

Algorithms (VI)

Yijia Chen
Shanghai Jiaotong University

Chapter 3. Decompositions of graphs

Why graphs?

- ▶ A wide range of problems can be expressed with clarity and precision in the concise pictorial language of graphs.
 - ▶ Graph coloring.
 - ▶ Graph connectivity and reachability.
 - ▶ Flow.
- ▶ Formally, a graph is specified by a set of **vertices** (also called **nodes**) V and by **edges** E between select pairs of vertices.
 - ▶ **Undirected graphs**, i.e., E is a symmetric relation.
 - ▶ **Directed graphs**.

How is a graph represented?

We can represent a graph by an *adjacency matrix*: if there are $n = |V|$ vertices v_1, \dots, v_n , this is an $n \times n$ array whose (i, j) th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

For undirected graph, the matrix is *symmetric* since an edge $\{u, v\}$ can be taken in either direction.

Pros: the presence of a particular edge can be checked in constant time, with just one memory access.

Cons: Takes up $O(n^2)$ space, which is wasteful if the graph does not have very many edges.

How is a graph represented? (cont'd)

We can represent a graph by an *adjacency list*: It consists of $|V|$ linked lists, one per vertex.

The linked list for vertex u holds the names of vertices to which u has an *outgoing edge* – that is – vertices v for which $(u, v) \in E$.

Therefore, each edge appears in exactly one of the linked lists if the graph is directed or two of the lists if the graph is undirected.

Pros: The total size of the data structure is $O(|E|)$.

Cons: Checking for a particular edge (u, v) is no longer constant time.

Depth-first search in undirected graphs

The basic question

WHAT PARTS OF THE GRAPH ARE REACHABLE FROM A GIVEN VERTEX?

Exploring graphs

EXPLORE(G, v)

Input: $G = (V, E)$ is a graph; $v \in V$

Output: $\text{visited}(u)$ is set to true for all nodes u **reachable** from v

1. $\text{visited}(v) = \text{true}$
2. $\text{PREVIST}(v)$
3. **for** each edge $(v, u) \in E$ **do**
4. **if** not $\text{visited}(u)$ **then** $\text{EXPLORE}(u)$
5. $\text{POSTVISIT}(v)$

PREVIST and POSTVISIT procedures are optional, meant for performing operations on a vertex when it is **first discovered** and also when it is being **left for the last time**.

Exploring graphs (cont'd)

Theorem

`EXPLORE`(G, v) is *correct*, i.e., it visits exactly all nodes that are reachable from v .

Proof.

Every node which it visits must be reachable from v :

`EXPLORE` only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v .

Every node which is reachable from v must be visited eventually:

If there is some u that `EXPLORE` misses, choose any path from v to u , and look at the last vertex v on that path that the procedure actually visited. Let w be the node immediately after it on the same path.

So z was visited but w was not. This is a contradiction: while `EXPLORE` was at node z , it would have noticed w and moved on to it. □

Edge types

Those edges in G that are traversed by EXPLORE are *tree edges*.

The rest are *back edges*.

Depth-first search

DFS(G)

1. **for all** $v \in V$ **do**
2. $\text{visited}(v) = \text{false}$
3. **for all** $v \in V$ **do**
4. **if not** $\text{visited}(v)$ **then** EXPLORE(v)

Running time of DFS

Because of the visited array, each vertex is EXPLORE'd just *once*.

During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work – marking the spot as visited, and the PRE/POSTVISIT.
2. A loop in which adjacent edges are scanned, to see if they lead somewhere new.

This loop takes a different amount of time for each vertex. The total work done in step 1 is then $O(|V|)$. In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly *twice*, once during EXPLORE(x) and once during EXPLORE(y).

The overall time for step 2 is therefore $O(|E|)$ and so the depth-first search has a running time of $O(|V| + |E|)$.

Connectivity in undirected graphs

Definition

An undirected graph is **connected**, if there is a path between any pair of vertices.

Definition

A **connected component** is a subgraph that is internally connected but has no edges to the remaining vertices.

When `EXPLORE` is started at a particular vertex, it identifies precisely the connected component containing that vertex.

Each time the DFS outer loop calls `EXPLORE`, a new connected component is picked out.

Connectivity in undirected graphs (cont'd)

Thus depth-first search is trivially adapted to check if a graph is connected.

More generally, to assign each node v an integer $\text{ccnum}[v]$ identifying the connected component to which it belongs.

All it takes is

PREVISIT(v)

$\text{ccnum}[v] = cc$

where cc needs to be initialized to zero and to be incremented each time the DFS procedure calls `EXPLORE`.

Previsit and postvisit orderings

For each node, we will note down the times of two important events:

- ▶ the moment of first discovery (corresponding to `PREVISIT`);
- ▶ and the moment of final departure (`POSTVISIT`).

PREVISIT(v)

`pre`[v] = clock

clock = clock + 1

POSTVISIT(v)

`post`[v] = clock

clock = clock + 1

Lemma

For any nodes u and v , the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are either disjoint or one is contained within the other.

Depth-first search in directed graphs

Types of edges

DFS yields a **search tree/forests**.

- ▶ root.
- ▶ descendant and ancestor.
- ▶ parent and child.

- ▶ **Tree edges** are actually part of the DFS forest.
- ▶ **Forward edges** lead from a node to a nonchild descendant in the DFS tree.
- ▶ **Backedges** lead to an ancestor in the DFS tree.
- ▶ **Cross edges** lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

Types of edges (cont'd)

pre/post ordering for (u, v)	Edge type
$[u \ [v \]v \]u$	Tree/forward
$[v \ [u \]u \]v$	Back
$[v \]v \ [u \]u$	Cross

Directed acyclic graphs (DAG)

Definition

A cycle in a directed graph is a circular path

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots v_k \rightarrow v_0.$$

Lemma

A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Proof.

One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

Conversely, if the graph has a cycle $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots v_k \rightarrow v_0$, look at the first node v_i on this cycle to be discovered (the node with the lowest pre number). All the other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree. In particular, the edge $v_{i-1} \rightarrow v_i$ (or $v_k \rightarrow v_0$ if $i = 0$) is a back edge. □

Directed acyclic graphs (cont'd)

Linearization/Topologically Sort: Order the vertices such that every edge goes from a small vertex to a large one.

Lemma

In a dag, every edge leads to a vertex with a lower post number.

Hence there is a linear-time algorithm for ordering the nodes of a dag.

Since a dag is linearized by decreasing post numbers, the vertex with the smallest post number comes last in this linearization, and it must be a **sink** – no outgoing edges. Symmetrically, the one with the highest post is a **source**, a node with no incoming edges.

Lemma

Every dag has at least one source and at least one sink.

The guaranteed existence of a source suggests an alternative approach to linearization:

1. Find a source, output it, and delete it from the graph.
2. Repeat until the graph is empty.

Strongly connected components

Defining connectivity for directed graphs

Definition

Two nodes u and v of a directed graph are **connected** if there is a path from u to v and a path from v to u .

This relation partitions V into disjoint sets that we call **strongly connected components**.

Lemma

Every directed graph is a dag of its strongly connected components.

An efficient algorithm

Lemma

If the EXPLORE subroutine is started at node u , then it will terminate precisely when all nodes reachable from u have been visited.

Therefore, if we call explore on a node that lies somewhere in a **sink strongly connected component** (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component.

We have two problems:

- (A) How do we find a node that we know for sure lies in a sink strongly connected component?
- (B) How do we continue once this first component has been discovered?

An efficient algorithm (cont'd)

Lemma

*The node that receives the highest post number in a depth-first search must lie in a **source strongly connected component**.*

Lemma

If C and C' are strongly connected components, and there is an edge from a node in C to a node in C' , then the highest post number in C is bigger than the highest post number in C' .

Hence the strongly connected components can be linearized by arranging them in decreasing order of their highest post numbers.

Solving problem A

Consider the **reverse graph** G^R , the same as G but with all edges **reversed**.

G^R has exactly the same strongly connected components as G .

So, if we do a depth-first search of G^R , the node with the highest post number will come from a source strongly connected component in G^R , which is to say a sink strongly connected component in G .

Solving problem B

Once we have found the first strongly connected component and deleted it from the graph, the node with the highest post number among those remaining will belong to a sink strongly connected component of whatever remains of G .

Therefore we can keep using the post numbering from our initial depth-first search on G^R to successively output the second strongly connected component, the third strongly connected component, and so on.

The linear-time algorithm

1. Run depth-first search on G^R .
2. Run the undirected connected components algorithm on G , and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1.