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

Lecture IV
Pure Graph Problems

Graph Theory is said to have originated with Euler (1707–1783) who was asked to resolve an pastime question of the citizens of Königsberg whether it is possible to traverse all the 7 bridges joining two islands and the mainland, without retracing any path. Euler recognized in this problem the essense of Leibnitz's interest in founding a new kind of mathematics ("analysis situs") which might be interpreted as topological or combinatorial analysis in modern language.

A graph is fundamentally a set of mathematical relations (called incidence relations) connecting two sets, a vertex set V and an edge set E. 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[1] 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 $u = v$. We say the vertices u and v are incident on $u{-}v$.

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

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.

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. The algorithmic issues of pure

[1] We have taken this highly suggestive notation from [2].

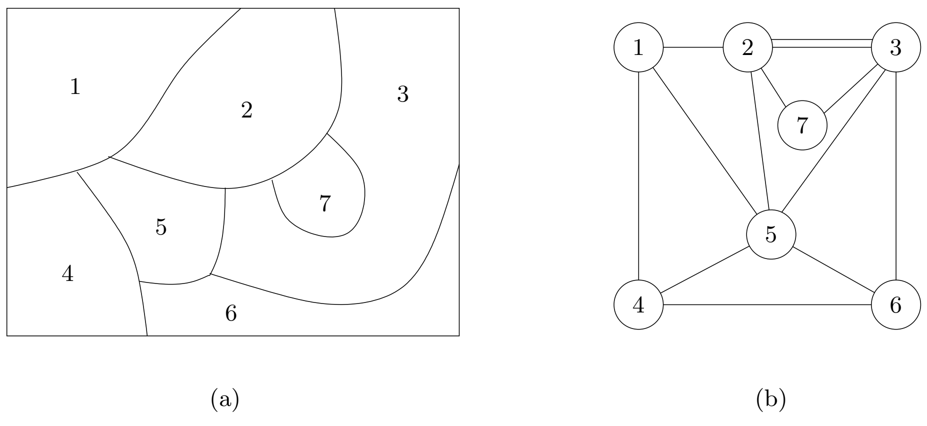


Figure 1: (a) Political map of 7 countries (b) Adjacency relationship of countries

graphs mostly 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). Two other important, if less elementary, problems of pure graphs are: testing if two graphs are isomorphic, and testing if a graph is planar. We will treat planarity testing. Tarjan [3] was one of the first to systematically study the DFS algorithm and its applications. A lucid account of basic graph algorithms may be found Sedgewick [2].

§1. Bigraphs and Digraphs

Basic graph definitions are given. In this book, "graphs" refer to either directed graphs ("digraphs") or undirected graphs ("bigraphs"). All graphs are assumed to be simple.

Set-Theoretic Notations for Simple Graphs. A graph G is basically given by two sets, V and E. These are called the vertex set and edge set, respectively. We begin by describing "simple graphs" in the three most important cases. The terminology "simple" will become clear later.

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

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

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 $(V^k, 2^V)$ 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:

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

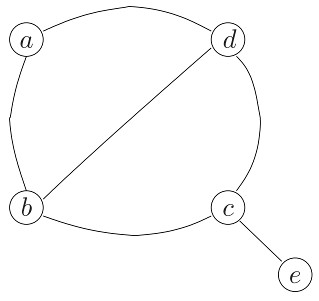
We can define our 3 varieties of simple graphs as follows:

- 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 $E \subseteq V^2$.

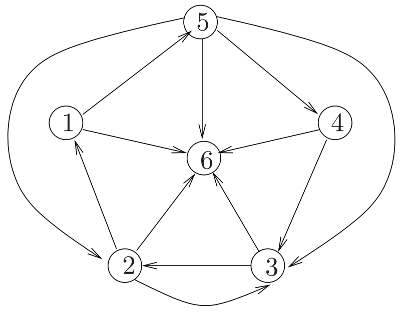
- A **undirected graph** (or simply, **bigraph**) is a pair $G = (V, E)$ where $E \subseteq \binom{V}{2}$.

In all three cases, the elements of V are called **vertices**. Elements of E may be called **edges**, but we shall reserve this term only for digraphs or bigraphs; for hypergraphs, we call them **hyperedges**. Hence, each edge is defined by a pair $u, v \in V$, and we use the notation $u{-}v$ to represent the corresponding edge of the digraph or bigraph. This convention is useful because many of our definitions cover both digraphs and bigraphs. Similarly, the term "graph" will cover digraphs and bigraphs. Some basic graph terminology is collected in §I (Appendix A). Hypergraphs are sometimes called **set systems** (see matroid theory in Chapter 5).

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 representation: elements of V are represented by points (or circles) in the plane and elements of E are represented by finite curve segments connecting these points.



(a) bigraph



(b) digraph

Figure 2: Bigraph and digraph.

In Figure 2(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 2(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. Thus, in Figure 2(b), all the edges incident on vertex 6 has 6 as the stop vertex and so the arrow heads are all pointed at 6. Thus edges are "directed" from the start to the stop vertex. In contrast, the curve segments in bigraphs are undirected (bi-directional).

Auxiliary Data Convention. We want to associate some additional data with a graph. For instance, we may want to designate two vertices $s, t \in V$ as the source and destination vertices. In this case we may write $G = (V, E; s, t)$. In general, auxiliary data will be separated from the pure graph data by a semi-colon, $G = (V, E; \cdots)$.

Exercises

Exercise 1.1: Prove or disprove: there exists a bigraph $G = (V, E)$ where $|V|$ is odd and the degree of each vertex is odd. ♢

Exercise 1.2:
(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. $\diamondsuit$

Exercise 1.3: 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 $n = 4, 5, 6$. $\diamondsuit$

End Exercises

§2. Path Concepts

Most basic concepts in 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. Note that adjacency is an asymmetric relation for digraphs but symmetric for bigraphs. A typical usage is this: "for each v adjacent to u, do $\ldots v \ldots$".

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 **simple** if all its vertices, with the possible exception of $v_0 = v_k$, are distinct. Note that a trivial path is always 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.

The Path 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. It is easy to see that

- $\delta(u, v) \geq 0$, with equality iff $u = v$.
- (Triangular Inequality) $\delta(u, v) \leq \delta(u, w) + \delta(w, v)$.
- When G is a bigraph, then $\delta(u, v) = \delta(v, u)$.

These three properties amounts 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**.**

Subpaths. Let path 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 the new path. 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 "$p; q$", etc, we will assume that the operation is well-defined.

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. Clearly p must be a closed path because the source of r and the target of r' must be the same in order for $r'; r$ to be well-defined, but this means that 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. 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:

$$(1–2–3–4–1) = (1–2–3); (3–4–1), \qquad (3–4–1–2–3) = (3–4–1); (1–2–3).$$

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 particular, a trivial path $p = (v_0)$ gives rise to the cycle which is an empty sequence $Z = [\,]$. We call this the **trivial cycle**. 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 $[u]$.

Path concepts that are invariant under cyclic equivalence are "transferred" to cycles automatically: for instance, we may speak of the **length** or **reverse** of a cycle, etc. A cycle $[v_1 – \cdots – v_k]$ is **simple** if the vertices $v_1, \ldots, v_k$ are distinct. If we excise a finite number of closed subpaths from a closed path p, we obtain a closed subpath q; call $[q]$ a **subcycle** of $[p]$. For instance, $[1–2–3]$ is a subcycle of

$$[1–2–a–b–c–2–3–d–e–3].$$

From the general transfer principle, we say a cycle $Z = [p]$ is **trivial** iff p is a trivial path. We next wish to 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. For instance, for any edge $u – v$ in a bigraph, we get the closed path $(u – v – u)$ and hence the non-trivial cycle $[u – v]$. Thus our definition of "cyclic graphs" is unsuitable for bigraphs.

Thus, for bigraphs, we proceed as follows: first, define a closed path $p = (v_0 – v_1 – \cdots – v_k)$ to be **reducible** if one of the following two conditions hold:

- $v_{i-1} = v_{i+1}$ for some $i = 1, \ldots, k-1$,
- $k \geq 2$ and $v_1 = v_{k-1}$.

Otherwise p is said to be **irreducible**. If a closed path is reducible then it is a non-simple path. A cycle $Z = [p]$ is reducible iff any of its representative is reducible. So the trivial cycle and self-loop cycle $[u]$ is irreducible. Finally, a bigraph is said to be **cyclic** if it contains some irreducible non-trivial cycle. Note that irreducible non-trivial cycles have length at least 3.

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, and for emphasis, we may add the qualifier "strong" when discussing components of digraphs.

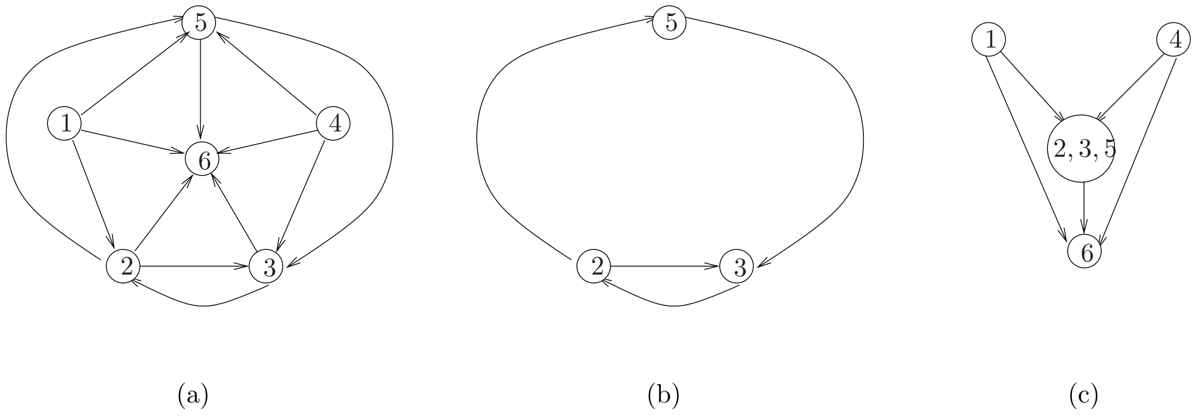


Figure 3: (a) Digraph G_6, (b) Component graph of $C = \{2, 3, 5\}$, (c) Reduced graph G_6^c

For example, the graph G_6 in Figure 3(a) has $C_2 = \{2, 3, 5\}$ as a component. The component graph corresponding to C is shown in Figure 3(b). The other components of G are $\{1\}, \{4\}, \{6\}$, all trivial.

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 3(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.

Note that the reduced graph is essentially trivial for bigraphs, so this concept is only applied to digraphs. But for bigraphs, we will later introduce a stronger notion of connectivity, called bi-connectivity.

DAGs and Trees. We have just defined cyclic bigraphs and digraphs. A graph is **acyclic** if it is not cyclic. Acyclic graphs is a very important subclass of graphs. The common acronym for a directed acyclic graph is **DAG**. A **tree** is a DAG in which there is a unique vertex u_0 called the **root** such that there exists a unique path from u_0 to any other vertex. Trees are ubiquitous in computer science. Thus, we have free trees, rooted trees, ordered trees, search trees, etc.

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. Can u be adjacent to itself if G is a bigraph? If G is a digraph? $\diamondsuit$

Exercise 2.2: Describe an efficient 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). What is the complexity of your algorithm? Make explicit any assumptions you need about representation of paths and vertices. $\diamondsuit$

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: a linked list of the vertices of G and a list edges of G. The lists may be singly- or doubly-linked. E.g., the edge list representations of the two graphs in Figure 2 would be

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

  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 2 are

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

  and

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

- 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 2 are

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

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

Size Parameter. Two size parameters are used in measuring the computational complexity of graph problems: $|V|$ and $|E|$. These are typically denoted by n and m. It is clear that m is not completely independent, but satisfies the bounds $0 \le m \le n^2$. 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**. For example, the family $\mathcal{G}$ of planar graphs is sparse because $m = O(n)$ in planar graphs. Some computational techniques can exploit sparsity of input graphs.

Thus, the first two method of representing graphs use $O(m+n)$ space while the last method uses $O(n^2)$ space. Thus the adjacency matrix representation is generally not used for sparse graphs.

Arrays. 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. For instance, this allows us to iterate over all the vertices using an integer variable. To associate an attribute A with each vertex, 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 is weight, then $A[i]$ is the weight of vertex i.

Coloring Scheme. In many graph algorithms we need to keep track of some "processing status" of the vertices. Initially, the vertices are unprocessed, and finally they are processed. Sometimes, we want to indicate an intermediate status of being partially processed. 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.). Initially, all vertices are unseen or white. The color transitions of each vertex are always in this order:

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

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.

§4. Breadth First Search

A **graph traversal** is any systematic method to "visit" each vertex and edge of a graph. Here is the generic graph traversal algorithm: assume that all vertices are initially colored as `unseen`. *Start from any vertex s_0, mark it as* `seen`. *Add all the edges out of s_0 to a set Q. While Q is not empty, remove an edge (u, v) from Q. If v is unseen, color it as* `seen`, *and add add edges out of v to Q.*

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 will follow this tradition. In both algorithms, we assume that 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 2(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 4(a). It is called the "BFS tree".

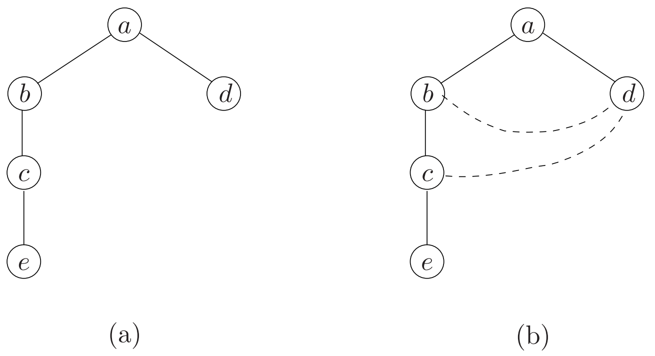


Figure 4: (a) BFS tree. (b) Non-tree edges.

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

$$\delta(s_0, u) < \delta(s_0, v) < \infty. \qquad (2)$$

If $\delta(s_0, u) = \infty$, then u will not be visited from s_0. The BFS algorithm does not explicitly compute the relation (2) 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 (1)). We could use three colors, but for our current purposes, two suffice: `white`/`gray` or `unseen`/`seen`.

We formulate our BFS algorithm as a "skeleton" for accomplishing application-specific functions:

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

This algorithm is a skeleton because we have embedded in it several subroutines

$$\text{INIT, PREVISIT, VISIT and POSTVISIT} \tag{3}$$

that are application-specific; these subroutines will be assumed to be[2] null operations unless otherwise specified. We refer to the subroutines in (3) as **shell subroutines**. An application of BFS will amount to filling these shell subroutines with actual code. We can usually omit the PREVISIT step, but see §6 for an example of using this subroutine.

Note that VISIT(v, u) represents visiting v **from** u; a similar interpretation holds for PREVISIT(v, u). 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`.

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 4(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 (2) holds. E.g., let G be the bigraph in Figure 2(a) and s_0 is vertex a. Then two possible BFS listing at a are

$$(a, b, d, c, e) \qquad \text{and} \qquad (a, d, b, c, e). \tag{4}$$

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

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

- Suppose you wish to print a BFS listing of the vertices reachable from s_0. Then POSTVISIT(u) simply prints the name of u. Other subroutines 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 initial a set T to be empty. In VISIT(v, u), we add the edge u–v to T.

[2]Alternatively, we could fold the coloring steps into these subroutines, so that they are non-null. We choose to expose these coloring steps in BFS.

- 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 $d[i] \geq 0$.

BFS Analysis. We shall analyze the behavior of the BFS algorithm on a bigraph. A basic property that is implicit in the following discussion is that the BFS algorithm terminates – this is left as an Exercise. For instance, termination 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 first prove a simple lemma:

Lemma 1 (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:*
$$\delta(u_1) \leq \delta(u_2) \leq \cdots \leq \delta(u_k) \leq 1 + \delta(u_1). \tag{5}$$

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 (5) holds. In Line 2, 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 8, 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 has this property. This proves the invariant (5). **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 2.
(i) For any edge $u-v$, $|\delta(u) - \delta(v)| \leq 1$.
(ii) For each vertex u in the BFS Tree,
$$\delta(u) = \delta(s_0, u),$$
i.e., $\delta(u)$ is the length of the shortest path from s_0 to u.

Proof. (i) Consider the edge $u-v$. Without loss of generality, assume that u was seen before v. If there is an instant in time when u and v both appear on the queue, then we are done, since $\delta(u) - \delta(v) \leq 1$ from the monotone 0-1 property. Assume otherwise. Now there instant t_1 when u reached the head of the queue. Then v is still unseen. But as we process the vertices adjacent to u, the vertex v will be seen from u and in this case $\delta(u) - \delta(v) = 1$.

(ii) 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 $i \geq 1$, part(i) 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 3. *If $\delta(u) < \delta(v)$ then u is VISITed before v is VISITed, and u is POSTVISITed before v is POSTVISITed.*

The edges of the graph G can be classified into the following types by the BFS Algorithm (cf. Figure 4(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 4(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 4(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 4(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 2(i). Hence we have:

Theorem 4. *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 subroutines to produce the above classification of edges.

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        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 subroutine called GLOBAL_INIT to our collection (3).

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 subroutines; but as long as these subroutines are $O(1)$ time our complexity analysis will remain 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 $m' \le m$ 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 subroutine are pairwise disjoint. Hence the Driver program takes time $\Theta(n + m)$.

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) subroutine simply increments the `count` variable. Finally, the VISIT(v, u) subroutine simply assigns $C[v] \leftarrow$ `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.

Application: Testing Bipartiteness. A bigraph $G = (V, E)$ is bipartite if V can be partititioned into $V = V_1 \uplus V_2$ such that $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$. It is clear that all cycles in a bipartite graphs must be **even** (i.e., has an even number of edges). In an exercise, we ask you to show the converse: 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. It is not immediate that if there is a cross-edge (u, v), then we have found an odd cycle. In the exercise, we ask you to show that all odd cycles is represented by such a cross-edge. It is now a simple matter to modify BFS to detect cross-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 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 cross-edge. This exception is caught by the BFS Driver program.

Exercises

Exercise 4.1: Prove that every vertex that is reachable from the source will be seen by BFS. ♢

Exercise 4.2: Prove that the BFS algorithm terminates. ♢

Exercise 4.3: Show that each node is VISITed and POSTVISITed at most once. Is this true for PREVISIT as well? ♢

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

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

Exercise 4.6: 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 4.7: Why does the above algorithm for computing the connected component of a bigraph fail when we apply it to a digraph? ♢

Exercise 4.8: Referring to the Bipartite Test in the text, prove the following assertions:
(a) If a bigraph has no odd cycles, then it is bipartite.
(b) If a connect graph has an odd cycle, then BFS search from any source vertex will detect a cross-edge.
(c) Write the pseudo code for bipartite test algorithm outlined in the text. This algorithm is to return YES or NO only.
(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. Moreover, in the case of NO, it returns an odd cycle. ♢

Exercise 4.9: (a) Let $G = (V, E)$ be a connected bigraph. For any vertex $v \in V$ define

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

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 $u, v \in V$. Prove that $2 \cdot \mathrm{radius}(G) \geq \mathrm{diameter}(G)$.
(b) Show that for every natural number n, there are graphs G_n and H_n such that $n = \mathrm{radius}(G_n) = \mathrm{diameter}(G_n)$ and $n = \mathrm{radius}(H_n) = \lceil \mathrm{diameter}(H_n)/2 \rceil$.
(c) Using DFS, give an efficient algorithm to compute the diameter of a undirected tree (i.e., connected acyclic undirected graph). Please use the "shell" subroutines for DFS. Be sure to prove the correctness of your algorithm. What is the complexity of your algorithm?
(d) Same as (c) but now using BFS. ♢

Exercise 4.10: Conjecture why the BFS Algorithm is little-used in the processing of digraphs. ♢

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 2(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; for our present example, we obtain the tree shown in Figure 5(a).

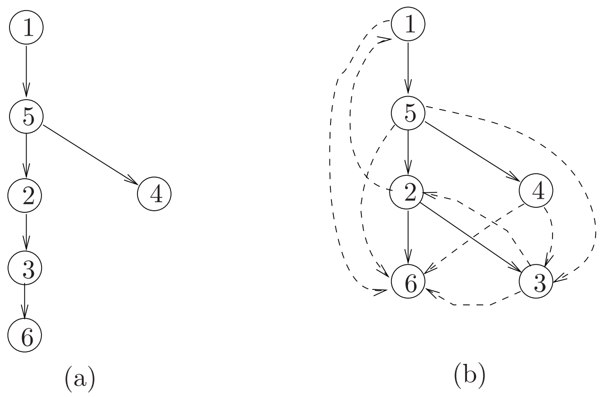


Figure 5: DFS Tree.

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

```
Recursive 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           PREVISIT($v, s_0$)
4           if $v$ is unseen then
5               VISIT($v, s_0$).
6               Simple DFS(($V, E; v$)) ◁ Recursive call
7       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). The first call to this recursive algorithm must be made by some DFS Driver Program which must do the necessary setup, including initializing the vertex colors:

```
DFS Driver
      Input:    $G = (V, E)$ a graph (bi- or dip)
      Output:   Application-specific
1           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                       VISIT($v$, nil)
6                       Simple DFS(($V, E; v$)) ◃ recursive call
```

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 subroutines,

$$\text{INIT, PREVISIT, VISIT POSTVISIT, GLOBAL\_INIT.} \tag{6}$$

These shell subroutines will be null operations unless otherwise specified.

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 5(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 the following basic result.

Lemma 5 (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*[3] *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 remark that the inductive hypothesis is a little subtle (see Exercise for another proof, and also a wrong approach). The subtlety is that the DFS algorithm has its own order for visiting vertices adjacent to each u, and your induction must account for this order.

First, we define 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 $k, \ell \geq 0$) are two distinct paths from u to v, we say $p <_{\texttt{dfs}} q$ if $u_1 = v_1, \ldots, u_m = v_m$ and $u_{m+1} < v_{m+1}$ relative to u_m. Note that m is well-defined (in particular, $m < \min\{k, \ell\}$). 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

$$\pi : (u{-}u_1{-}\cdots{-}u_k{-}v) \tag{7}$$

[3]If we use the white-black color terminology, this would be called a "white path" as in [1].

where each node $u_1, \ldots, u_k, v$ is unseen at time when we frist 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 (7) 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 $(uv-)$. 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 $(u{-}v'); \pi' <_{\texttt{dfs}} (u{-}v)$). Hence, when we recursively visit v_1, we will never color v as `seen`. Hence, we will eventually color v as `seen` from u, *i.e.*, $u{-}v$ is an edge of the DFS tree.

INDUCTIVE CASE: Suppose $k > 0$. 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. We can also verify that pi' is 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 $\pi = (u{-}u_1); \pi'$ is in the DFS tree. **Q.E.D.**

Classification of edges. First consider a digraph G. Upon calling $DFS(G, s_0)$, the edges of G becomes as follows (see Figure 5(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 5(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 5(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 5(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.

What if G is a bigraph? We adopt the convention that an undirected edge $\{u, v\}$ will regarded as the directed edge $u{-}v$ if we u is seen before v. If u, v remain unseen, then $\{u, v\}$ will remain undirected. Then the above classification of edges can now be applied to G, but the classifications 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 between 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.

Biconnectivity of Digraphs. A bigraph G is **biconnected** if for every pair u, v of distinct vertices there is a closed path that passes through u and v. Equivalently, there are two vertex-disjoint paths π, π' from u to v; vertex-disjoint means the set of vertices in π and in π' are pairwise disjoint, except for their obvious intersection in u and v. This is clearly a strong notion of connectivity.

A vertex v of G is called a **cut-vertex** if the removal of v and all the edges incident on v will disconnect the resulting subgraph. Clearly, if G has a cut-vertex, then it is not biconnected. The converse is almost true: if G has more than 2 vertices and is biconnected, then it has no cut-vertices.

Assume G is connected, and T is a DFS tree of G. There are two ways in ways a vertex u is a cut-vertex:
(i) If u is the root of T has more than one child, then u is a cut-vertex.
(ii) If u is not the root, but for every descendent v of u, if $v{-}w$ is an edge, then w is also a descendent of u.

We leave as an exercise to show that there are no other cut-vertices except those listed under (i) and (ii).

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.

It is now a relatively exercise to program the shell subroutines of the DFS algorithm to detect cut-vertices, and hence recognize biconnectivity.

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

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

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 (bi- or di-).
      Output:  Application specific
      ▷ Initialization:
0           INIT$(G, s_0)$ ◁ Make all vertices unseen except for $s_0$
1           Initialize the stack $S$ to contain $s_0$.
      ▷ Main Loop:
            while $S \neq \emptyset$ do
2                 $u \leftarrow S$.pop().
3                 for each $v$ adjacent to $u$ do
4                       PREVISIT$(v, u)$
5                       if $v$ is unseen then
6                             color $v$ seen
7                             VISIT$(v, u)$
8                             $S$.push$(v)$.
9                 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.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 (bi- or di-).
        Output:  Application specific
        ▷ Initialization:
0           INIT($G, s_0$); ◁ Make all vertices unseen except for $s_0$
1           Initialize the stack $S$ to contain $s_0$.
2           $curr \leftarrow first(s_0)$;
        ▷ Main Loop:
            while $S \neq \emptyset$ do
3               if ($curr$ = nil)
4                   $curr \leftarrow S$.pop()
5                   POSTVISIT($curr$)
6                   $curr \leftarrow next(S.\mathtt{top}(), curr)$ ◁ may be nil
7               else
8                   PREVISIT($curr$)
9                   if $curr$ is unseen
10                      color $curr$ seen
11                      VISIT($curr, S$.top())
12                      $S$.push($curr$)
13                      $curr \leftarrow first(curr)$
```

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: 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                   VISIT($v, s_0$).
4                   Simple DFS($(V, E; v)$)
5           POSTVISIT($s_0$).
6           Color $s_0$ as seen.
```

◊

Exercise 5.3: Give an alternative proof of the Unseen Path Lemma, by contradiction. Try to do this proof without explicitly invoking the ordering properties of $<_{\texttt{dfs}}$. ◊

Exercise 5.4: Here is an attempt to prove the Unseen Path Lemma (Lemma 5). Please point out the error in the following proof: "We use induction on the length ℓ of any unseen path from u to v. Base Case: If $\ell = 1$, then the DFS algorithm is sure to visit v from u, and so u–v is an edge in the DFS tree. Inductive Case: If $\ell > 1$, let v' be the parent of v in the unseen path from u to v. Then there is an unseen path from u to v' of length $\ell - 1$. By Induction Hypothesis, v' will be a descendent of u in the DFS Tree. Hence, when we visit v', we will also visit v from v'." ◊

Exercise 5.5: Prove that our classification of edges is complete. ◊

Exercise 5.6: Suppose T is the DFS Tree for a connected bigraph G. Let u–v be a non-tree edge of G. Prove that either u is an ancestor of v or vice-versa in the tree T. ◊

Exercise 5.7: Suppose G is a bigraph. Show that a DFS tree for G does not have any forward or cross edges. Give a complete classification of the edges produced by the DFS algorithm on bigraphs. HINT: Recall our convention about how edges in a bigraph are turned into ordered edges by the DFS algorithm. See the previous Exercise. ◊

Exercise 5.8: Prove that our nonrecursive DFS algorithm is equivalent to the recursive version. ◊

Exercise 5.9: When might we prefer the BDFS Algorithm in place of the standard DFS or BFS algorithms? ◊

End Exercises

§6. Full Depth First Search

To perform certain computations using 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 as `unseen`/`seen`/`done`(or `white`/`gray`/`black`). The POSTVISIT(u) subroutine can be used to color the vertex u as `done`. The `seen` vertices are precisely those are currently in the recursion stack.

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 $v \in V$. 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 use the Recursive DFS and the DFS Driver algorithms by insert the following lines of code into their shell subroutines:

- GLOBAL_INIT(G): `clock` $\leftarrow 0$.
- VISIT(v, u): `firstTime`$[v] \leftarrow$ `clock`++.
- POSTVISIT(v): `lastTime`$[v] \leftarrow$ `clock`++.

The tricolor scheme is subsumed by the timestamp scheme, since the time when a node v has color `gray` corresponds to the clock value lying between `firstTime`$[v]$ and `lastTime`$[v]$.

In some applications, we may only need one of these two 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 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 6. *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 `firstTime`$[u] <$ `firstTime`$[v]$. Moreover, since the vertex u will remain active until v is discovered, we also have `lastTime`$[v] <$ `lastTime`$[u]$. Hence `active`$(v) \subseteq$ `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 7. *If $u{-}v$ is an edge then*

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

This criteria can be programmed into our shell subroutines to correctly classify each edge of G. In fact, we only need to modify two shell programs: Initially, we use INIT(G, s_0) to mark every edge as "unseen". We also initialize `lastTime`$[v]$ to ∞ for all $v \in V$. In PREVISIT(v, u), we perform the following tests:

```
PREVISIT($v, u$)
    If $v$ is unseen, mark $u{-}v$ as "tree edge";
    else if (firstTime[$v$] > firstTime[$u$]), mark $u{-}v$ as "forward edge";
    else if (lastTime[$v$] = $\infty$), mark $u{-}v$ as "back edge";
    else mark $u{-}v$ as "cross edge".
```

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.

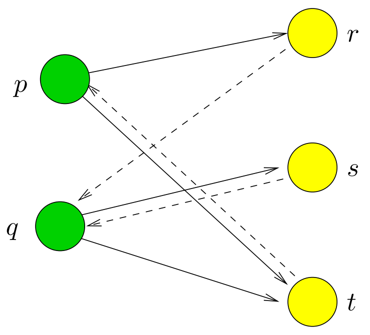


Figure 6: Process-resource Graph: $V_P = \{p, q\}, V_R = \{r, s, t\}$.

We consider a digraph $G = (V, E)$ where $V = V_P \cup V_R$ 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 6 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.

Call G 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 6 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: Fill in the shell subroutines of the DFS Algorithm so that it correctly classifies every edge of an input digraph. You must briefly argue why your classification is correct. ◊

Exercise 6.2: Suppose $G = (V, E; \lambda)$ is a strongly connected digraph in which $\lambda : E \to \mathbb{R}$.
(a) A **potential function** of G is $\phi : V \to \mathbb{R}$ such that for all $u{-}v \in E$,

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

Assuming G has a potential function, give an an algorithm to find one.
(b) Let C be a subgraph of G. Describe an easy-to-check property P of C such that G does not have a potential function iff C has property P. We may call any C with property P a "witness" for the non-existence of a potential function.
(c) Modify your solution to (a) so that for any G, it either finds a potential function or produces a "witness" C. ♢

Exercise 6.3: (a) Show that you can detect cycles in a digraph in $O(m)$ time.
(b) When applied to the process-resource graph, can you exploit its special properties to achieve a faster algorithm? ♢

Exercise 6.4: 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 6.5: 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).

Topological Sort. One motivation is the so-called[4] 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.

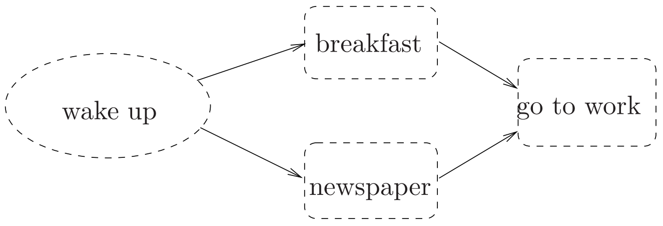


Figure 7: PERT graph

Suppose G is a DAG with $|V| = n$. An injective function $p : \{1, 2, \ldots, n\} \to V$ is called a **topological ranking** (or topological sorting) of G if for all $(u{-}v) \in E$, we have $p^{-1}(u) < p^{-1}(v)$. If $V = \{1, \ldots, n\}$ then the topological ranking p can be represented by an array $per[1..n]$ such that if for all $u \in V$, $pre[u] = p(u)$. Thus we have:

$$(per[i]{-}per[j]) \in E \text{ implies } i < j. \tag{8}$$

Property (8) ensures that if we perform activities in the order

$$per[1], per[2], \ldots, per[n],$$

then there is no "direct" inversion of priority. Thus, we interpret $per[i]$ *to be the name of the ith activity to be performed.* In Figure 7, a possible topological ranking is

$$per[1] = \text{wake up}, per[2] = \text{take breakfast}, per[3] = \text{read newspaper}, per[4] = \text{go to work}.$$

The only other topological ranking is where we take breakfast after reading newspaper.

We say "direct" in referring to priority because the precondition of (8) is information that is directly represented by the edges of G. But we could have "indirect" priorities derived by transitivity (e.g., waking up before going to work is indirectly represented in Figure 7. If we satisfy all the direct priorities, could there be inversion of indirect priorities? The answer is no. In proof, suppose that $i < j$ and $per[i]$ depends on $per[j]$. This means there is a path in the graph G from $per[j]$ to $per[i]$. Let this path be

$$(per[j_0]{-}per[j_1]{-}\cdots{-}per[j_k])$$

where $j_0 = j$ and $j_k = i$. By (8), we know that $j = j_0 < j_1 < \cdots < j_k = i$. This is a contradiction.

Here is the algorithm to compute such a topological ranking of a DAG: we use the DFS Driver Program, which in turn calls DFS. We just have to fill in the subroutines in these two shell programs.

The global initialization of G will color all vertices as unseen, and set a counter variable $count$ to $n = |V|$. The POSTVISIT(u) is simply

$$per[count] = u; \quad count{-}{-};$$

[4]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.

Now, let us prove the correctness of this algorithm. Suppose $(per[i], per[j]) \in E$. We need to show $i < j$. Let $per[i] = u$ and $per[j] = v$. When node u is assigned to $per[i]$, we know that any vertex that is reachable from u has already been been assigned. In particular, v has been assigned. Since v is assigned to $p[j]$, it means that the *count* variable was equal to i and j (resp.) when u and v were assigned. Since $v = per[j]$ was assigned earlier than $u = per[i]$, and since the *count* variable is decreasing, it means $i < j$. This is what we needed to prove.

Strong Components. Computing the components of digraphs is somewhat more subtle than the corresponding problem for bigraphs. In fact, three versions of such an algorithm are known. Here, we will develop a simple yet subtle algorithm based on what we might call "reverse graph search".

Let $G = (V, E)$ be a digraph where $V = \{1, \ldots, n\}$. 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_SUBROUTINE($G, per$)
    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 loop is a standard driver program, except that we use $per[i]$ to determine the choice of the next vertex to visit. 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.

First, let us see how the above subroutine will perform on the digraph G_6 in Figure 3(a). Let us also assume that the permutation is

$$per[1, 2, 3, 4, 5, 6] = (6, 3, 5, 2, 1, 4) \tag{9}$$

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

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

Since these are the four strong components of G_6, the algorithm is correct. We now prove that, with a suitable permutation, this is always the case:

LEMMA 8. *There exists a permutation $per[1..n]$ such that the* STRONG_COMPONENT_SUBROUTINE *is correct, that is, each each DFS Tree that is output in Step 4 corresponds to a strong component of G.*

Proof. Consider the reduced graph G^c of G. Consider a permutation $per[1..n]$ that is a **reverse topological sort** of the vertices of G. More precisely, if $per[i] = u$, we think of i as the ranking of vertex u in our reverse topological sort, and write $rank[u] = i$. So $rank[1..n]$ is just the inverse of $per[1..n]$. In other words,

$$rank[u] = i \qquad \Longleftrightarrow \qquad per[i] = u.$$

Suppose C_1, C_2 are two components of G and (C_1, C_2) is an edge in G^c, then for each vertex $u_1 \in C_1$ and $u_2 \in C_2$, we require the property

$$rank[u_1] > rank[u_2]. \tag{10}$$

With this property, we see that in our Main Loop (line 2) of the above subroutine, we will consider vertex u_2 before vertex u_1.

We must show this actually works, that is, if the algorithm calls $DFS_1(u_2)$ in line 4 within the Main Loop, it will output precisely C_2. We will use induction based on the partial order induced by the rank function. In other words, for all u_0 whose rank is less than $rank[u_2]$, a call to $DFS_1(u_0)$ produces the component of u_0.

This is certainly true in the base case (i.e., when C_2 is a sink in the DAG G^c). Inductively, assume that all previous calls to DFS_1 has correctly output only strong components. This implies that no vertices of C_2 has been output when we first call $DFS_1(u_2)$. Then, it is clear that $DFS_1(u_2)$ will reach and output every vertex in C_2.

We must next show that it is impossible to output vertices that are NOT in C_2. Suppose $DFS_1(u_2)$ reaches some unseen vertex u_0 that belongs to another component C_0. We may assume that u_0 is the first such vertex, and hence (C_2, C_0) is an edge of G^c. By assumption (10), $rank[u_0] < rank[u_2]$. This is a contradiction because in our main loop, we would have considered the vertex u_0 before u_2. This means that $color[u_0] =$ `seen` by the time we consider u_2. **Q.E.D.**

How do we compute $per[1..n]$ satisfying (10) in the preceding proof? We said that $per[1..n]$ is a topological sort of the the graph G^{rev}, the *reverse* of graph G. However, it is unnecessary to compute G^{rev}. We just run the original topological sort algorithm on G with the following simple modification:

- The `count` variable is globally initialized to 1 instead of n.
- In the POSTVISIT procedure, we increment `count` instead of decrementing it.

Here then is the code; we `count` to `rank`:

```
STRONG_COMPONENT_ALGORITHM($G$)
    INPUT: Digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
    OUTPUT: A permutation $per[1..n]$ of $V$
        ▷ Initialization
1.      For $i = 1, \ldots, n$, $color[i] =$ unseen.
2.      Declare array $per[1..n]$.
3.      rank = 1 ◁ Global counter
        ▷ Main Loop
4.      For $i = 1, \ldots, n$,
5.          If ($color[i] =$ unseen),
6.              $DFS_0(i)$ ◁ Updates per with postorder ranking
        ▷ Calls Main Subroutine
6.      STRONG_COMPONENT_SUBROUTINE($G, per$)
```

The code for DFS_0 is as follows:

```
$DFS_0(i)$
      INPUT: Vertex $i$ in $G = (V, E)$
      OUTPUT: Update of array $per[1..n]$
            ▷ Main Loop
3.          For each vertex $v$ adjacent to $i$,
4.                If ($color[v] = \texttt{unseen}$),
5.                      $DFS_0(v)$ ◁ Recursion
6.          $per[\texttt{rank}++] = i$ ◁ Give vertex $i$ its rank
```

We may verify that the permutation $per[1..6]$ computed by our algorithm on G_6 is precisely that shown in (9).

Remarks. Tarjan [3] 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 [2].

Exercises

Exercise 7.1: 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. ♢

Exercise 7.2: Give an algorithm to compute the **essential** of a DAG. An edge $u-v$ is **inessential** if there exist $w \in V \setminus \{u, v\}$ such that there is a path from u to w and a path from w to v. ♢

End Exercises

References

[1] 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.

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

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