"Read Euler, read Euler, he is our master in everything"
– Pierre-Simon Laplace (1749–1827)

Lecture IV
PURE GRAPH ALGORITHMS

Graph Theory is said to have originated with Euler (1707–1783). The citizens of the city[1] of Königsberg asked him to resolve their favorite pastime question: *is it possible to traverse all the 7 bridges joining two islands in the river Pregel and the mainland, without retracing any path?* See Figure 1(a) for a schematic layout of these bridges. Euler recognized[2] in this problem the essense of Leibnitz's earlier interest in founding a new kind of mathematics called "analysis situs". This can be interpreted as topological or combinatorial analysis in modern language. A graph corresponding to the 7 bridges and their interconnections is shown in Figure 1(b). Computational graph theory has a relatively recent history. Among the earliest papers on graph algorithms are Prim's minimum spanning tree algorithm (1957) and Dijkstra's shortest path algorithm (1959). Tarjan is one of the first to systematically design and analyze many of the basic graph algorithms, including applications of DFS which we will study.

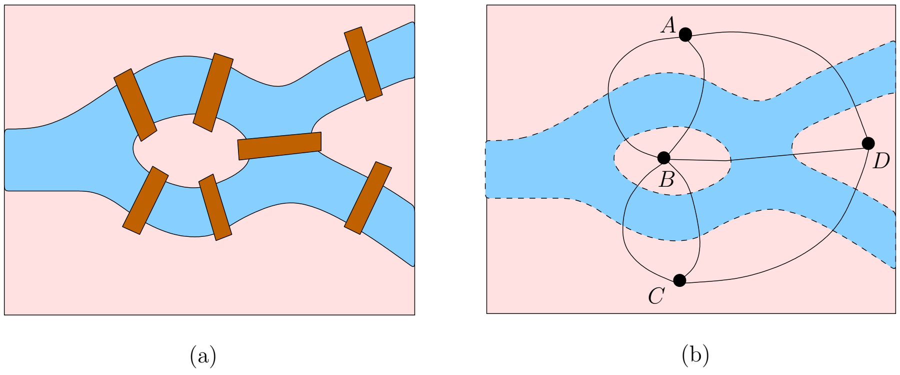


Figure 1: The 7 Bridges of Konigsberg

Graphs are useful for modeling abstract mathematical relations in computer science as well as in many other disciplines. Here are some examples of graphs:

Adjacency between Countries Figure 2(a) shows a political map of 7 countries. Figure 2(b) represents a graph with vertex set $V = \{1, 2, \ldots, 7\}$ representing these countries. An edge $i{-}j$ represents the relationship between countries i and j that share a continuous (i.e., connected) common border. Note that countries 2 and 3 share two continuous common borders, and so we have two copies of the edge $2{-}3$.

[1]This Prussian city is also called Kaninsgrad in Russian
[2]His paper was entitled "Solutio problematis ad geometriam situs pertinentis" (The solution of a problem relating to the geometry of position).

Flight Connections A graph can represent the flight connections of a particular airline, with the set V representing the airports and the set E representing the flight segments that connect pairs of airports. Each edge will typically have auxiliary data associated with it. For example, the data may be numbers representing flying time of that flight segment.

Hypertext Links In hypertext documents on the world wide web, a document will generally have links ("hyper-references") to other documents. We can represent these linkages structure by a graph whose vertices V represent individual documents, and each edge $(u, v) \in V \times V$ indicates that there is a link from document u to document v.

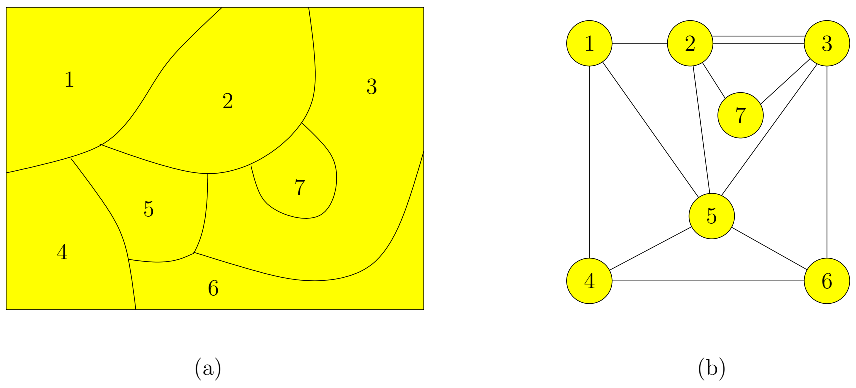


Figure 2: (a) Political map of 7 countries (b) Their adjacency relationship

A graph is fundamentally a set of mathematical relations (called incidence relations) connecting two sets, a vertex set V and an edge set E. In Figure 1(b), the vertex set is $V = \{A, B, C, D\}$ and the edges are the 7 arcs connecting pairs of vertices. A simple notion of an edge $e \in E$ is where e is a pair of vertices $u, v \in V$. The pair can be ordered $e = (u, v)$ or unordered $e = \{u, v\}$, leading to two different kinds of graphs. We shall denote[3] such a pair by "u–v", and rely on context to determine whether an ordered or unordered edge is meant. For unordered edges, we have u–$v = v$–u; but for ordered edges, u–$v \neq v$–u unless $u = v$. Note that this simple model of edges (as ordered or unordered pairs) is unable to model the Konigsberg graph Figure 1(b) since it has two copies of the edge between A and B. Such multiple copies of edges can be captured by the concept of multi-edges.

In many applications, our graphs have associated data such as numerical values ("weights") attached to the edges and vertices. These are called **weighted graphs**. The flight connection graph above is an example of this. Graphs without such numerical values are called **pure graphs**. In this chapter, we restrict attention to pure graph problems; weighted graphs will be treated in later chapters. Many algorithmic issues of pure graphs relate to the concepts of connectivity and paths. Many of these algorithms can be embedded in one of two graph searching strategies called depth-first search (DFS) and breadth-first search (BFS).

What can be impure of graphs?

Some other important problems of pure graphs are: testing if a bigraph is planar, finding a maximum matching in a bigraph, and testing isomorphism of bigraphs. Tarjan [4] was one of the first to systematically study the DFS algorithm and its applications. A lucid account of basic graph theory is Bondy and Murty [1]; for a more algorithmic treatment, see Sedgewick [3].

§1. Varieties of Graphs

[3]We have taken this highly suggestive notation from [3].

In this book, "graphs" refer to either directed graphs ("digraphs") or undirected graphs ("bigraphs"). Additional basic graph terminology is collected in Lecture I (Appendix A) for reference.

¶1. Set-Theoretic Notations for Simple Graphs. Although there are many varieties of graph concepts studied in the literature, two main ones are emphasized in this book. These correspond to graphs whose edges $u{-}v$ are **directed** or **undirected**. Graphs with directed edges are called **directed graphs** or simply, **digraphs**. Undirected edges are also said to be **bidirectional**, and the corresponding graphs are known as **undirected graphs** or **bigraphs**.

A graph G is basically given by two sets, V and E. These are called the **vertex set** and **edge set**, respectively. We focus on the "simple" versions of three main varieties of graphs. The terminology "simple" will become clear below.

For any set V and integer $k \geq 0$, let

$$V^k, \qquad 2^V, \qquad \binom{V}{k} \tag{1}$$

denote, respectively, the k-fold **Cartesian product** of V, **power set** of V and the **set of k-subsets** of V. The first two notations (V^k and 2^V) are standard notations; the last one is less so. These notations have a certain "umbral quality" because they satisfy the following equations on set cardinality:

$$\left|V^k\right| = |V|^k, \qquad \left|2^V\right| = 2^{|V|}, \qquad \left|\binom{V}{k}\right| = \binom{|V|}{k}.$$

We can define our 3 varieties of simple graphs as follows:

- A **hypergraph** is a pair $G = (V, E)$ where $E \subseteq 2^V$.
- A **directed graph** (or simply, **digraph**) is a pair $G = (V, E)$ where $E \subseteq V^2$.
- A **undirected graph** (or[4] simply, **bigraph**) is a pair $G = (V, E)$ where $E \subseteq \binom{V}{2}$.

In all three cases, the elements of V are called **vertices**. Elements of E are called **directed edges** for digraphs, **undirected edges** for bigrpahs, and **hyperedges** for hypergraphs. Formally, a directed edge is an ordered pair (u, v), and an undirected edge is a set $\{u, v\}$. But we shall also use the notation $u{-}v$ to represent an **edge** which can be directed or undirected, depending on the context. This convention is useful because many of our definitions cover both digraphs and bigraphs. Similarly, the term **graph** will cover both digraphs and bigraphs. Hypergraphs are sometimes called **set systems** (see matroid theory in Chapter 5).

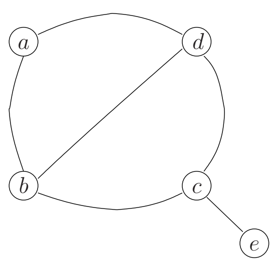
So $u{-}v$ can mean (u, v) or $\{u, v\}$!

An edge $u{-}v$ is said to be **incident** on u and v; conversely, we say u and v **bounds** the edge $\{u, v\}$. This terminology comes from the geometric interpretation of edges as a curve segment whose endpoints are vertices. In case $u{-}v$ is directed, we call u the **start vertex** and v the **stop vertex**.

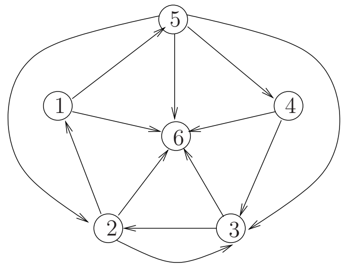
If $G = (V, E)$ and $G' = (V', E')$ are graphs such that $V \subseteq V'$ and $E \subseteq E'$ then we call G a **subgraph** of G'. When $E = E' \cap \binom{V}{2}$, we call G' the subgraph of G that is **induced by** V.

[4]While the digraph terminology is fairly common, the bigraph terminology is peculiar to this book. We hope that this convenient and suggestive terminology find wider adoption.

¶2. Graphical Representation of Graphs. Bigraphs and digraphs are "linear graphs" in which each edge is incident on one or two vertices. Such graphs have natural graphical (i.e., pictorial) representation: elements of V are represented by points (small circles, etc) in the plane and elements of E are represented by finite curve segments connecting these points.



(a) bigraph



(b) digraph

Figure 3: Two graphs

In Figure 3(a), we display a bigraph (V, E) where $V = \{a, b, c, d, e\}$ and $E = \{a–b, b–c, c–d, d–a, c–e, b–d\}$. In Figure 3(b), we display a digraph (V, E) where $V = \{1, 2, \ldots, 6\}$ and $E = \{1–5, 5–4, 4–3, 3–2, 2–1, 1–6, 2–6, 3–6, 4–6, 5–6, 5–2, 5–3, 2–3\}$. We display a digraph edge $u–v$ by drawing an arrow from the start vertex u to the stop vertex v. E.g., in Figure 3(b), vertex 6 is the stop vertex of each of the edges that it is incident on. So all these edges are "directed" towards vertex 6. In contrast, the curve segments in bigraphs are undirected (bi-directional).

¶3. Non-Simple Graphs. Our definition of bigraphs, digraphs and hypergraphs is not the only reasonable one, obviously. To distinguish them from other possible approaches, we call the graphs of our definition "simple graphs". Let us see how some non-simple graphs might look like. An edge of the form $u–u$ is called a **loop**. For bigraphs, a loop would correspond to a set $\{u, u\} = \{u\}$. But such edges are excluded by definition. If we want to allow loops, we must define E as a subset of $\binom{V}{2} \cup \binom{V}{1}$. Note that our digraphs may have loops, which is at variance with some other definitions of "simple digraphs". In Figures 1(b) and in 2(b), we see the phenemenon of **multi-edges** (also known as **parallel edges**). These are edges that can occur more than once in the graph.

More generally, we view E as a multiset. A **multiset** S is an ordinary set $\underline{S}$ together with a function $\mu : \underline{S} \to \mathbb{N}$. We call $\underline{S}$ the **underlying set** of S and $\mu(x)$ is the **multiplicity** of $x \in \underline{S}$. E.g., if $\underline{S} = \{a, b, c\}$ and $\mu(a) = 1, \mu(b) = 2, \mu(c) = 1$, then we could display S as $\{a, b, b, c\}$, and this is not the same as the multiset $\{a, b, b, b, c\}$, for instance.

¶4. Special Classes of Graphs. First consider bigraphs. **Planar graphs** are those bigraphs with an embedding in the Euclidean plane such that no two edges cross. Planar graphs have many special properties: for instance, a planar graph with n vertices has at most $3n - 6$ edges. The two smallest examples of non-planar graphs are the graphs K_5 and $K_{3,3}$ in Figure 4.

Bipartite graphs are those whose vertex set V can be partitioned in two sets $A \uplus B = U$ such that each edge is incident on some vertex in A and on some vertex in B.

In Appendix (Lecture I), we defined the complete graph K_n and the complete bipartite graph $K_{m,n}$ for all $m, n \in \mathbb{N}$. The two special cases of K_5 and $K_{3,3}$ are notable: these are the Kuratowski

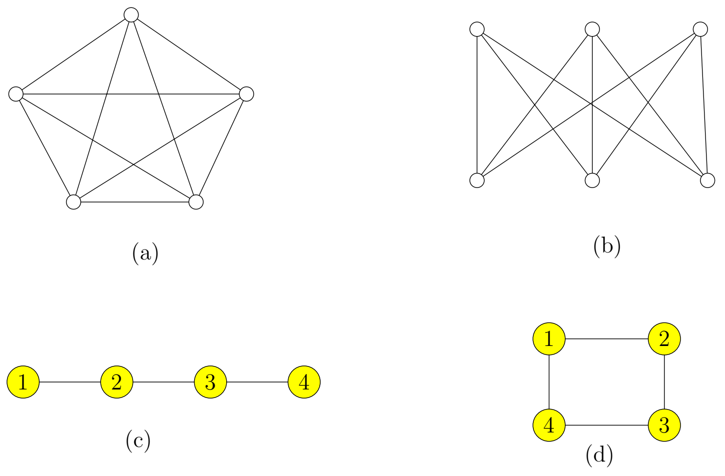


Figure 4: (a) K_5, (b) $K_{3,3}$, (c) L_4, (d) C_4

graphs, and they are non-planar. See Figure 4(a,b). We can also define the **line graphs** L_n whose nodes are $\{1, \ldots, n\}$, with edges i–$i+1$ for $i = 1, \ldots, n-1$. Closely related is the **cyclic graphs** C_n which is obtained from L_n by adding the extra edge n–1. These are illustrated in Figure 4(c,d).

These graphs, $K_n, K_{m,n}, L_n, C_n$ are usually viewed as bigraphs, but there are obvious digraphs versions.

Graph Isomorphism. The concept of graph isomorphism (see Appendix, Lecture I) is important to understand. It is implicit in many of our discussions that we are only interested in graphs *up to isomorphism*. For instance, we defined K_n ($n \in \mathbb{N}$) as "the complete graphs on n vertices" (Appendix, Lecture I). But we never specified the vertex set of K_n. This is because K_n is really an isomorphism class. For instance, $G = (V, E)$ where $V = \{a, b, c, d\}$ and $E = \binom{V}{2}$ and $G' = (V', E')$ where $V' = \{1, 2, 3, 4\}$ and $E' = \binom{V'}{2}$ are isomorphic to each other. Both belong to the isomophism class K_4. There is usually a way to avoid isomorphism classes, but picking a canonical representative. In the case of K_n, we can just view it as a bigraph whose vertex set is a particular set, $V_n = \{1, 2, \ldots, n\}$. Then the edge set (in case of K_n) is completely determined.

¶5. Auxiliary Data Convention. We may want to associate some additional data with a graph. Suppose we associate a real number $W(e)$ for each $e \in E$. Then graph $G = (V, E; W)$ is called **weighted graph** with weight function $W : E \to \mathbb{R}$. Again, suppose we want to designate two vertices $s, t \in V$ as the **source** and **destination**, respectively. We may write this graph as $G = (V, E; s, t)$. In general, auxiliary data such as W, s, t will be separated from the pure graph data by a semi-colon, $G = (V, E; \cdots)$. Alternatively, G is a graph, and we want to add some additional data d, d', we may also write $(G; d, d')$, etc.

Exercises

Exercise 1.1: (Euler) Prove that there is no way to traverse all seven bridges in Figure 1(a) without going any bridge twice. ♢

Exercise 1.2: Suppose we have a political map as in Figure 2(a), and its corresponding adjacency relation is a multigraph $G = (V, E)$ where E is not a multiset whose underlying set is a subset of $\binom{V}{2}$.
(a) Suppose vertex u has the property that there is a unique vertex v such that $u{-}v$ is an edge. What can you say about the country corresponding to u?
(b) Suppose $u{-}v$ has multiplicity ≥ 2. Consider the set $W = \{w \in V : w{-}v \in E, w{-}u \in E\}$. What can you say about the set W? ♢

Exercise 1.3: Prove or disprove: there exists a bigraph $G = (V, E)$ where $|V|$ is odd and the degree of each vertex is odd. ♢

Exercise 1.4:
(i) How many bigraphs, digraphs, hypergraphs are there on n vertices?
(ii) How many non-isomorphic bigraphs, digraphs, hypergraphs are there on n vertices? Estimate these with upper and lower bounds. ♢

Exercise 1.5: A trigraph is $G = (V, E)$ where $E \subseteq \binom{V}{3}$. An element $f \in E$ is called a **face** (not "edge"). A pair $\{u, v\} \in \binom{V}{2}$ is called an **edge** provided $\{u, v\} \subseteq f$ for some face f; in this case, we say f is **incident** on e, and e **bound** f). The trigraph is an (abstract) **surface** if each edge bounds exactly two faces. How many nonisomorphic surfaces are there on $n = |V|$ vertices? First consider the case $n = 4, 5, 6$. ♢

End Exercises

§2. Path Concepts

We now go into some of these concepts in slightly more detail. Most basic concepts of pure graphs revolve around the notion of a path.

Let $G = (V, E)$ be a graph (*i.e.*, digraph or bigraph). If $u{-}v$ is an edge, we say that v is **adjacent to** u, and also u is **adjacent from** v. The typical usage of this definition of adjacency is in a program loop:

```
for each $v$ adjacent to $u$,
    do " ... $v$ ... "
```

Let $p = (v_0, v_1, , \ldots, v_k)$, $(k \geq 0)$ be a sequence of vertices. We call p a **path** if v_i is adjacent to v_{i-1} for all $i = 1, 2, \ldots, k$. In this case, we can denote p by $(v_0{-}v_1{-}\cdots{-}v_k)$.

The **length** of p is k (not $k + 1$). The path is **trivial** if it has length 0, $p = (v_0)$. Call v_0 is the **source** and v_k the **target** of p. Both v_0 and v_k are **endpoints** of p. We also say p is a path **from** v_0 **to** v_k The path p is **closed** if $v_0 = v_k$ and it is **simple** if all its vertices, with the possible exception of $v_0 = v_k$, are distinct. Note that a trivial path is closed and simple. The **reverse** of $p = (v_0{-}v_1{-}\cdots{-}v_k)$ is the path

$$p^R := (v_k{-}v_{k-1}{-}\cdots{-}v_0).$$

In a bigraph, p is a path iff p^R is a path.

¶6. The Link Distance Metric. Define $\delta_G(u,v)$, or simply $\delta(u,v)$, to be the minimum length of a path from u to v. If there is no path from u to v, then $\delta(u,v)=\infty$. We also call $\delta(u,v)$ the **link distance** from u to v; this terminology will be useful when $\delta(u,v)$ is later generalized to weighted graphs, and when we still need to refer to the ungeneralized concept. The following is easy to see:

- (Non-negativity) $\delta(u,v)\ge 0$, with equality iff $u=v$.
- (Triangular Inequality) $\delta(u,v)\le\delta(u,w)+\delta(w,v)$.
- (Symmetry) When G is a bigraph, then $\delta(u,v)=\delta(v,u)$.

These three properties amount to saying that $\delta(u,v)$ is a metric on V in the case of a bigraph. If $\delta(u,v)<\infty$, we say v is **reachable from** u**.**

¶7. Subpaths. Let p and q be two paths:

$$p=(v_0{-}v_1{-}\cdots{-}v_k),\quad q=(u_0{-}u_1{-}\cdots{-}u_\ell),$$

If p terminates at the vertex where path q begins, i.e., $v_k=u_0$, then the operation of **concatenation** is well-defined. The concatenation of p and q gives a new path, written

$$p;q:=(v_0{-}v_1{-}\cdots{-}v_{k-1}{-}v_k{-}u_1{-}u_2{-}\cdots{-}u_\ell).$$

Note that the common vertex v_k and u_0 are "merged" in $p;q$. Clearly concatenation of paths is associative: $(p;q);r=p;(q;r)$, which we may simply write as $p;q;r$. We say that a path p **contains** q **as a subpath** if $p=p';q;p''$ for some p',p''. If in addition, q is a closed path, we can **excise** q from p to obtain the path $p';p''$. Whenever we write a concatenation expression such as "$p;q$", it is assume that the operation is well-defined.

¶8. Cycles. Two paths p,q are **cyclic equivalent** if there exists paths r,r' such that

$$p=r;r',\qquad q=r';r.$$

We write $p\equiv q$ in this case.

For instance, the following four closed paths are cyclic equivalent:

$$(1{-}2{-}3{-}4{-}1)\equiv(2{-}3{-}4{-}1{-}2)\equiv(3{-}4{-}1{-}2{-}3)\equiv(4{-}1{-}2{-}3{-}4).$$

The first and the third closed paths are cyclic equivalent because of the following decomposition:

$$(1{-}2{-}3{-}4{-}1)=(1{-}2{-}3);(3{-}4{-}1),\qquad(3{-}4{-}1{-}2{-}3)=(3{-}4{-}1);(1{-}2{-}3).$$

If $p=r;r'$ and $r';r$ is defined, then p must be a closed path because the source of r and the target of r' must be the same, and so the source and target of p are identical. Similarly, q must be a closed path.

It is easily checked that cyclic equivalence is a mathematical equivalence relation. We define a **cycle** as an equivalence class of closed paths. If the equivalence class of p is the cycle Z, we call p a **representative** of Z; if $p=(v_0,v_1,\ldots,v_k)$ then we write Z as

$$Z=[p]=[v_1{-}v_2{-}\cdots{-}v_k]=[v_2{-}v_3{-}\cdots{-}v_k{-}v_1].$$

Note that if p has $k+1$ vertices, then $[p]$ is written with only k vertices since the last vertex may be omitted. In case of digraphs, we can have self-loops of the form $u{-}u$ and $p=(u,u)$ is a closed path. The corresponding cycle is $[u]$. However, the trivial path $p=(v_0)$ gives rise to the cycle which is an empty sequence $Z=[\,]$. We call this the **trivial cycle**. Thus, there is only one trivial cycle, independent of any choice of vertex v_0.

Path concepts that are invariant under cyclic equivalence are "transferred" to cycles automatically. Here are some examples: let $Z=[p]$ be a cycle.

- The **length** of Z is the length of p.
- Say Z is **simple** if p is simple.
- We may speak of subcycles of Z: if we excise zero or more closed subpaths from a closed path p, we obtain a closed subpath q; call $[q]$ a **subcycle** of $[p]$. In particular, the trivial cycle is a subcycle of Z. For instance, $[1{-}2{-}3]$ is a subcycle of
$$[1{-}2{-}a{-}b{-}c{-}2{-}3{-}d{-}e{-}3].$$
- The **reverse** of Z is the cycle which has the reverse of p as representative.
- A cycle $Z=[p]$ is trivial if p is a trivial path. So a trivial cycle is written $[(v_0)]=[\,]$.

We now define the notion of a "cyclic graph". For a digraph G, we say it is **cyclic** if it contains any nontrivial cycle. But for bigraphs, this simple definition will not do. To see why, we note that every edge $u{-}v$ in a bigraph gives rise to the nontrivial cycle $[u,v]$. Hence, to define cyclic bigraphs, we proceed as follows: first, define a closed path $p=(v_0{-}v_1{-}\cdots{-}v_{k-1},v_0)$ to be **reducible** if $k\ge 2$ and for some $i=1,\ldots,k$,
$$v_{i-1}=v_{i+1}$$
where subscript arithmetic are modulo k (so $v_k=v_0$ and $v_{k+1}=v_1$). Otherwise p is said to be **irreducible**. A cycle $Z=[p]$ is reducible iff any of its representative p is reducible. Finally, a bigraph is said to be **cyclic** if it contains some irreducible non-trivial cycle.

Let us explore some consequences of these definitions on bigraphs: by definition, the trivial path (v_0) is irreducible. Hence the trivial cycle $[\,]$ is irreducible. There are no cycles of length 1, and any cycle $[u,v]$ of length 2 is always reducible. Hence, irreducible non-trivial cycles have length at least 3. If a closed path $(v_0,\ldots,v_{k-1},v_0)$ is reducible and $k\ge 3$, then it is a non-simple path.

¶9. Connectivity. Let $G=(V,E)$ be a graph (either di- or bigraph). Two vertices u,v in G are **connected** if there is a path from u to v and a path from v to u. Equivalently, $\delta(u,v)$ and $\delta(v,u)$ are both finite. Clearly, connectedness is an equivalence relation on V. A subset C of V is a **connected component** of G if it is an equivalence class of this relation. For short, we may simply call C a **component** of G. Alternatively, C is a non-empty maximal subset of vertices in which any two are connected. Thus V is partitioned into disjoint components. If G has only one connected component, it is said to be **connected**. When $|C|=1$, we call it a **trivial component**. The subgraph of G induced by C is called a **component graph** of G. NOTE: It is customary we may add the qualifier "strong" when discussing components of digraphs. Thus strong components is always a reference to digraphs.

For example, the graph G_6 in Figure 5(a) has $C=\{2,3,5\}$ as a component. The component graph corresponding to C is shown in Figure 5(b). The other components of G are $\{1\},\{4\},\{6\}$, all trivial.

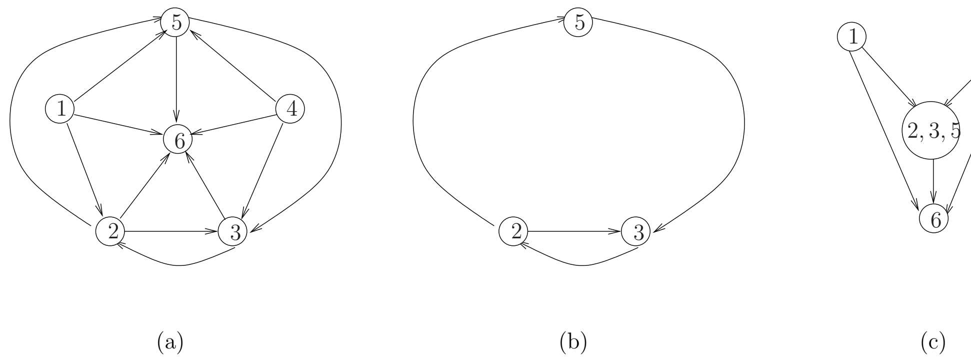


Figure 5: (a) Digraph G_6, (b) Component graph of $C = \{2,3,5\}$, (c) Reduced graph G^c_6

Given G, we define the **reduced graph** $G^c = (V^c, E^c)$ whose vertices comprise the components of G, and whose edges are $(C, C') \in E^c$ such that there exists an edge from some vertex in C to some vertex in C'. This is illustrated in Figure 5(c).

CLAIM: G^c is acycic. In proof, suppose there is a non-trivial cycle Z^c in G^c. This translates into a cycle Z in G that involves at least two components C, C'. The existence of Z contradicts the assumption that C, C' are distinct components.

Although the concept of connected components is meaningful for bigraphs and digraphs, the concept of reduced graph is trivial for bigraphs: this is because there are no edges in G^c when G is a bigraph. Hence the concept of reduced graphs will be reserved for digraphs only. For bigraphs, we will introduce another concept called **biconnected components** below. When G is a bigraph, the notation G^c will be re-interpreted using biconnectivity.

¶10. DAGs and Trees. We have defined cyclic bigraphs and digraphs. A graph is **acyclic** if it is not cyclic. The common acronym for a **directed acyclic graph** is **DAG**. A **tree** is a DAG in which there is a vertex u_0 called the **root** such that there exists a unique path from u_0 to any other vertex. Clearly, the root is unique. Trees, as noted in Lecture III, are ubiquitous in computer science.

Our motto is "know thy tree"

A **free tree** is a connected acyclic bigraph. Such a tree it has exactly $|V|-1$ edges and for every pair of vertices, there is a unique path connecting them. These two properties could also be used as the definition of a free tree. A **rooted tree** is a free tree together with a distinguished vertex called the **root**. We can convert a rooted tree into a directed graph in two ways: by directing each of its edges away from the root (so the edges are child pointers), or by directing each edge towards the root (so the edges are parent pointers).

Exercises

Exercise 2.1: Let u be a vertex in a graph G.
(a) Can u be adjacent to itself if G is a bigraph?
(b) Can u be adjacent to itself if G is a digraph?
(c) Let $p = (v_0, v_1, v_2, v_0)$ be a closed path in a bigraph. Can p be non-simple? ♢

Exercise 2.2: Define $N(m)$ to be the largest value of n such that there is a *connected* bigraph $G = (V, E)$ with $m = |E|$ edges and $n = |V|$ vertices. For instance, $N(1) = 2$ since with one edge, you can have at most 2 nodes in the connected graph G. We also see that $N(0) = 1$. What is $N(2)$? Prove a general formula for $N(m)$.

$\diamondsuit$

Exercise 2.3: Give an algorithm which, given two closed paths $p = (v_0 - v_1 - \cdots - v_k)$ and $q = (u_0 - u_1 - \cdots - u_\ell)$, determine whether they represent the same cycle (i.e., are equivalent). The complexity of your algorithm should be $O(k^2)$ in general, but $O(k)$ for when q is a simple cycle. NOTE: Assume that vertices are integers, and the closed path $p = (v_0 - \cdots - v_k)$ is represented by an array of integers $p[0..k]$, where $p[i] = v_i$ and $p[0] = p[k]$. $\diamondsuit$

End Exercises

§3. Graph Representation

The representation of graphs in computers is relatively straightforward if we assume array capabilities or pointer structures. The three main representations are:

- Edge list: this consists of a list of the vertices of G, and a list of the edges of G. The lists may be singly- or doubly-linked. If there are no isolated vertices, we may omit the vertex list. E.g., the edge list representations of the two graphs in Figure 3 would be

  "a–b" denotes an edge

$$\{a{-}b, b{-}c, c{-}d, d{-}a, d{-}b, c{-}e\}$$

  and

$$\{1{-}6, 2{-}1, 2{-}3, 2{-}6, 3{-}2, 3{-}6, 4{-}3, 4{-}6, 5{-}2, 5{-}3, 5{-}6\}.$$

- Adjacency list: a list of the vertices of G and for each vertex v, we store the list of vertices that are adjacent to v. If the vertices adjacent to u are $v_1, v_2, \ldots, v_m$, we may denote an adjacency list for u by $u : (v_1, v_2, \ldots, v_m)$. E.g., the adjacency list representation of the graphs in Figure 3 are

$$\{a : (b, d), b : (a, d, c), c : (b, d, e), d : (a, b, c), e : (c)\}$$

  and

$$\{1 : (5, 6), 2 : (1, 3, 6), 3 : (2, 6), 4 : (3, 6), 5 : (4, 6), 6 : ()\}$$

  Typically, we have an array $A[1..n]$ indexed by the vertices. Each array entry $A[v]$ points to the adjacency list for vertex v, represented by a linked list.

- Adjacency matrix: this is a $n \times n$ Boolean matrix where the (i, j)-th entry is 1 iff vertex j is adjacent to vertex i. E.g., the adjacency matrix representation of the graphs in Figure 3 are

$$\begin{array}{c|ccccc|}
a & 0 & 1 & 0 & 1 & 0 \\
b & 1 & 0 & 1 & 1 & 0 \\
c & 0 & 1 & 0 & 1 & 1 \\
d & 1 & 1 & 1 & 0 & 0 \\
e & 0 & 0 & 1 & 0 & 0 \\
 & a & b & c & d & e
\end{array}, \qquad
\begin{array}{c|cccccc|}
1 & 0 & 0 & 0 & 0 & 1 & 1 \\
2 & 1 & 0 & 1 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 1 \\
4 & 0 & 0 & 1 & 0 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}.$$

  Note that the matrix for bigraphs are symmetric. The adjacency matrix can be generalized to store arbitrary values to represent weighted graphs.

¶11. Size Parameters. Two size parameters are used in measuring the computational complexity of graph problems: $|V|$ and $|E|$. These are typically denoted by n and m. Thus the running time of graph algorithms are typically denoted by a function of the form $T(n, m)$. A linear time algorithm would have $T(n, m) = \mathcal{O}(m + n)$. It is clear that n, m are not independent, but satisfy the bounds $0 \le m \le n^2$. Thus, the edge list and adjacency list methods of representing graphs use $O(m + n)$ space while the last method uses $O(n^2)$ space.

If $m = o(n^2)$ for graphs in a family $\mathcal{G}$, we say $\mathcal{G}$ is a **sparse** family of graphs; otherwise the family is **dense**. Thus the adjacency matrix representation is not a space-efficient way to represent sparse graphs. Some algorithms can exploit sparsity of input graphs. For example, the family $\mathcal{G}$ of planar bigraphs is sparse because (as noted earlier) $m \le 3n - 6$ in such graphs (Exercise). Planar graphs are those that can be drawn on the plane without any crossing edges.

¶12. Arrays and Attributes. If A is an array, and $i \le j$ are integers, we write $A[i..j]$ to indicate that the array A has $j - i + 1$ elements which are indexed from i to j. Thus A contains the set of elements $\{A[i], A[i + 1], \ldots, A[j]\}$.

In description of graph algorithms, it is convenient to assume that the vertex set of a graph is $V = \{1, 2, \ldots, n\}$. The list structures can now be replaced by arrays indexed by the vertex set, affording great simplification in our descriptions. Of course, arrays also has more efficient access and use less space than linked lists. For instance, arrays allows us to iterate over all the vertices using an integer variable.

Often, we want to compute and store a particular **attribute** (or property) with each vertices. We can use an array $A[1..n]$ where $A[i]$ is the value of the A-attribute of vertex i. For instance, if the attribute values are real numbers, we often call $A[i]$ the "weight" of vertex i. If the attribute values are elements of some finite set, we may call $A[i]$ the "color" of vertex i.

¶13. Coloring Scheme. In many graph algorithms we need to keep track of the processing status of vertices. Initially, the vertices are unprocessed, and finally they are processed. We may need to indicate some intermediate status as well. Viewing the status as colors, we then have a three-color scheme: `white` or `gray` or `black`. They correspond to unprocessed, partially processed and completely processed statuses. Alternatively, the three colors may be called `unseen`, `seen` and `done` (resp.), or $0, 1, 2$. Initially, all vertices are unseen or white or 0. The color transitions of each vertex are always in this order:

$$\begin{array}{l} \texttt{white} \Rightarrow \texttt{gray} \Rightarrow \texttt{black}, \\ \texttt{unseen} \Rightarrow \texttt{seen} \Rightarrow \texttt{done} \\ 0 \Rightarrow 1 \Rightarrow 2. \end{array} \tag{2}$$

For instance, let the color status be represented by the integer array `color`$[1..n]$, with the convention that `white/unseen` is 0, `gray/seen` is 1 and `black/done` is 2. Then color transition for vertex i is achieved by the increment operation `color`$[i]$`++`. Sometimes, a two-color scheme is sufficient: in this case we omit the `gray` color or the `done` status.

Exercises

Exercise 3.1: The following is a basic operation for many algorithms: given a digraph G represented by adjacency lists, compute the reverse digraph G^{rev} in time $O(m + n)$. Recall

(Lecture 1, Appendix) that $u{-}v$ is an edge of G iff $v{-}u$ is an edge of G^{rev}. Show that your algorithm has the stated running time. ♢

Exercise 3.2: Let G is a planar bigraph.
(a) Show that if a planar embedding of G has f faces, then $v - e + f = n - m + f = 2$ where $v = n = |V|, e = m = |E|$. Thus, f is independent of the choice of embedding. HINT: use induction on f. Note that when $f = 1$, G is a free tree.
(b) Show that $2e \geq 3f$. HINT: Count the number of (edge-face) incidences in two ways: by summing over all edges, and by summing over all faces.
(c) Conclude that $e \leq 3v - 6$. When is equality attained? ♢

Exercise 3.3: The average degree of vertices in a planar bigraph is less than 6. ♢

Exercise 3.4: Let G be a planar bigraph with 60 vertices. What is the maximum number of edges it may have? ♢

Exercise 3.5: Prove that $K_{3,3}$ is nonplanar. HINT: Use the fact that every face of an embedding of $K_{3,3}$ is incident on at least 4 edges. NOTE: In a previous Exercises (Chapter 1, Appendix) we proved that K_5 is nonplanar. ♢

End Exercises

§4. Breadth First Search

A **graph traversal** is any systematic method to "visit" each vertex and each edge of a graph. In this section, we study two main traversal methods, known as BFS and DFS. The graph traversal problem may be traced back to the Greek mythology about threading through mazes (Theseus and the Minotaur legend), and to Trémaux's cave exploration algorithm in the 19th Century [3]. Tarjan's 1972 paper on DFS was seminal in Computer Science.

Hey, haven't we seen this before for trees?

Here is the generic graph traversal algorithm:

```
Generic Graph Traversal:
Input: $G = (V, E; s_0)$ where $s_0$ is source node
    Color all vertices as initially unseen.
    Mark $s_0$ as seen, and insert into $Q$
    While $Q$ is non-empty
        $u \leftarrow Q.Remove()$
        For each vertex $v$ adjacent to $u$
            If $v$ is unseen,
                color it as seen
                $Q$.insert($v$)
```

This algorithm will reach all nodes that are reachable from the source s_0. To visit all nodes, we can use another driver routine which invokes this traversal routine with different choices for source nodes (see Below).

The set Q is stored in some container data-structure. There are two standard containers: either a queue or a stack. These two data structures give rise to the two algorithms for graph traversal: **Breadth First Search** (BFS) and **Depth First Search** (DFS), respectively.

Both traversal methods apply to digraphs and bigraphs. However, BFS is often described for bigraphs only and DFS for digraphs only. We generally follow this tradition. In both algorithms, the input graph $G = (V, E; s_0)$ is represented by adjacency lists, and $s_0 \in V$ is called the **source** for the search.

The idea of BFS is to systematically visit vertices that are nearer to s_0 before visiting those vertices that are further away. For example, suppose we start searching from vertex $s_0 = a$ in the bigraph of Figure 3(a). From vertex a, we first visit the vertices b and d which are distance 1 from vertex a. Next, from vertex b, we find vertices c and d that are distance 1 away; but we only visit vertex c but not vertex d (which had already been visited). And so on. The trace of this search can be represented by a tree as shown in Figure 6(a). It is called the "BFS tree".

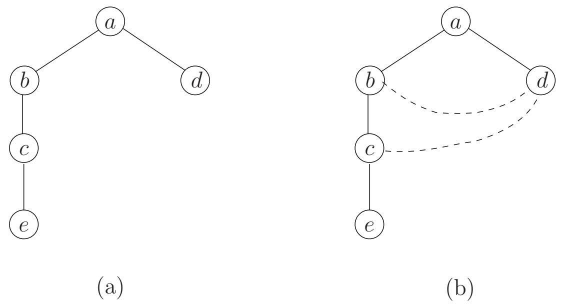


Figure 6: (a) BFS tree. (b) Non-tree edges.

More precisely, recall that $\delta(u, v)$ denote the (link) distance from u to v in a graph. The characteristic property of the BFS algorithm is that we will visit u before v whenever

$$\delta(s_0, u) < \delta(s_0, v) < \infty. \tag{3}$$

If $\delta(s_0, u) = \infty$, then u will not be visited from s_0. The BFS algorithm does not explicitly compute the relation (3) to decide the next node to visit: we will prove below that this is a consequence of using the queue data structure.

The key to the BFS algorithm is the **queue** ADT which supports the insertion and deletion of an item following the First-In First-Out (FIFO) discipline. If Q is a queue and x an item, we denote the insert and delete operations by

$$Q.\texttt{enqueue}(x), \quad x \leftarrow Q.\texttt{dequeue}(),$$

respectively. To keep track of the status of vertices we will use the color scheme in the previous section (see (2)). We could use three colors, but for our current purposes, two suffice: `white`/`gray` or `unseen`/`seen`.

We formulate the BFS algorithm as a "skeleton" or shell routine:

```
BFS Algorithm
     Input:   $G = (V, E; s_0)$ a graph (bi- or di-).
     Output:  This is application specific.
     ▷ Initialization:
0          $\boxed{\text{INIT}}(G, s_0)$   ◁ If this is standalone, then color all vertices except $s_0$ as unseen
1          Initialize the queue $Q$ to contain just $s_0$.
2          $\boxed{\text{VISIT}}(s_0, \mathsf{nil})$   ◁ Visit $v$ as root
     ▷ Main Loop:
           while $Q \neq \emptyset$ do
3               $u \leftarrow Q$.dequeue().   ◁ Begin processing $u$
4               for each $v$ adjacent to $u$ do   ◁ Process edge $u - v$
5                    $\boxed{\text{PREVISIT}}(v, u)$   ◁ Previsit $v$ from $u$
6                    if $v$ is unseen then
7                         Color $v$ seen
8                         $\boxed{\text{VISIT}}(v, u)$   ◁ Visit $v$ from $u$
9                         $Q$.enqueue($v$).
10              $\boxed{\text{POSTVISIT}}(u)$
```

Our shell program contains the following **shell macros**

$$\text{INIT, PREVISIT, VISIT and POSTVISIT} \qquad (4)$$

which will application-specific. These macros may be assumed[5] to be null operations unless otherwise specified. The term "macro" also suggests only small and local (i.e., $\mathcal{O}(1)$ time) modications. An application of BFS will amount to filling these shell macros with actual code. We can usually omit the PREVISIT step, but see §6 for an example of using this macro.

Note that VISIT(v, u) represents visiting v **from** u; a similar interpretation holds for PREVISIT(v, u). We allow $u = \mathsf{nil}$ in case v is the root of a BFS tree. If this BFS algorithm is a standalone code, then INIT(G, s_0) may be expected to initialize the color of all vertices to `unseen`, and s_0 has color `seen`. Otherwise, the initial coloring of vertices must be done before calling BFS.

There is an underlying tree structure in each BFS computation: the root is s_0. If v is `seen` from u (see Line 6 in the BFS Algorithm), then the edge $u - v$ is an edge in this tree. This tree is called the **BFS tree** (see Figure 6(a)). A **BFS listing at** s_0 is a list of all the vertices reachable from s_0 in which a vertex u appears before another vertex v in the list whenever (3) holds. E.g., let G be the bigraph in Figure 3(a) and s_0 is vertex a. Then two possible BFS listing at a are

$$(a, b, d, c, e) \qquad \text{and} \qquad (a, d, b, c, e). \qquad (5)$$

We can produce such a listing just by enumerating the vertices of the BFS tree in the order they are visited.

¶14. Applications of BFS. We now show how to program the shell macros in BFS to solve a variety of problems:

[5]Alternatively, we could fold the coloring steps into these macros, so that they may be non-null. But we choose to expose these coloring steps in our BFS shell.

- Suppose you wish to print a BFS listing of the vertices reachable from s_0. Then POSTVISIT(u) simply prints the key (or some identifier or name) at u. Other macros can remain null operations.
- Suppose you wish to compute the BFS tree T. If we view T as a set of edges, then INIT(G, s_0) could initialize the set T to be empty. In VISIT(v, u), we add the edge u–v to T.
- Suppose you wish to determine the depth $d[u]$ of each vertex u in the BFS Tree. Then INIT(G, s_0) could initialize
$$d[u] = \begin{cases} \infty & \text{if } u \neq s_0, \\ 0 & \text{if } u = s_0. \end{cases}$$
and in VISIT(v, u), we set $d[v] = 1 + d[u]$. Also, the coloring scheme (unseen/seen) could be implemented using the array $d[1..n]$ instead of having a separate array. More precisely, we simply use the special value $d[i] = -1$ to indicate unseen vertices; seen vertices satisfy $d[i] \geq 0$.

¶15. BFS Analysis. We shall derive basic properties of the BFS algorithm. These results will apply to both bigraphs and digraphs unless otherwise noted. The following two properties are often taken for granted:

Lemma 1.
(i) The BFS algorithms terminates.
(ii) Starting from source s_0, the BFS algorithm visits every node reachable from s_0.

We leave its proof for an Exercise. For instance, this assures us that each vertex of the BFS tree will eventually become the front element of the queue.

Let $\delta(v) \geq 0$ denote the **depth** of a vertex v in the BFS tree. Note that if v is visited from u, then $\delta(v) = \delta(u) + 1$. We prove a key property of BFS:

Lemma 2 (Monotone $0-1$ Property). *Let the vertices in the queue Q at some time instance be $(u_1, u_2, \ldots, u_k)$ for some $k \geq 1$, with u_1 the earliest enqueued vertex and u_k the last enqueued vertex. The following invariant holds:*

$$\delta(u_1) \leq \delta(u_2) \leq \cdots \leq \delta(u_k) \leq 1 + \delta(u_1). \tag{6}$$

Proof. The result is clearly true when $k = 1$. Suppose $(u_1, \ldots, u_k)$ is the state of the queue at the beginning of the while-loop, and (6) holds. In Line 3, we removed u_1 and assign it to the variable u. Now the queue contains $(u_2, \ldots, u_k)$ and clearly, it satisfies the corresponding inequality

$$\delta(u_2) \leq \delta(u_3) \leq \cdots \leq \delta(u_k) \leq 1 + \delta(u_2).$$

Suppose in the for-loop, in Line 9, we enqueued a node v that is adjacent to $u = u_1$. Then Q contains $(u_2, \ldots, u_k, v)$ and we see that

$$\delta(u_2) \leq \delta(u_3) \leq \cdots \leq \delta(u_k) \leq \delta(v) \leq 1 + \delta(u_2)$$

holds because $\delta(v) = 1 + \delta(u_1) \leq 1 + \delta(u_2)$. In fact, every vertex v enqueued in this for-loop preserves this property. This proves the invariant (6). **Q.E.D.**

This lemma shows that $\delta(u_i)$ is monotone non-decreasing. Indeed, $\delta(u_i)$ will remain constant throughout the list, except possibly for a single jump to the next integer. Thus, it has this "$0-1$ property", that $\varepsilon_j := \delta(u_{j+1}) - \delta(u_j) = 0$ or 1 for all $j = i, \ldots, k-1$. Moreover, there is at most one j such that $\varepsilon_j = 1$. From this lemma, we deduce other basic properties the BFS algorithm:

Lemma 3. *For each vertex u in the BFS Tree,*

$$\delta(u) = \delta(s_0, u),$$

i.e., $\delta(u)$ is the link distance from s_0 to u.

Proof. Let $\pi : (u_0 - u_1 - u_2 - \cdots - u_k)$ be a shortest path from $u_0 = s_0$ to $u_k = u$ of length $k \geq 1$. It is sufficient to prove that $\delta(u_k) = k$. For $i \geq 1$, lemma 2 tells us that $\delta(u_i) \leq \delta(u_{i-1}) + 1$. This implies $\delta(u_k) \leq k + \delta(u_0) = k$. On the other hand, the inequality $\delta(u_k) \geq k$ is immediate because, $\delta(s_0, u_k) = k$ by our choice of π, and $\delta(u_k) \geq \delta(s_0, u_k)$ because there is a path of length $\delta(u_k)$ from s_0 to u_k. **Q.E.D.**

As corollary: *if we print the vertices $u_1, u_2, \ldots, u_k$ of the BFS tree, in the order that they are enqueued, this would represent a BFS listing.* This is because $\delta(u_i)$ is non-decreasing with i, and $\delta(u_i) = \delta(s_0, u_i)$.

Another basic property is:

Lemma 4. *If $\delta(u) < \delta(v)$ then u is VISITed before v is VISITed, and u is POSTVISITed before v is POSTVISITed.*

¶16. Classification of bigraph edges. Let us now consider the case of a bigraph G. The edges of G can be classified into the following types by the BFS Algorithm (cf. Figure 6(b)):

- **Tree edges**: these are the edges of the BFS tree.
- **Level edges**: these are edges between vertices in the same level of the BFS tree. E.g., edge *bd* in Figure 6(b).
- **Cross-Level edges**: these are non-tree edges that connect vertices in two different levels. But note that the two levels differ by exactly one. E.g., edge *cd* in Figure 6(b).
- **Unseen edges**: these are edges that are not used during the computation. The involved vertices not reachable from s_0.

Each of these four types of edges can arise (see Figure 6(b) for tree, level and cross-level edges). But is the classification complete (i.e., exhaustive)? It is, because any other kind of edges must connect vertices at non-adjacent levels of the BFS tree, and this is forbidden by Lemma 3. Hence we have:

Theorem 5. *If G is a bigraph, the above classification of edges is complete.*

We will leave it as an exercise to fill in our BFS shell macros to produce the above classification of edges.

¶17. Driver Program. In our BFS algorithm we assume that a source vertex $s_0 \in V$ is given. This is guaranteed to visit all vertices reachable from s_0. What if we need to process all vertices, not just those reachable from a given vertex? In this case, we write a "driver program" that repeatedly calls our BFS algorithm. We assume a global initialization which sets all vertices to `unseen`. Here is the driver program:

```
BFS Driver Algorithm
      Input:    $G = (V, E)$ a graph.
      Output:   Application-dependent.
      ▷ Initialization:
1           Color all vertices as unseen.
2           $\boxed{\text{GLOBAL\_INIT}}(G)$
      ▷ Main Loop:
3           for each vertex $v$ in $V$ do
4                 if $v$ is unseen then
5                       call BFS($(V, E; v)$).
```

Note that with the BFS Driver, we add another shell macro called GLOBAL_INIT to our collection (4).

¶18. Time Analysis. Let us determine the time complexity of the BFS Algorithm and the BFS Driver program. We will discount the time for the application-specific macros; but as long as these macros are $O(1)$ time, our complexity analysis remains valid. Also, it is assumed that the Adjacency List representation of graphs is used. The time complexity will be given as a function of $n = |V|$ and $m = |E|$.

Here is the time bound for the BFS algorithm: the initialization is $O(1)$ time and the main loop is $\Theta(m')$ where $m' \leq m$ is the number of edges reachable from the source s_0. This giving a total complexity of $\Theta(m')$.

Next consider the BFS Driver program. The initialization is $\Theta(n)$ and line 3 is executed n times. For each actual call to BFS, we had shown that the time is $\Theta(m')$ where m' is the number of reachable edges. Summing over all such m', we obtain a total time of $\Theta(m)$. Here we use the fact the sets of reachable edges for different calls to the BFS routine are pairwise disjoint. Hence the Driver program takes time $\Theta(n + m)$.

¶19. Application: Computing Connected Components. Suppose we wish to compute the connected components of a bigraph G. Assuming $V = \{1, \ldots, n\}$, let us encode this task as computing an integer array $C[1..n]$ satisfying the property $C[u] = C[v]$ iff u, v belongs to the same component. Intuitively, $C[u]$ is the name of the component that contains u. The component number is arbitrary.

To accomplish this task, we assume a global variable called `count` that is initialized to 0 by GLOBAL_INIT(G). Inside the BFS algorithm, the INIT(G, s_0) macro simply increments the `count` variable. Finally, the VISIT(v, u) macro is simply the assignment, $C[v] \leftarrow$ `count`. The correctness of this algorithm should be clear. If we want to know the number of components in the graph, we can output the value of `count` at the end of the driver program.

¶20. Application: Testing Bipartiteness. A graph $G = (V, E)$ is **bipartite** if V can be partititioned into $V = V_1 \uplus V_2$ such that $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$. Note that this definition applies to digraphs as well as bigraphs. It is clear that all cycles in a bipartite graphs must be **even** (i.e., has an even number of edges). The converse is shown in an Exercise: if G has no **odd cycles** then G is bipartitite. We use the Driver Driver to call call BFS($V, E; s$) for various s. It is sufficient to show how to detect odd cycles in the component of s. If there is a level-edge (u, v), then we have found an odd cycle: this cycle comprise the tree path from the root to u, the edge $(u{-}v)$, and the tree path from v back to the root. In the exercise, we ask you to show that all odd cycles is represented by such level-edges. It is now a simple matter to modify BFS to detect level-edges.

In trying to implement the Bipartitite Test above, and especially in recursive routines, it is useful to be able to jump out of nested macro and subroutine calls. For this, the `Java`'s ability to **throw exceptions** and to **catch exceptions** is very useful. In our bipartite test, BFS can immediately throw an exception when its finds a level-edge. This exception is caught by the BFS Driver program.

Exercises

Exercise 4.1: Prove Lemma 1: that the BFS algorithm terminates, and every vertex that is reachable from s_0 will be seen by BFS(s_0). ♢

Exercise 4.2: Show that each node is VISITed and POSTVISITed at most once. Is this true for PREVISIT as well? ♢

Exercise 4.3: Assume $u{-}v$.
(a) Show that $\delta(v) \le 1 + \delta(u)$.
(b) Show that in digraphs, the inequality in (a) can be arbitarily far from an equality.
(c) Show that in bigraphs, $|\delta(u) - \delta(v)| \le 1$. ♢

Exercise 4.4: Reorganize the BFS algorithm so that the coloring steps are folded into the shell macros of INIT, VISIT, etc. ♢

Exercise 4.5: Fill in the shell macros so that the BFS Algorithm will correctly classify every edge of the input bigraph. ♢

Exercise 4.6: Classification the edges of a digraph relative to a given BFS tree. ♢

Exercise 4.7: Let $G = (V, E; \lambda)$ be a connected bigraph in which each vertex $v \in V$ has an associated value $\lambda(v) \in \mathbb{R}$.
(a) Give an algorithm to compute the sum $\sum_{v \in V} \lambda(v)$.
(b) Give an algorithm to label every edge $e \in E$ with the value $|\lambda(u) - \lambda(v)|$ where $e = u{-}v$. ♢

Exercise 4.8: Give an algorithm that determines whether or not a bigraph $G = (V, E)$ contains a cycle. Your algorithm should run in time $O(|V|)$, independent of $|E|$. You must use the shell macros, and also justify the claim that your algorithm is $O(|V|)$. ♢

Exercise 4.9: The text sketched an algorithm for testing if a graph is bipartite. We verify some of the assertions there:
(a) Prove that if a bigraph has no odd cycles, then it is bipartite.
(b) Prove that if a connected graph has an odd cycle, then BFS search from any source vertex will detect a level-edge.
(c) Write the pseudo code for bipartite test algorithm outlined in the text. This algorithm is to return YES or NO only. You only need to program the shell routines.
(d) Modify the algorithm in (c) so that in case of YES, it returns a Boolean array $B[1..n]$ such that $V_0 = \{i \in V : B[i] = \mathsf{false}\}$ and $V_1 = \{i \in V : B[i] = \mathsf{true}\}$ is a witness to the bipartiteness of G. In the case of NO, it returns an odd cycle. ♢

Exercise 4.10: Let G be a digraph. A **global sink** is a node u such that for every node $v \in V$, there is path from v to u. A **global source** is a node u such that for every node $v \in V$, there is path from u to v.
(a) Assume G is a DAG. Give a simple algorithm to detect if G has a global sink and a global source. Your algorithm returns YES if both exists, and returns NO otherwise. Make sure that your algorithm takes $O(m + n)$ time.
(b) Does your algorithm work if G is not a DAG? If not, give a counter example which makes your algorithm fail. ♢

Exercise 4.11: Let $k \ge 1$ be an integer. A **k-coloring** of a bigraph $G = (V, E)$ is a function $c : V \to \{1, 2, \ldots, k\}$ such that for all u–v in E, $c(u) \ne c(v)$. We say G is **k-colorable** if G has a k-coloring. We say G is **k-chromatic** if it is k-colorable but not $(k-1)$-colorable. Thus, a graph is bipartite iff it is 2-colorable.
(a) How do you test the 3-colorability of bigraphs if every vertex has degree ≤ 2?
(b) What is the smallest graph which is not 3-colorable?
(c) The **subdivision** of an edge u–v is the operation where the edge is deleted and replaced by a path u–w–v of length 2 and w is a new vertex. Call G' a subdivision of another graph G if G' is obtained from G be a finite sequence of edge subdivisions. Dirac (1952) shows that G is 4-chromatic, then it contains a subdivision of K_4. Is there a polynomial time to determine if a given connected bigraph G contains a subdivision of K_4? ♢

End Exercises

§5. Simple Depth First Search

The DFS algorithm turns out to be more subtle than BFS. In some applications, however, it is sufficient to use a simplified version that is as easy as the BFS algorithm. In fact, it might even be easier because we can exploit recursion.

Here is an account of this simplified DFS algorithm. As in BFS, we use a 2-color scheme: each vertex is `unseen` or `seen`. We similarly define a **DFS tree** underlying any particular DFS computation: the edges of this tree are precisely those u–v such that v is `seen` from u. Starting the search from the source s_0, the idea is to go as deeply as possibly along any path *without visiting any vertex twice*. When it is no longer possible to continue a path, we backup towards the source s_0. But we only backup enough for us to go forward in depth again.

In illustration, suppose G is the digraph in Figure 3(b), and s_0 is vertex 1. One possible deepest path from vertex 1 is (1–5–2–3–6). From vertex 6, we backup to vertex 2, from where we can advance to vertex 3. Again we need to backup, and so on. The DFS tree is a trace of this search process; this tree is shown in Figure 7(a). The non-tree edges of the graph are shown in dashed lines. For the same graph, if we visit adjacent nodes in a different order, we get a different DFS tree, as in Figure 7(b). However, the DFS tree in Figure 7(a) is the unique solution if we follow our usual convention of visiting vertices with smaller indices first.

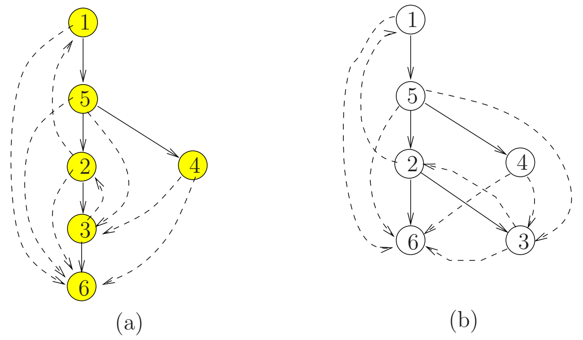


Figure 7: Two DFS Trees for the digraph in Figure 3(b).

The simple DFS algorithm is compactly presented using recursion as follows:

```
SIMPLE DFS
    Input:   $G = (V, E; s_0)$ a graph (bi- or di-)
             The vertices in $V$ have been colored seen or unseen.
    Output   Application dependent
1       Color $s_0$ as seen.
2       for each $v$ adjacent to $s_0$ do
3           $\boxed{\text{PREVISIT}}(v, s_0)$
4           if $v$ is unseen then
5               Color $v$ seen.
6               $\boxed{\text{VISIT}}(v, s_0)$.
7               Simple DFS($(V, E; v)$)     ◁ Recursive call
8       $\boxed{\text{POSTVISIT}}(s_0)$.
```

In this recursive version, there is no INIT(G, s_0) step — we do not want to initialize G with every recursive call. Also, VISIT(v, u) is called just before each DFS call. But we could consider a variant[6] in which we do the VISIT after the recursive call (i.e., interchange lines 6 and 7). As in BFS, we choose to expose the coloring steps rather than putting them inside the shell macros. The first call to this DFS algorithm must be made by some DFS Driver Program which performs the necessary setup:

[6]We could also entertain a visit *before*, and another visit *after*, the recursive call.

```
DFS Driver
      Input:   $G = (V, E)$ a graph (bi- or dip)
      Output:  Application-specific
1         $\boxed{\text{GLOBAL\_INIT}}(G)$
2         Color each vertex in $V$ as unseen.
3         for each $v$ in $V$ do
4               if $v$ is unseen then
5                     $\boxed{\text{INIT}}(G, v)$
6                     Simple DFS$(V, E; v)$     ◁ recursive call
7                     $\boxed{\text{CLEANUP}}(V, E; v)$
```

As in the BFS case, we view both the above algorithms as algorithmic skeletons, and their complete behaviour will depend on the specification of the shell macros,

$$\text{PREVISIT, VISIT, POSTVISIT, GLOBAL\_INIT, INIT, CLEANUP.} \tag{7}$$

These shell macros may be assumed to be null operations unless otherwise specified.

¶21. DFS Tree. The root of the DFS tree is s_0, and the vertices of the tree are those vertices visited during this DFS search (see Figure 7(a)). This tree can easily be constructed by appropriate definitions of INIT(G, s_0), VISIT$(v{-}u)$ and POSTVISIT(u), and is left as an Exercise. We prove a basic fact about DFS:

Lemma 6 (Unseen Path). *Let $u, v \in V$. Then v is a descendent of u in the DFS tree if and only if at the time that u was first seen, there is*[7] *a "unseen path" from u to v, i.e., a path $(u{-}\cdots{-}v)$ comprising only of unseen vertices.*

Proof. Let t_0 be the time when we first see u.

($\Rightarrow$) We first prove the easy direction: if v is a descendent of u then there is an unseen path from u to v at time t_0. For, if there is a path $(u{-}u_1{-}\cdots{-}u_k{-}v)$ from u to v in the DFS tree, then each u_i must be unseen at the time we first see u_{i-1} ($u_0 = u$ and $u_{k+1} = v$). Let t_i be the time we first see u_i. Then we have $t_0 < t_1 < \cdots < t_{k+1}$ and thus each u_i was unseen at time t_0. Here we use the fact that each vertex is initially unseen, and once seen, will never revert to unseen.

($\Leftarrow$) We use an inductive proof. The subtlety is that the DFS algorithm has its own order for visiting vertices adjacent to each u, and your induction must somehow account for this order. We proceed by defining a total order on all paths from u to v: If a, b are two vertices adjacent to a vertex u and we visit a before b, then we say "$a <_{\texttt{dfs}} b$ (relative to u)". If $p = (u{-}u_1{-}u_2{-}\cdots{-}u_k{-}v)$ and $q = (u{-}v_1{-}v_2{-}\cdots{-}v_\ell{-}v)$ (where $k, \ell \ge 0$) are two distinct paths from u to v, we say $p <_{\texttt{dfs}} q$ if there is an m ($1 \le m < \min\{k, \ell\}$) such that $u_1 = v_1, \ldots, u_m = v_m$ and $u_{m+1} < v_{m+1}$ relative to u_m. Note that m is well-defined. Now define the **DFS-distance** between u and v to be the length of the $<_{\texttt{dfs}}$-least *unseen path* from u to v at time we first see u. By an **unseen path** from u to v, we mean one

$$\pi : (u{-}u_1{-}\cdots{-}u_k{-}v) \tag{8}$$

[7] If we use the white-black coloring scheme, this would be called a "white path" as in [2].

where each node $u_1, \ldots, u_k, v$ is unseen at time when we first see u. If there are no unseen paths from u to v, the DFS-distance from u to v is infinite.

For any $k \in \mathbb{N}$, let IND(k) be the statement: "If the DFS-distance from u to v has length $k+1$, and (8) is the $<_{\texttt{dfs}}$-least unseen path from u to v, then this path is a path in the DFS tree". Hence our goal is to prove the validity of IND(k).

BASE CASE: Suppose $k = 0$. The $<_{\texttt{dfs}}$-least unseen path from u to v is just $(u{-}v)$. So v is adjacent to u. Suppose there is a vertex v' such that $v' <_{\texttt{dfs}} v$ (relative to u). Then there does not exist an unseen path π' from v' to v; otherwise, we get the contradiction that the path $(u{-}v'); \pi'$ is $<_{\texttt{dfs}}$ than than $(u{-}v)$). Hence, when we recursively visit v', we will never color v as `seen` (using the easy direction of this lemma). Hence, we will eventually color v as `seen` from u, *i.e.*, $u{-}v$ is an edge of the DFS tree.

INDUCTIVE CASE: Suppose $k > 0$. Let π in (8) be the $<_{\texttt{dfs}}$-least unseen path of length $k+1$ from u to v. As before, if $v' <_{\texttt{dfs}} u_1$ then we will recursively visit v', we will never color any of the vertices $u_1, u_2, \ldots, u_k, v$ as `seen`. Therefore, we will eventually visit u_1 from u at some time $t_1 > t_0$. Moreover, the sub path $\pi' : (u_1{-}u_2{-}\cdots{-}u_k{-}v)$ is still `unseen` at this time. Moreover, π' remains the $<_{\texttt{dfs}}$-least unseen path from u_1 to v at time t_1. By IND($k-1$), the subpath π' is in the DFS tree. Hence the path $\pi = (u{-}u_1); \pi'$ is in the DFS tree. **Q.E.D.**

¶22. Classification of edges. First consider a digraph G. Upon calling $DFS(G, s_0)$, the edges of G becomes classified as follows (see Figure 7(b)):

- **Tree edges**: these are the edges belonging to the DFS tree.
- **Back edges**: these are non-tree edges $u{-}v \in E$ where v is an ancestor of u. Note: $u{-}u$ is considered a back edge. E.g., edges $2{-}1$ and $3{-}2$ in Figure 7(b).
- **Forward edges**: these are non-tree edges $u{-}v \in E$ where v is a descendent of u. E.g., edges $1{-}6$ and $5{-}6$ in Figure 7(b).
- **Cross edges**: these are edges $u{-}v$ that are not classified by the above, but where u, v are visited. E.g., edges $4{-}6$, $3{-}6$ and $4{-}3$ in Figure 7(b).
- **Unseen edges**: all other edges are put in this category. These are edges $u{-}v$ in which u is unseen at the end of the algorithm.

¶23. DFS of Bigraphs. As we noted, DFS applies to bigraphs as well as digraphs. There are two ways to view such bigraphs in DFS: (1) One is to view the bigraph as a digraph whose directed edges happen to come in pairs of the form (u, v) and (v, u), one such pair for each undirected $u{-}v$. (2) Adopt the convention that an undirected edge $\{u, v\}$ will regarded as the directed edge $(u{-}v)$ if u is seen before v. Then the other edge $(v{-}u)$ will not appear as a tree or non-tree edge. If u, v remain unseen, then $\{u, v\}$ will remain undirected. Call (2) our **standard treatment** of bigraph edges, and it will be assumed unless otherwise noted. The classification of bigraph edges under (2) will be simplified; see Exercises.

Unfortunately, our simple DFS algorithm cannot easily determine these edge classification. In particular, the bicolor scheme (seen/unseen) is no longer sufficient. E.g., we cannot distinguish a cross edge from a forward or back edge. We will defer the problem of classifying edges of the DFS tree to the next section.

¶24. Biconnectivity. When we discussed reduced graph above, we said that it is not a useful concept for bigraphs. We now introduce the appropriate analogue for bigraphs.

Let G be a bigraph $G = (V, E)$. A subset $C \subseteq V$ is a **biconnected set** of G if for every pair u, v of distinct vertices in C, there is a simple cycle of length ≥ 3 containing u and v. If, in addition, C is not properly contained in a biconnected subset of G, then we call C a **biconnected component**. If G has only one biconnected component, then G is called a **biconnected graph**. Biconnectivity is clearly a strong notion of connectivity.

Trivially, any subset of size 1 is biconnected; no subset of size 2 can be biconnected. E.g., the graph in Figure 3(a), has two biconnected components, $\{a, b, c, d\}$ and $\{e\}$. E.g., the complete graph K_n and cyclic graph C_n (see Figure 4(a),(d)) consists of a single biconnected component.

A vertex v of G is called[8] a **cut-vertex** if the removal of v, and also all edges incident on v, will increase the number of connected (not biconnected!) components of resulting bigraph. Alternatively, if there exist vertices a, b (both different from v) such that all paths from a to b must pass through v. Clearly, if G has a cut-vertex, then it is not biconnected. The converse is also true. There is an edge analogue of cut-vertex: an edge $u-v$ is called a **bridge** if the removal of this edge will increase the number of connected components of the resulting bigraph.

E.g., in the line graph L_n (see Figure 4(c)) with vertex set $V = \{1, \ldots, n\}$, a vertex i is a cut-vertex iff $1 < i < n$. Also, every edge of L_n is a bridge. The graph in Figure 3(a), has one cut-vertex c and one bridge $c-e$.

Note that two biconnected components of G can share at most one vertex, which is necessarily a cut-vertex. Given a bigraph G, we define[9] a bigraph $G^c = (V^c, E^c)$ such that the elements of V^c are the biconnected components of G, and $(C, C') \in E^c$ iff $C \cap C'$ is non-empty. It follows from the preceding remark that two biconnected components share at most one vertex that G^c is acyclic. We may call G^c the **reduced graph** for G.

Assume G is connected, and T is a DFS tree of G. There are two ways in which a vertex u is a cut-vertex, as shown in the following lemma:

Lemma 7. *Let u be a vertex in the DFS tree T. Then u is a cut-vertex iff one of the following conditions hold:*
(i) If u is the root of T and has more than one child.
(ii) If u is not the root, but it has a child u' such that for every descendent v of u, if $v-w$ is an edge, then w is also a descendent of u. Note that a node is always a descendent of itself.

If (i) or (ii) holds, it is easy to see that u must be a cut-vertex. But one can also show the converse (Exercise).

There is another property we need: suppose $v-w$ is a non-tree edge. Then we claim that v is a descendent of w or vice-versa, and in fact, this edge appears as a back edge in the classification of DFS tree edges. It is now an exercise to program the shell macros of the DFS algorithm to detect cut-vertices, and hence recognize biconnectivity.

[8] Or, articulation point.

[9] Recall that G^c is the same notation used for the reduced graph of a digraph G. Since we will not apply the reduced graph concept to bigraphs, and we will not apply the concept of biconnectedness to digraphs, there should be no confusion in reusing this G^c notation.

¶25. Connection with BFS. There is a sense in which BFS and DFS are the same search strategies except for their use of a different container ADT. Basically, recursion is an implicit way to use the **stack** ADT. The stack ADT is similar to the queue ADT except that the insertion and deletion of items into the stack are based on the Last-In-First-Out (LIFO) discipline. These two operations are denoted

$$S.\texttt{push}(x), \quad x \leftarrow S.\texttt{pop}(),$$

where S is a stack and x an item.

It is instructive to try to make this connection between the DFS and BFS algorithms more explicit. The basic idea is to avoid recursion in DFS, and to explicitly use a stack in implementing DFS. Let us begin with a simple experiment: what if we simply replace the queue ADT in BFS by the stack ADT? Here is the hybrid algorithm which we may call **BDFS**, obtained *mutatis mutandis* from BFS algorithm:

```
BDFS Algorithm
     Input:   $G = (V, E; s_0)$ a graph.
     Output:  Application specific
     ▷ Initialization:
0         Initialize the stack $S$ to contain $s_0$.
1         $\boxed{\text{INIT}}(G, s_0)$    ◁ If standalone, make all vertices unseen except for $s_0$
     ▷ Main Loop:
          while $S \neq \emptyset$ do
2              $u \leftarrow S.\texttt{pop}()$.
3              for each $v$ adjacent to $u$ do
4                   $\boxed{\text{PREVISIT}}(v, u)$
5                   if $v$ is unseen then
6                        color $v$ seen
7                        $\boxed{\text{VISIT}}(v, u)$
8                        $S.\texttt{push}(v)$.
9              $\boxed{\text{POSTVISIT}}(u)$
```

This algorithm shares properties of BFS and DFS, but is distinct from both. Many standard computations can still be accomplished using BDFS. To write a non-recursive version of DFS using this framework, we need to make several changes.

Let $S.\texttt{top}()$ refer to the top element of the stack. The invariant is that the sequence of vertices in the stack is path to the current vertex $curr$. Assume that we have two functions $first(u)$ and $next(u, v)$ that gives enables us to iterate over the adjacency list of u: $first(u)$ returns the first vertex that is adjacent to u, and $next(u, v)$ returns the next vertex after v that is adjacent to u (assuming v is adjacent to u). Both functions may return a null pointer, and also $next(\mathsf{nil}, v) = \mathsf{nil}$.

```
NONRECURSIVE DFS ALGORITHM
      Input:    $G = (V, E; s_0)$ a graph.
      Output:   Application specific
      ▷ Initialization:
0         Initialize the stack $S$ to contain $s_0$.
1         $\boxed{\text{INIT}}(G, s_0)$;   ◁ If standalone, make all vertices unseen except for $s_0$
2         $curr \leftarrow first(s_0)$;
      ▷ Main Loop:
3         while $S \neq \emptyset$ do
4             if ($curr \neq$ nil)
5                 $\boxed{\text{PREVISIT}}(curr, S.\texttt{top}())$
6                 if $curr$ is unseen
7                     color $curr$ seen
8                     $\boxed{\text{VISIT}}(curr, S.\texttt{top}())$
9                     $S.\texttt{push}(curr)$
10                    $curr \leftarrow first(curr)$
11                else $curr \leftarrow next(S.\texttt{top}(), curr)$
12            else
13                $curr \leftarrow S.\texttt{pop}()$
14                $\boxed{\text{POSTVISIT}}(curr)$
15                $curr \leftarrow next(S.\texttt{top}(), curr)$   ◁ may be nil
```

We leave it as an exercise to prove that this code is equivalent to the Simple (recursive) DFS algorithm.

Exercises

Exercise 5.1:
(a) Give the appropriate definitions for INIT(G), VISIT((v, u)) and POSTVISIT(u) so that our DFS Algorithm computes the DFS Tree, say represented by a data structure T
(b) Prove that the object T constructed in (a) is indeed a tree, and is the DFS tree as defined in the text. ♢

Exercise 5.2: Programming in the straightjacket of our shell macros is convenient when our format fits the application. But the exact placement of these shell macros, and the macro arguments, may sometimes require some modifications.
(a) We have defined VISIT(u, v) to take two arguments. Show that we could have defined this it as VISIT(u), and not lost any functionality in our shell programs. HINT: take advantage of PREVISIT(u, v).
(b) Give an example where it is useful for the Driver to call CLEANUP(u) after DFS(u). ♢

Exercise 5.3: Explore the relationship between the traversals of binary trees and DFS.
(a) Why are there not two versions of DFS, corresponding to pre- and postorder tree traversal? What about inorder traversal?
(b) Give the analogue of DFS for binary trees. As usual, you must provide place holders for shell routines. Further assume that the DFS returns some values which is processed at the appropriate place. ♢

Exercise 5.4: Why does the following variation of the recursive DFS fail?

```
Simple DFS (recursive form)
    Input:    $G = (V, E; s_0)$ a graph.
1       for each $v$ adjacent to $s_0$ do
2           if $v$ is unseen then
3               $\boxed{\text{VISIT}}(v, s_0)$.
4               Simple DFS($(V, E; v)$)
5       $\boxed{\text{POSTVISIT}}(s_0)$.
6       Color $s_0$ as seen.
```

◊

Exercise 5.5: Give an alternative proof of the Unseen Path Lemma, without explicitly invoking the ordering properties of $<_{\texttt{dfs}}$. Also, do not invoke properties of the Full DFS (with time stamps). ◊

Exercise 5.6: Prove that our classification of edges for DFS is complete. ◊

Exercise 5.7: Suppose T is the DFS Tree for a connected bigraph G. Recall our standard treatment of edges of a bigraph in DFS. Let $u{-}v$ be an edge of G. Prove that
(a) Either u is an ancestor of v or vice-versa in the tree T.
(b) If $u{-}v$ is a non-tree edge, it is a back edge.
(c) Give a complete classification of the edges as produced by the DFS algorithm. ◊

Exercise 5.8: Use the characterization of cut-vertices in Lemma 7 to design an algorithm to detect cut-vertices in a bigraph.

HINT: Let $ft(u)$ be the smallest value of $\texttt{firstTime}[w]$, where w is a vertex that can be reached by a back edge $v{-}w$, for some proper descendent v of u in the DFT tree; if there is no such back edge, then we define $ft(u)$ to be $\texttt{firstTime}[u]$. You need to address two questions: (a) How can $ft(u)$ help you determine whether a vertex v is a cut-vertex? (b) How can you compute $sft(u)$? ◊

Exercise 5.9: Let $G = (V, E)$ be a connected bigraph. For any vertex $v \in V$ define

$$\text{radius}(v, G) := \max_{u \in V} \text{distance}(u, v)$$

where distance(u, v) is the length of the shortest path from u to v. The *center* of G is the vertex v_0 such that radius(v_0, G) is minimized. We call radius(v_0, G) the *radius* of G and denote it by radius(G). Define the *diameter* diameter(G) of G to be the maximum value of distance(u, v) where $u, v \in V$.
(a) Prove that $2 \cdot \text{radius}(G) \geq \text{diameter}(G)$.
(b) Show that for every natural number n, there are graphs G_n and H_n such that $n = \text{radius}(G_n) = \text{diameter}(G_n)$ and $n = \text{radius}(H_n) = \lceil \text{diameter}(H_n)/2 \rceil$.
(c) Using DFS, give an efficient algorithm to compute the diameter of a undirected tree (i.e., connected acyclic undirected graph). Please use shell programming. Prove the correctness of your algorithm. What is the complexity of your algorithm? ◊

Exercise 5.10: Re-do the previous question (part (c)) to compute the diameter, but instead of using DGS, use BFS. $\diamondsuit$

Exercise 5.11: Prove that our nonrecursive DFS algorithm is equivalent to the recursive version. $\diamondsuit$

Exercise 5.12: When might we prefer the BDFS Algorithm in place of the standard DFS or BFS algorithms? $\diamondsuit$

END EXERCISES

§6. Full Depth First Search

To perform certain computations in the DFS framework, it is useful to compute additional information about the DFS tree. In particular, we may wish to classify the edges as described in the previous algorithm. Instead of the bicolor scheme, we tricolor each vertex, e.g., `unseen`/`seen`/`done`. The `seen` vertices are those currently in the recursion stack. The POSTVISIT(u) macro can be used to color the vertex u as `done`.

A more profound embellishment is to **timestamp** the vertices. There are two kinds of time stamp for each vertex time when first encountered, and time when last encountered. To implement timestamps, we assume a global counter `clock` that is initially 0. Also, we introduce two arrays, `firstTime`[v] and `lastTime`[v] where $v \in V$. Both arrays are initiallized to -1. When we see the vertex v for the first time or the last time, the current value of `clock` will be assigned to these array entries; the value of `clock` will be incremented after such an assignment.

More precisely, we may use the SIMPLE DFS and the DFS DRIVER shells and the following macro definitions:

- GLOBAL_INIT(G)$\equiv$ `clock` $\leftarrow 0$; (for $v \in V$)[`firstTime`[v] $\leftarrow$ `lastTime`[v] $\leftarrow -1$].
- VISIT(v, u)$\equiv$ `firstTime`[v] $\leftarrow$ `clock++`.
- POSTVISIT(v)$\equiv$ `lastTime`[v] $\leftarrow$ `clock++`.

The tricolor scheme is subsumed by this timestamp scheme: a node is `unseen` if the current value of `firstTime`[v] is -1; a node v is `seen` while the current values of `firstTime`[v] is ≥ 0 and `lastTime`[v] is -1; it is `done` if the current value `lastTime`[v] is ≥ 0.

In some applications, we may only need one of these two time values. Let `active`(u) denote the time interval [`firstTime`[u], `lastTime`[u]], and we say u is **active** within this interval. It is clear from the nature of the recursion that two active intervals are either disjoint or has a containment relationship. In case of non-containment, we may write `active`(v) $<$ `active`(u) if `lastTime`[v] $<$ `firstTime`[u]. We have the following characterization of edges using timestamps:

LEMMA 8. *Let $u, v \in V$. Then v is a descendent of u in the DFS tree if and only if*

$$\texttt{active}(v) \subseteq \texttt{active}(u).$$

Proof. This result can be shown using the Unseen Path Lemma. If there is a unseen path, then by induction on the length of this path, every vertex on this path will be a descendent of u. Conversely, if v is descendent of u then by induction on the distance of v from u, there will be a unseen path to u.

Now, if there is a unseen path from u to v when u was first discovered, we must have $\texttt{firstTime}[u] < \texttt{firstTime}[v]$. Moreover, since the vertex u will remain active until v is discovered, we also have $\texttt{lastTime}[v] < \texttt{lastTime}[u]$. Hence $\texttt{active}(v) \subseteq \texttt{active}(u)$. **Q.E.D.**

We return to the problem of classifying every edge of a digraph G relative to a DFS tree on G:

Lemma 9. *If $u{-}v$ is an edge then*

1. *$u{-}v$ is a back edge iff* $\texttt{active}(u) \subseteq \texttt{active}(v)$.
2. *$u{-}v$ is a cross edge iff* $\texttt{active}(v) < \texttt{active}(u)$.
3. *$u{-}v$ is a forward edge iff there exists some $w \in V \setminus \{u, v\}$ such that* $\texttt{active}(v) \subseteq \texttt{active}(w) \subseteq \texttt{active}(u)$.
4. *$u{-}v$ is a tree edge iff* $\texttt{active}(v) \subseteq \texttt{active}(u)$ *but it is not a forward edge.*

This above classification of edges by active ranges is illustrated in Figure 8.

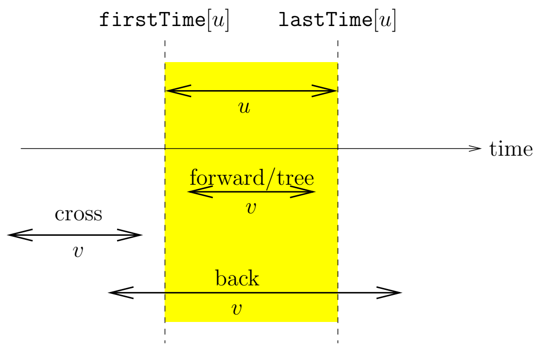


Figure 8: Relative positions of active ranges of u, v and the classification of edge $(u{-}v)$

These criteria can be used by the PREVISIT(v, u) macro to classify edges of G:

```
PREVISIT$(v, u)$
▷ Visiting $v$, from $u$
    if ($\texttt{firstTime}[v] = -1$), mark $u{-}v$ as "tree edge"
    elif ($\texttt{firstTime}[v] > \texttt{firstTime}[u]$), mark $u{-}v$ as "forward edge"
    elif ($\texttt{lastTime}[v] = \infty$), mark $u{-}v$ as "back edge"
    else mark $u{-}v$ as "cross edge".
```

To verify the correctness of this classification, we first note that tree edges are clearly correctly labeled. The remaining edges are non-tree edges. An inspection of Figure 8 will reveal that the tests that we perform are sufficient to distinguish among the forward, back and cross edges.

¶26. Application to detecting cycles. We claim that the graph is acyclic iff there are no back edges. One direction is clear — if there a back edge, we have a cycle. Conversely, if there is a cycle $Z = [u_1 - \cdots - u_k]$, then there must be a vertex (say, u_1) in Z that is first reached by the DFS algorithm. Thus there is an unseen path from u_1 to u_k, and so $\mathtt{active}(u_k) \subseteq \mathtt{active}(u_1)$. Thus there is a back edge from u_k to u_1. Hence, we can use the DFS algorithm to check if a graph is acyclic. A simple way is to run DFS starting from each vertex of the graph, looking for cycles. This takes $O(mn)$ time. A more efficient solution is given in the Exercise.

Cycle detection is a basic task in many applications. In operating systems, we have **processes** and **resources**: a process can **request** a resource, and a resource can be **acquired** by a process. We assume that that processes require exclusive use of resources: a request for a resource will be **blocked** if that resource is currently acquired by another process. Finally, a process can **release** a resource that it has acquired.

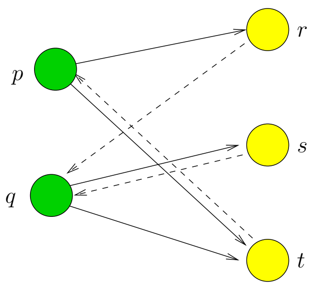


Figure 9: Process-resource Graph: $V_P = \{p, q\}, V_R = \{r, s, t\}$.

We consider a digraph $G = (V, E)$ where $V = V_P \uplus V_R$ and $E \subseteq (V_P \times V_R) \cup (V_R \times V_P)$. With this restriction on E, we call G a **bipartite graph** and write $G = (V_P, V_R, E)$ instead of $G = (V, E)$. See Figure 9 for an example with 2 processes and 3 resources. Each $p \in V_P$ represents a process and $r \in V_R$ represents a resource. An edge $(p, r) \in E$ means that p requests r but is blocked. An edge $(r, p) \in E$ means r is acquired by p. If the outdegree of p is positive, we say p is **blocked**. If the outdegree of r is positive, we say r is **acquired**. The graph satisfies three conditions:

- (1) Either (p, r) or (r, p) is not in E.
- (2) $(p, r) \in E$ implies there exist p' such that $(r, p') \in E$.
- (3) The outdegree of each r is 0 or 1.

In operating systems (Holt 1971), G is called a **process-resource graph**. It represents the current state of blocked processes and acquired resources. A cycle in G is called a **deadlock** if it contains a cycle. For instance, the graph in Figure 9 has a deadlock. In this situation, a certain subset of the processes could not make any progress. Thus our cycle detection algorithm can be used to detect this situation. In the Exercise, we elaborate on this model.

Exercises

Exercise 6.1: Suppose $G = (V, E; \lambda)$ is a strongly connected digraph in which $\lambda : E \to \mathbb{R}_{>0}$. A **potential function** of G is $\phi : V \to \mathbb{R}$ such that for all $u - v \in E$,

$$\lambda(u, v) = \phi(u) - \phi(v).$$

(a) Consider the cylic graphs C_n (see Figure 4(d)). Show that if $G = (C_n; \lambda)$ then G does not have a potential function.
(b) Generalize the observation in part (a) to give an easy-to-check property $P(G)$ of G such that G has a potential function iff property $P(G)$ holds.
(c) Give an algorithm to compute a potential function for G iff $P(G)$ holds. You must prove that your algorithm is correct. EXTRA: modify your algorithm to output a "witness" in case $P(G)$ does not hold. ♢

Exercise 6.2: Give an efficient algorithm to detect a deadlock in the process-resource graph. ♢

Exercise 6.3: Process-Resource Graphs. Let $G = (V_P, V_R, E)$ be a process-resource graph — all the following concepts are defined relative to such a graph G. We now model processes in some detail. A process $p \in V_P$ is viewed as a sequence of instructions of the form $REQUEST(r)$ and $RELEASE(r)$ for some resource r. This sequence could be finite or infinite. A process p may **execute** an instruction to transform G to another graph $G' = (V_P, V_R, E')$ as follows:

- If p is blocked (relative to G) then $G' = G$. In the following, assume p is not blocked.
- Suppose the instruction is $REQUEST(r)$. If the outdegree of r is zero or if $(r, p) \in E$, then $E' = E \cup \{(r, p)\}$; otherwise, $E' = E \cup \{(p, r)\}$.
- Suppose the instruction is $RELEASE(r)$. Then $E' = E \setminus \{(r, p)\}$.

An **execution sequence** $e = p_1p_2p_3 \ldots$ $(p_i \in V_P)$ is just a finite or infinite sequence of processes. The **computation path** of e is a sequence of process-resource graphs, $(G_0, G_1, G_2, \ldots)$, of the same length as e, defined as follows: let $G_i = (V_P \cup V_R, E_i)$ where $E_0 = \emptyset$ (empty set) and for $i \geq 1$, if p_i is the jth occurrence of the process p_i in e, then G_i is the result of p_i executing its jth instruction on G_{i-1}. If p_i has no jth instruction, we just define $G_i = G_{i-1}$. We say e (and its associated computation path) is **valid** if for each $i = 1, \ldots, m$, the process p_i is not blocked relative to G_{i-1}, and no process occurs in e more times than the number of instructions in e. A process p is **terminated** in e if p has a finite number of instructions, and p occurs in e for exactly this many times. We say that a set V_P of processes **can deadlock** if some valid computation path contains a graph G_i with deadlock.
(a) Suppose each process in V_P has a finite number of instructions. Give an algorithm to decide if V_P can deadlock. That is, does there exist a valid computation path that contains a deadlock?
(b) A process is **cyclic** if it has an infinite number of instructions and there exists an integer $n > 0$ such that the ith instruction and the $(i + n)$th instruction are identical for all $i \geq 0$. Give an algorithm to decide if V_P can deadlock where V_P consists of two cyclic processes. ♢

Exercise 6.4: We continue with the previous model of processes and resources. In this question, we refine our concept of resources. With each resource r, we have a positive integer $N(r)$ which represents the number of copies of r. So when a process requests a resource r, the process does not block unless the outdegree of r is equal to $N(r)$. Redo the previous problem in this new setting. ♢

End Exercises

§7. Further Applications of Graph Traversal

In the following, assume $G = (V, E)$ is a digraph with $V = \{1, 2, \ldots, n\}$. Let $per[1..n]$ be an integer array that represents a permutation of V in the sense that $V = \{per[1], per[2], \ldots, per[n]\}$. This array can also be interpreted in other ways (e.g., a ranking of the vertices).

¶27. Topological Sort. One motivation is the so-called[10] PERT graphs: in their simplest form, these are DAG's where vertices represent activities. An edge $u{-}v \in E$ means that activity u must be performed before activity v. By transitivity, if there is a path from u to v, then u must be performed before v. A topological sort of such a graph amounts to a feasible order of execution of all these activities.

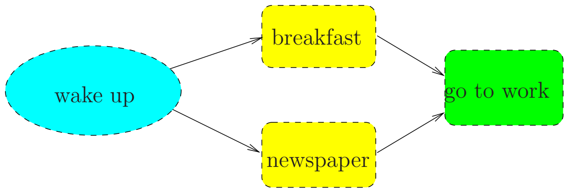


Figure 10: PERT graph

Let

$$(v_1, v_2, \ldots, v_n) \tag{9}$$

be a listing of the vertices in V. We call it a **topological sort** if every edge has the form $v_i{-}v_j$ where $i < j$. In other words, each edge points to the right, no edge points to the left. REMARK: if $(v_1, \ldots, v_n)$ is a topological sort, then $(v_n, v_{n-1}, \ldots, v_1)$ is called a **reverse topological sort**.

If an edges $u{-}v$ is intepreted as saying "activity u must precede activity v", then a topological sort give us one valid way for doing these activities (do activities $v_1, v_2, \ldots$ in this order).

Let us say that vertex v_i has **rank** i in the topological sort (9). Hence, we may represent this topological sort by a rank attribute array $Rank[1, \ldots, n]$, where $Rank[v_i] = i$ for all $v_i \in V$.

E.g., $(v_1, \ldots, v_n) = (v_3, v_1, v_2, v_4)$ in (9). The corresponding rank attribute array is $Rank[v_1, v_2, v_3, v_4] = [2, 3, 1, 4]$.

We use the DFS algorithm and the DFS Driver to compute the rank attribute array. First, we must initialize the $Rank$ array using the global initialization shell:

$$GLOBAL\_INIT(G) \equiv (\textsf{for } v = 1 \textsf{ to } n, Rank[v] \leftarrow -1).$$

Indeed, we need not use a separate color array: we simply interpret the $Rank$ of -1 as `unseen`. The idea is to use DFS(v) to assign a rank to v: but before we could assign a rank to v, we must (recursively) assign a larger rank to the vertices reachable from v. To do this, we use a global counter R that is initialized to n. Each time a vertex is to receive a rank, we use the current value of R, and then decrement R. So by the time v receives its rank, all those vertices reachable from

[10]PERT stands for "Program Evaluation and Review Technique", a project management technique that was developed for the U.S. Navy's Polaris project (a submarine-launched ballistic missile program) in the 1950's. The graphs here are also called networks. PERT is closely related to the CriticalPath Method (CPM) developed around the same time.

v would have received a larger rank. This idea can be implemented by programming the postvisit shell as follows:

$$POSTVISIT(v) \equiv (Rank[v] \leftarrow R; R \leftarrow R - 1).$$

It is easy to prove the correctness of this procedure, provided the input graph is a DAG. But what can go wrong in this code if the input is not a DAG?

REMARKS: Note that the rank function is just as the order of v according to `lastTime`$[v]$. In our strong component algorithm below, we prefer to compute the inverse of Rank, i.e., an array $Per[1..n]$ such that $Per[i] = v$ iff $Rank[v] = i$. The topological sort (9) is then equal to $(Per[1], Per[2], \ldots, Per[n])$. We leave it as an easy exercise to modify the above code to computer Per directly.

¶28. Robust Topological Sort. Suppose we want a more robust algorithm that will detect an error in case the input is not a DAG. We need the following fact: *G is cyclic iff there exists a back edge in every DFS traversal.* This was shown in the previous section. To detect back edges, when we need two modifications. The previous solution is implicitly a 2-color scheme ($Rank[v] = -1$ if v is `unseen`, and otherwise v is `seen`). Now, we need to a 3-color scheme where

$$Rank[v] \begin{cases} = -1 & \text{if } v \text{ is } \texttt{unseen}, \\ = 0 & \text{if } v \text{ is } \texttt{seen}, \\ > 0 & \text{if } v \text{ is } \texttt{done}. \end{cases}$$

To implement this, we just need to program the shell for visiting a vertex:

$$VISIT(v, u) \equiv (Rank[v] \leftarrow 0.)$$

The second modification is to check for back edges. This can be done during previsits to a vertex v from u:

$$PREVISIT(v, u) \equiv (\textsf{if } (Rank[v] = 0) \textsf{ then } ThrowException(\texttt{"Cycle detected"}))$$

¶29. Strong Components. Computing the components of digraphs is somewhat more subtle than the corresponding problem for bigraphs. In fact, at least three distinct algorithms for this problem are known. Here, we will develop the version based on "reverse graph search".

Let $G = (V, E)$ be a digraph where $V = \{1, \ldots, n\}$. For clarity, we also write "v_i" for $i \in V$. Let $Per[1..n]$ be an array that represents some permutation of the vertices, so $V = \{Per[1], Per[2], \ldots, Per[n]\}$. Let $DFS(i)$ denote the DFS algorithm starting from vertex i. Consider the following method to visit every vertex in G:

```
STRONG_COMPONENT_DRIVER($G, per$)
    INPUT: Digraph $G$ and permutation $Per[1..n]$.
    OUTPUT: A set of DFS Trees.
▷ Initialization
1.      For $i = 1, \ldots, n$, $color[i]$ =unseen.
▷ Main Loop
2.      For $i = 1, \ldots, n$,
3.          If ($color[Per[i]]$ =unseen)
4.              $DFS_1(Per[i])$    ◁ Outputs a DFS Tree
```

This program is the usual DFS Driver program, except that we use $Per[i]$ to determine the choice of the next vertex to visit, and it calls DFS_1, a variant of DFS. We assume that $DFS_1(i)$ will (1) change the color of every vertex that it visits, from `unseen` to `seen`, and (2) output the DFS tree rooted at i. If Per is correctly chosen, we want each DFS tree that is output to correspond to a strong component of G.

First, let us see how the above subroutine will perform on the digraph G_6 in Figure 5(a). Let us also assume that the permutation is

$$\begin{aligned} Per[1,2,3,4,5,6] &= [6,3,5,2,1,4] \\ &= [v_6, v_3, v_5, v_2, v_1, v_4]. \end{aligned} \tag{10}$$

The output of Strong_Component_Driver will be the DFS trees for on the following sets of vertices (in this order):

$$C_1 = \{v_6\}, \quad C_2 = \{v_3, v_2, v_5\}, \quad C_3 = \{v_1\}, \quad C_4 = \{v_4\}.$$

Since these are the four strong components of G_6, the algorithm is correct. It is not not hard to see that there always exist "good permutations" for which the output is correct. Here is the formal definition of what this means:

A permutation $Per[1..n]$ is **good** if, for any two strong components C, C' of G, if there is a path from C to C', then the *first vertex of C' is listed before the first vertex of C'.*

It is easy to see that our Strong Component Driver will give the correct output iff the given permutation is good. But how do we get good permutations? Roughly speaking, they correspond to weak forms of "reverse topological sort" of G. There are two problems: topological sorting of G is not really meaningful when G is not a DAG. Second, good permutations requires some knowledge of the strong components which is what we want to compute in the first place! Nevertheless, let us go ahead and run the topological sort algorithm (not the robust version) on G. We may assume that the algorithm returns an array $Per[1..n]$ (the inverse of the $Rank[1..n]$). The next lemma shows that $Per[1..n]$ almost has the properties we want. For any set $C \subseteq V$, we first define

$$Rank[C] = \min\{i : Per[i] \in C\} = \min\{Rank[v] : v \in C\}$$

Lemma 10. *Let C, C' be two distinct strong components of G.*
(a) If $u_0 \in C$ is the first vertex in C that is seen, then $Rank[u_0] = Rank[C]$.
(b) If there is path from C to C' in the reduced graph of G, then $Rank[C] < Rank[C']$.

Proof. (a) By the Unseen Path Lemma, every node $v \in C$ will be a descendent of u_0 in the DFS tree. Hence, $Rank[u_0] \le Rank[v]$, and the result follows since $Rank[C] = \min\{Rank[v] : v \in C\}$.
(b) Let u_0 be the first vertex in $C \cup C'$ which is seen. There are two possibilities: (1) Suppose $u_0 \in C$. By part (a), $Rank[C] = Rank[u_0]$. Since there is a path from C to C', an application of the Unseen Path Lemma says that every vertex in C' will be descendents of u_0. Let u_1 be the first vertex of C' that is seen. Since u_1 is a descendent of u_0, $Rank[u_0] < Rank[u_1]$. By part(a), $Rank[u_1] = Rank[C']$. Thus $Rank[C] < Rank[C']$. (2) Suppose $u_0 \in C'$. Since there is no path from u_0 to C, we would have assigned a rank to u_0 before any node in C is seen. Thus, $Rank[C_0] < Rank[u_0]$. But $Rank[u_0] = Rank[C']$. **Q.E.D.**

This lemma implies that, in the reverse "topological sort" ordering,

$$[Per[n], Per[n-1], \ldots, Per[1]] \tag{11}$$

if there is path from C to C', then the *last* vertex of C' in this list appears *before* the *last* vertex of C in this list. So this is not quite good.

We use another insight: consider the reverse graph G^{rev} (i.e., $u{-}v$ is an edge of G iff $v{-}u$ is an edge of G^{rev}). It is easy to see that C is a strong component of G^{rev} iff C is a strong component of G. However, there is a path from C to C' in G^{rev} iff there is a path from C' to C in G.

Lemma 11. *If $Per[1..n]$ is the result of running topological sort on G^{rev} then Per is a good permutation for G.*

Proof. Let C, C' be two components of G and there is a path from C to C' in G. Then there is a path from C' to C in the reverse graph. According to the above, the last vertex of C is listed before the last vertex of C' in (11). That means that the first vertex of C is listed after the first vertex of C' in the listing $[Per[1], Per[2], \ldots, Per[n]]$. This is good. **Q.E.D.**

We now have the complete algorithm:

```
Strong_Component_Algorithm($G$)
    Input: Digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
    Output: A list of strong components of $G$.
1.      Compute the reverse graph $G^{rev}$.
2.      Call topological sort on $G^{rev}$.
            This returns a permutation array $Per[1..n]$.
3.      Call Strong_Component_Driver($G, Per$)
```

Remarks. Tarjan [4] was the first to give a linear time algorithm for strong components. R. Kosaraju and M. Sharir independently discovered the reverse graph search method described here. The reverse graph search is conceptually elegant. But since it requires two passes over the graph input, it is slower in practice than the direct method of Tarjan. Yet a third method was discovered by Gabow in 1999. For further discussion of this problem, including history, we refer to Sedgewick [3].

Exercises

Exercise 7.1: Modify our topological sort algorithm so that it outputs the permutation array $Per[1..n]$ that is the inverse of $Rank[1..n]$. ♢

Exercise 7.2: Give an algorithm to compute the number $N[v]$ of distinct paths originating from each vertex v of a DAG. Thus $N[v] = 1$ iff v is a sink, and if $u{-}v$ is an edge, $N[u] \geq N[v]$. ♢

Exercise 7.3: Let G be a DAG.
(a) Prove that G has a topological ranking.
(b) If G has n vertices, then G has at most $n!$ topological rankings.
(c) Let G consists of 3 disjoint linear lists of vertices with n_1, n_2, n_3 vertices (resp.). How many topological rankings of G are there? ♢

Exercise 7.4: Prove that a digraph G is cylic iff every DFS search of G has a back edge. ♢

Exercise 7.5: Consider the following alternative algorithm for computing strong components of a digraph G: what we are trying to do in this code is to avoid computing the reverse of G.

```
STRONG_COMPONENT_ALGORITHM($G$)
    INPUT: Digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
    OUTPUT: A list of strong components of $G$.
1.      Call topological sort on $G$.
            This returns a permutation array $Per[1..n]$.
2.      Reverse the permutation:
            for $i = 1, \ldots, \lfloor n/2 \rfloor$, do the swap $Per[i] \leftrightarrow Per[n+1-i]$.
3.      Call STRONG_COMPONENT_DRIVER($G, Per$)
```

Either prove that this algorithm is correct or give a counter example. ♢

Exercise 7.6: An edge $u{-}v$ is **inessential** if there exists a $w \in V \setminus \{u, v\}$ such that there is a path from u to w and a path from w to v. Otherwise, we say the edge is **essential**. Give an algorithm to compute the essential edges of a DAG. ♢

Exercise 7.7: Let G_0 be a DAG with m edges. We want to construct a sequence $G_1, G_2, \ldots, G_m$ of DAG's such that each G_i is obtained from G_{i-1} by reversing a single edge so that finally G_m is the reverse of G_0. Give an $O(m+n)$ time algorithm to compute an ordering $(e_1, \ldots, e_m)$ of the edges corresponding to this sequence of DAGs.

NOTE: this problem arises in a tie breaking scheme. Let M be a triangulated mesh that represents a terrain. Each vertex v of M has a height $h(v) \geq 0$, and each pair u, v of adjacent vertices of M gives rise to a directed edge $u{-}v$ if $h(u) > h(v)$. Note that if the heights are all distinct, the resulting graph is a DAG. If $h(u) = h(v)$, we can arbitrarily pick one direction for the edge, as long as the graph remain a DAG. This is the DAG G_0 in our problem above. Suppose now we have two height functions h_0 and h_1, and we want to interpolate them: for each $t \in [0, 1]$, let $h_t(v) = th_0(v) + (1-t)h_1(v)$. We want to represent the transformation from h_0 to h_1 by a sequence of graphs, where each successive graph is obtained by changing the direction of one edge. ♢

Exercise 7.8: Let $D[u]$ denote the number of descendents a DAG $G = (V, E)$. Note that $D[u] = 1$ iff u is a sink. Show how to compute $D[u]$ for all $u \in V$ by programming the shell macros. What is the complexity of your algorithm? ♢

Exercise 7.9: A vertex u is called a **bottleneck** if for every other vertex $v \in V$, either there is a path from v to u, or there is a path from u to v. Give an algorithm to determine if a DAG has a bottleneck. HINT: You should be able to do this in at most $O(n(m+n))$ time. ♢

Exercise 7.10: In the previous problem, we defined bottlenecks. Now we want to classify these bottlenecks into "real" and "apparent" bottlenecks. A bottleneck u is "apparent" if there exists an ancestor v ($\neq u$) and a descendent w ($\neq u$) such that $v{-}w$ is an edge. Such an edge $v{-}w$ is called a by-pass for u. Give an efficient algorithm to detect all real bottlenecks of a DAG G. HINT: This can be done in $O(n + m \log n)$ time. ♢

Exercise 7.11: Given a DAG G, let $D[u]$ denote the number of descendents of u. Can we compute $D[u]$ for all $u \in V$ in $o((m+n)n)$ time, i.e., faster than the obvious solution? ♢

End Exercises

References

[1] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North Holland, New York, 1976.

[2] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, second edition, 2001.

[3] R. Sedgewick. *Algorithms in C: Part 5, Graph Algorithms*. Addison-Wesley, Boston, MA, 3rd edition edition, 2002.

[4] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2), 1972.