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.

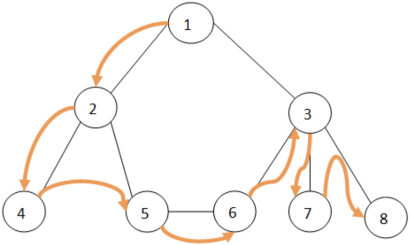
Examples:

- 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.

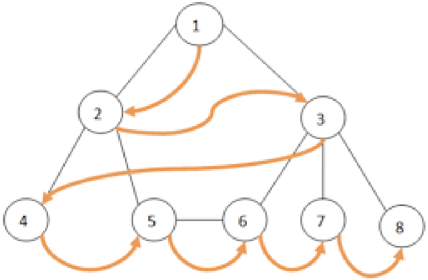
Characterization: A spanning tree of a connected graph G is a palm tree if and only if every *backward edge* xy (i.e., not in the spanning tree) satisfies that x is an ancestor of y, or y is an ancestor of x.

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 $v \in N(u)$:

    if !visited[v]:

      Q.enqueue(v)

      visited[v] := True

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

Shortest path tree

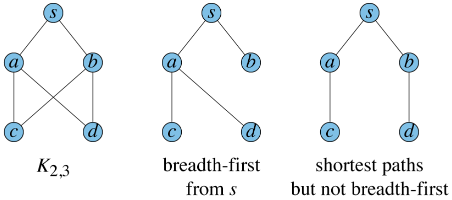
If G is connected, then we can define the distance $d_G(u, v) =$ minimum number of edges on a uv-path.

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:



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

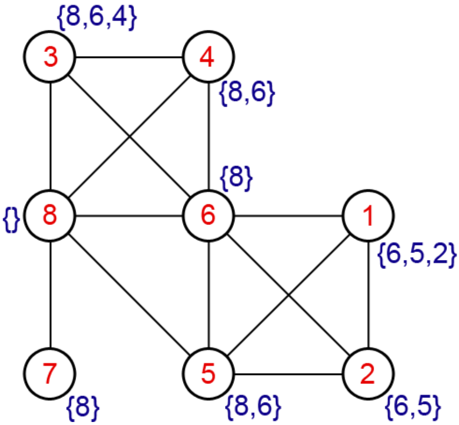
A weakening of BFS: Pick a **closest to the root visited** vertex with at least one neighbour unvisited. Then, go to an arbitrary unvisited neighbour of this vertex.

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 $X \setminus S$.

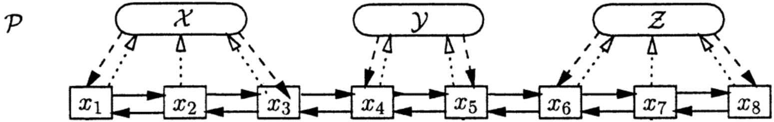
- Operation `init(V)` is in worst-case $\mathcal{O}(|V|)$. Each operation `refine(S)` is in worst-case $\mathcal{O}(|S|)$. Note that these are optimal runtimes!

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.



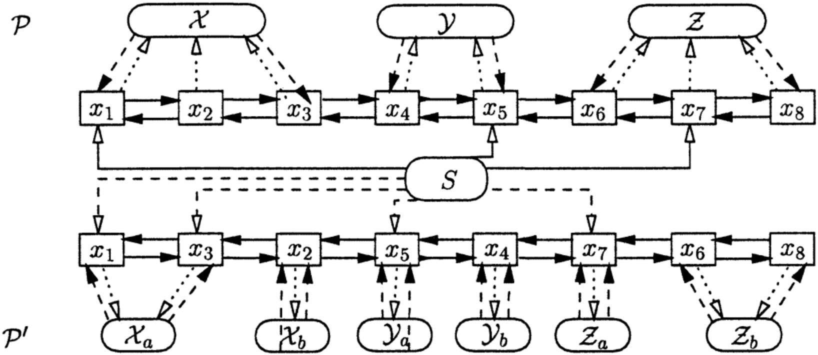
Refinement

- 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 $L[X]$.

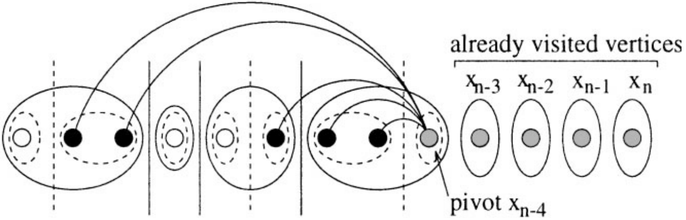
Refinement: illustration



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.

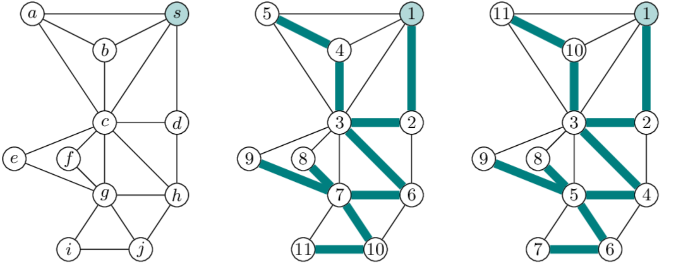
Tightly related to the following end-vertex problem:

- 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 $lab(X_{i,1}) \geq lab(X_{i,2}) \geq \ldots \geq lab(X_{i,q_i})$.

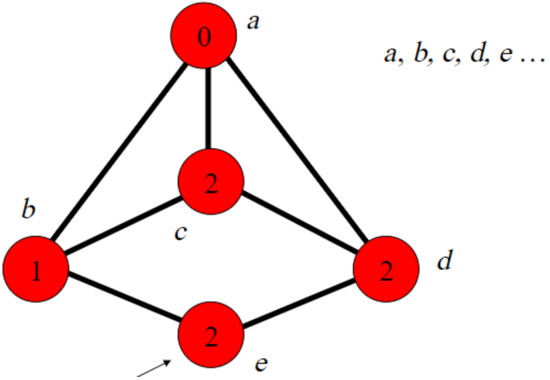
3) We refine: $X_{i,1} \cap N(v_i), X_{i,1} \setminus N(v_i), X_{i,2} \cap N(v_i), X_{i,2} \setminus N(v_i), \ldots, X_{i,q_i} \cap N(v_i), X_{i,q_i} \setminus N(v_i)$.

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**.

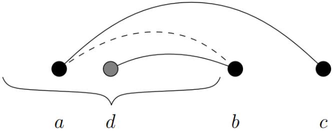
Far-reaching generalization of:

- LexBFS
- LexDFS
- MCS

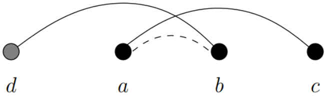
Common properties to all these graph searches can be explained/proved only once by proving them directly for MNS.

Four-point characterizations

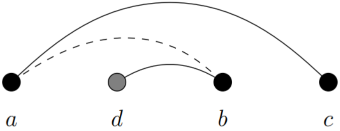
- Generic search:



- BFS:

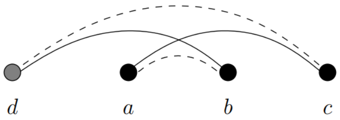


- DFS:

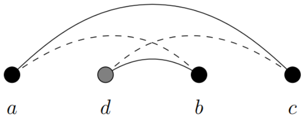


Four-point characterizations

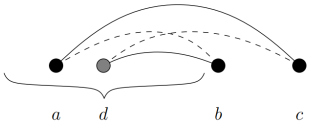
- LexBFS:



- LexDFS:



- MNS:



<u>Consequence</u>: 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 $d_i + w(v_i, u) < \texttt{H}[u]$: `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 $\mathtt{v}$ 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 $\mathtt{v}$.

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

Questions