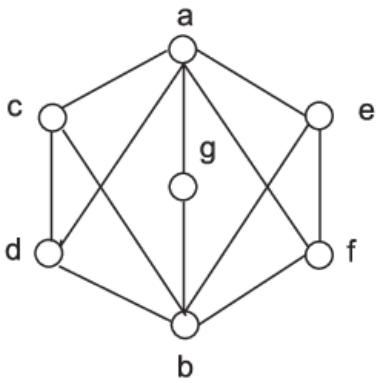
The cotree

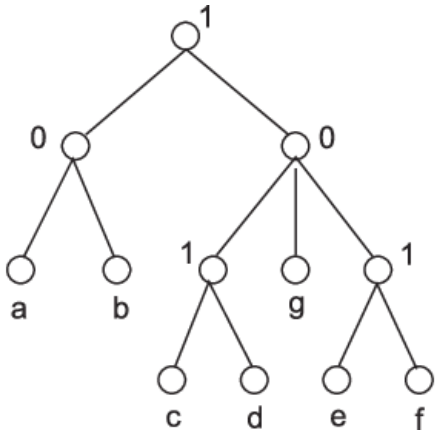
Every cograph G can be uniquely represented by a rooted tree, whose nodes represent subgraphs of G.

- The root represents G itself.
- The leaves represent the vertices of G.
- If a node represents a connected subgraph H, then it is labelled 1 and each child represents a different co-connected component of H.
- If a node represents a disconnected subgrah H, then it is labelled 0 and each child represents a different connected component of H.

Remark: The cotree fully determines the cograph.

Example





Consequence: many NP-hard problems can be solved in linear time on cographs (see the seminars...).

Pruning sequence

Two vertices u, v are **twins** if $N(u) \setminus \{v\} = N(v) \setminus \{u\}$.

- They are false twins if u, v are nonadjacent;
- true twins otherwise.

Theorem

Every cograph contains a pair of twin vertices.

Proof: Take a deeper internal node x in the cotree. All its children are leaves. The corresponding vertices of graph G are true twins if x is labelled 1, and they are false twins if x is labelled 0.

$\rightarrow$ New characterization of cographs: there exists a vertex ordering $v_1, v_2, \ldots, v_n$ such that each v_j, $j < n$, has a twin in the subgraph induced by $v_j, v_{j+1}, \ldots, v_n$.

Recognition of cographs

1) If G is disconnected then:

- Compute the connected components $C_1, C_2, \ldots, C_p$ of G
- Check whether $G[C_i]$ is a cograph for every $1 \leq i \leq p$.

2) Else, if $\overline{G}$ is disconnected, then:

- Compute the co-connected components $C'_1, C'_2, \ldots, C'_q$ of G
- Check whether $G[C'_j]$ is a cograph for every $1 \leq j \leq q$.

3) Else, reject.

Complexity: $\mathcal{O}(nm)$. The algorithm also computes the cotree.

Incremental algorithm

1) If G is a cograph, then so must be $G - v$, for any v. We apply our algorithm recursively on $G - v$. Doing so, we compute a cotree (T', r').

2) If v is universal, then there are two cases.

- r' is labelled 1: we add one leaf labelled with v as a child of r'.
- r' is labelled 0: we add one new root r with label 1 and respective children r' and a new leaf with label v.

Proceed similarly if v is isolated.

3) For every $u \in N(v)$, mark all the nodes of T' between u and the root. This can be done in $\mathcal{O}(n)$ time.

4) Wlog the root has label 0. At least one child must be unmarked. If one child is marked, then we proceed recursively. Otherwise, each connected component must be either disjoint from $N(v)$ or fully contained in $N(v)$. Then, we correct the cotree by adding three more nodes.

Complexity: $\mathcal{O}(m + n^2) = \mathcal{O}(n^2)$.

Recursion depth

We proceed recursively if, for instance:

- the root r_0 is labelled 0, v is not isolated, and there is one marked child r_1;
- r_1 is labelled 1, v is not universal, and all its non-neighbours are descendants of one child r_2 ;
- r_2 is labelled 0, v is not isolated, and there is one marked child r_3;
- ...

Observation 1: all nodes r_i, except maybe the last one, are marked.

Observation 2: if r_i is labelled 1, then some neighbour u is on another branch than r_{i+1}.

$\Longrightarrow$ recursion depth in $\mathcal{O}(d(v))$.

Better analysis

Observation: The bottleneck is the marking process, that runs in $\mathcal{O}(n)$ time for every new vertex v to be inserted.

Let $H_1, H_2, \ldots, H_q$ be the marked children at the root. Note that $q = \mathcal{O}(d(v))$. Furthermore, for every i such that $1 \leq i \leq q$, the number of nodes in the subtree of T' rooted at H_i is in $\mathcal{O}(|V(H_i)|)$.

If G is a cograph and v is neither isolated nor universal, then either $q = 1$, $q = d_{T'}(r') - 1$, or we must have $V(H_i) \subseteq N(v)$ for every i such that $1 \leq i \leq q$.

Therefore, the marking process actually runs in $\mathcal{O}(d(v))$ time!

Theorem

We can recognize cographs, and construct a corresponding cotree, in $\mathcal{O}(n + m)$ time.

Advanced Graph Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Motivations

- Cographs have nice characterizations and algorithmic properties.
- We would like to extend these results to larger graph classes.
- How do we measure closeness to a cograph?
- How to recognize graphs that are "close-to-cographs"?

Reminder: twins

Definition

Two vertices u, v are twins if $N(u) \setminus \{v\} = N(v) \setminus \{u\}$.

Proposition: Being twins is an equivalence relation!

Proof: Assume $N(u) \setminus \{v\} = N(v) \setminus \{u\}$ and $N(v) \setminus \{w\} = N(w) \setminus \{v\}$.

- Assume $u \in N(v)$. Then, $u \in N(v) \setminus \{w\} \subseteq N(w)$. Similarly, $w \in N(u) \setminus \{v\} \subseteq N(v)$. Therefore,

$$\begin{aligned} N(w) \setminus \{u\} &= \{v\} \cup (N(w) \setminus \{u, v\}) = \{v\} \cup (N(v) \setminus \{u, w\}) \\ &= \{v\} \cup (N(u) \setminus \{v, w\}) = N(u) \setminus \{w\} \end{aligned}$$

- Otherwise, u and v are adjacent in the complement $\overline{G}$. Since being twins in G is equivalent to being twins in $\overline{G}$, we are back to the previous case.

Computing false twins

Observation: True twins in G are False twins in $\overline{G}$, and vice-versa.

1) Initialize in $\mathcal{O}(n)$ time a **partition refinement** data structure with one group equal to $V(G)$.

2) For every $v \in V(G)$, refine existing groups according to $N(v)$. it takes $\mathcal{O}(d(v))$ time.

3) Two vertices are false twins if and only if they belong to the same final group.

Complexity: $\mathcal{O}(n + m)$.

$\rightarrow$ For true twins, it suffices to refine according to $N[v]$.

Neighbourhood diversity

Definition (Neighbourhood diversity)

Number of twin equivalence classes.

- Complete graphs have neighbourhood diversity equal to 1.

- Stars, and more generally complete bipartite graphs, have neighbourhood diversity equal to 2.

- However, cographs have unbounded neighbourhood diversity!

$\Longrightarrow$ Need for a stronger property.

Modules

Definition (Module)

A vertex subset M such that, for every $x, y \in M$, we have $N(x) \setminus M = N(y) \setminus M$.

Remark: Twin classes are modules. The converse is not true in general.

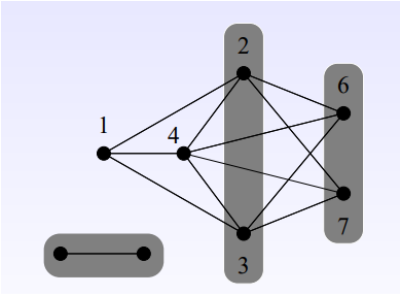
- In every graph $G = (V, E)$, the sets $\emptyset$, V and $\{v\}$ for every vertex $v \in V$ are always modules.

- A graph is **prime** if it only has trivial modules.

$\rightarrow$ A cograph with > 1 vertices is never prime because it contains a pair of twins. It implies that every prime graph contains an induced P_4.

Examples

- Every connected component of a graph G is a module.
- Every co-connected component of a graph G is also a module.
- The same holds for any disjoint union of (co)-connected components.



$\rightarrow$ a prime graph must be connected and co-connected.

$\rightarrow$ the number of modules can be exponential (e.g., in a clique).

$\rightarrow$ the nodes in a cotree represent modules of a cograph. We aim at obtaining a similar tree representation for the modules in an arbitrary graph.

Basic properties

(1) M is a module of G if and only if M is a module of $\overline{G}$.

(2) If M is a module of G, and M' is a module of $G[M]$, then M' is also a module of G.

(3) If M, M' are intersecting modules of G then $M \cap M'$, $M \cup M'$, $M \setminus M'$, $M' \setminus M$ and $M \Delta M'$ (symmetric difference) are also modules of G.

Strong modules

Definition

A module M is strong if it does not overlap any other module, *i.e.*, for any module $M' \neq M$, either $M \cap M' = \emptyset$, $M \subseteq M'$, or $M' \subseteq M$.

A **maximal strong module** is a strong module $M \neq V$ that is inclusion-wise maximal.

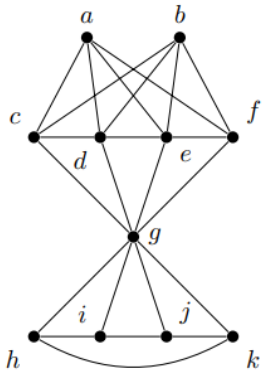
Proposition: the family $\mathcal{M}(G)$ of maximal strong modules of G is a partition of $V(G)$.

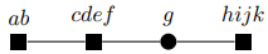
Proof: every vertex v is in a strong module, namely, $\{v\}$. Furthermore, since modules of $\mathcal{M}(G)$ are strong, they cannot overlap.

Quotient graph

Definition

The quotient subgraph of G, denoted by $G_{/\mathcal{M}(G)}$, is the induced subgraph obtained by keeping one vertex in every maximal strong module of G.





Modular decomposition theorem

Theorem (Gallai, 1967)

For every graph $G = (V, E)$ with at least four vertices, exactly one of the following conditions must be true:

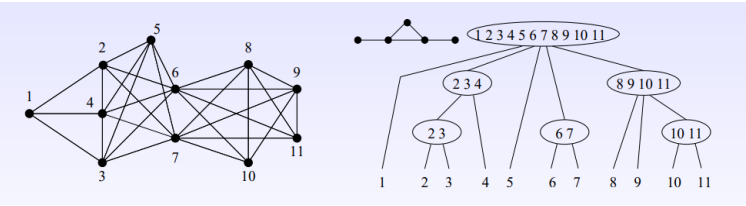
- *G is disconnected;*
- *$\overline{G}$ is disconnected;*
- *$G_{/\mathcal{M}(G)}$ is prime.*

Remark: for cographs, we always fall in one of the two first cases.

Modular decomposition tree

Generalization of the cotree, where we implicitly represent all modules of a graph G. Every node represents a different strong module of G.

- The root represents V
- The leaves represent the vertices of G
- For a strong module M with > 1 vertices, if $G[M]$ is (co-)disconnected then the children of M must represent the (co-)connected components of $G[M]$. Otherwise, the children of M represent $\mathcal{M}(G[M])$.



Computation of the modular decomposition tree

1) For every two vertices x, y, we compute the smallest module $m(x, y)$ that contains both x, y.

m(x,y) := {x,y}

while there exists a vertex v with both a neighbour and a non-neighbour in m(x,y):

add all such vertices v to m(x,y)

2) If $m(x, y) = V$ for every x, y then G is prime.

3) Otherwise, let $A = m(x, y) \neq V$ be arbitrary. We replace A in G by a new vertex a, that results in a new graph G_a. We compute the modular decompostion of G_a and $G[A]$ separately.

State of the art

Our algorithm from the previous slide is polynomial, but far from linear.

Theorem (Tedder et al., 2008)

The modular decomposition tree of any graph can be computed in $\mathcal{O}(n + m)$ time.

This result will be admitted in the subsequent classes and seminars.