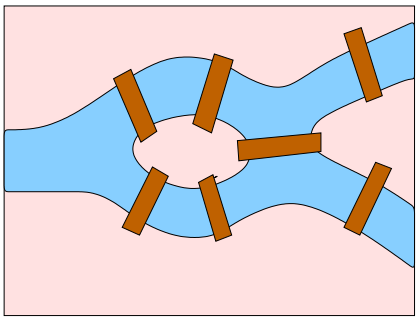
"Liesez Euler, liesez Euler, c'est notre maître á tous"

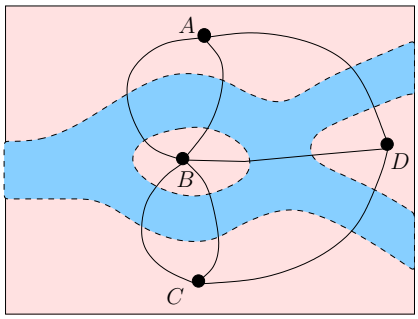
(Read Euler, read Euler, he is our master in everything)
— Pierre-Simon Laplace (1749–1827)

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 essence 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 Boruvka's (1926) and Jarník (1930) minimum spanning tree algorithm, and Dijkstra's shortest path algorithm (1959). Tarjan [7] 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 [3]; for algorithmic treatments, see Even [5] and Sedgewick [6].

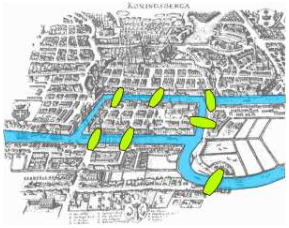


(a)



(b)

Figure 1: The 7 Bridges of Konigsberg



The real bridge
Credit: wikipedia

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) shows a graph with vertex set $V = \{1, 2, \ldots, 7\}$ representing these countries. An edge

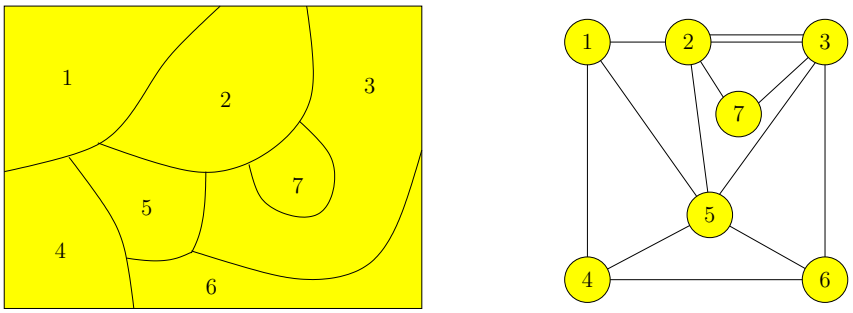
[1] This former Prussian city is now in Russia, called Kaninsgrad. See article by Walter Gautschi (SIAM Review, Vol.50, No.1, 2008, pp.3-33) on the occasion of the 300th Anniversary of Euler's birth.

[2] His paper was entitled "Solutio problematis ad geometriam situs pertinentis" (The solution of a problem relating to the geometry of position).

i–j represents the relationship between countries i and j that share a continuous (i.e., connected) common border. Thus the graph is an abstraction of the map. 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 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.



(a) (b)

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 requires the general formulation of graphs as a relationship between two independent sets V and E.

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 traversal strategies called depth-first search (DFS) and breadth-first search (BFS).

What could be impure of graphs?

shell programming again!

Some other important problems of pure graphs are: testing if a graph is planar, finding a maximum matching in a graph, and testing isomorphism of graphs.

[3] We have taken this highly suggestive notation from Sedgewick's book [6].

§1. Varieties of Graphs

In this book, "graphs" refer to either directed graphs ("digraphs") or undirected graphs ("bigraphs"). Additional 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 will be called **bigraphs**. Bigraphs are more commonly known as **undirected graphs**.

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 are natural because they satisfy a certain "umbral property" given by the following equations on set cardinality:

umbra = shade or shadow (Latin)

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

For example, let $V = \{a, b\}$. Then

$$V^2 = \{(a,a),(a,b),(b,a),(b,b)\}, \quad 2^V = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}, \quad \binom{V}{2} = \{\{a,b\}\}.$$

So $|V^2| = |2^V| = 2^2 = 4$ and $\left|\binom{V}{2}\right| = \left|\binom{2}{2}\right| = 1$. 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 bigraphs, 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

So u–v can mean (u, v) or $\{u, v\}$!

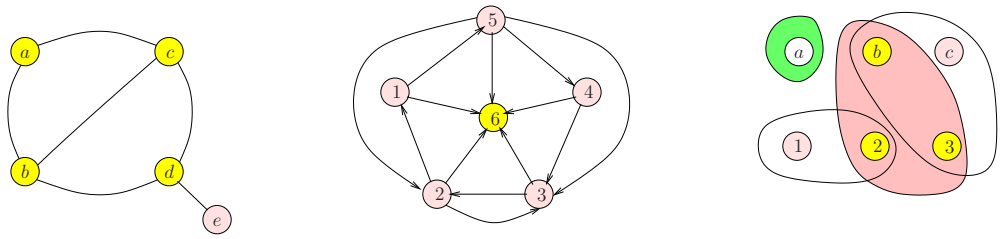
[4] While the digraph terminology is fairly common, the bigraph terminology is peculiar to this book, but we think it merits wider adoption. Students sometimes confuse "bigraph" with "bipartite graph" which is of course something else.

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). Berge [1] or Bollobás [2] is a basic reference on hypergraphs.

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.

¶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 G_6 (c) hypergraph

Figure 3: Three varieties of 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–b, c–a, d–$e\}$. 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 (or bi-directional). In Figure 3(c) we have a hypergraph on $V = \{a, b, c, 1, 2, 3\}$ with four hyperedges $\{a\}, \{1, 2\}$, $\{b, 2, 3\}$ and $\{b, c, 3\}$.

¶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 phenomenon 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. In Appendix (Lecture I), we defined special graphs such as acyclic graphs and trees. We mention note some additional classes of graphs here.

First consider bigraphs. The complete graph K_n and the complete bipartite graph $K_{m,n}$ were also defined in Lecture I Appendix. See Figure 4(a,b) for the cases of K_5 and $K_{3,3}$. In general, **bipartite graphs** are those whose vertex set V can be partitioned in two disjoint sets $A \uplus B = V$ such that each edge is incident on some vertex in A and on some vertex in B. Instead of writing $G = (V, E)$, we may write $G = (A, B, E)$ for such a bipartite graph with $E \subseteq A \times B$. Bipartite graphs are important in practice because they model relations between two sets of entities (man versus woman, students versus courses, etc).

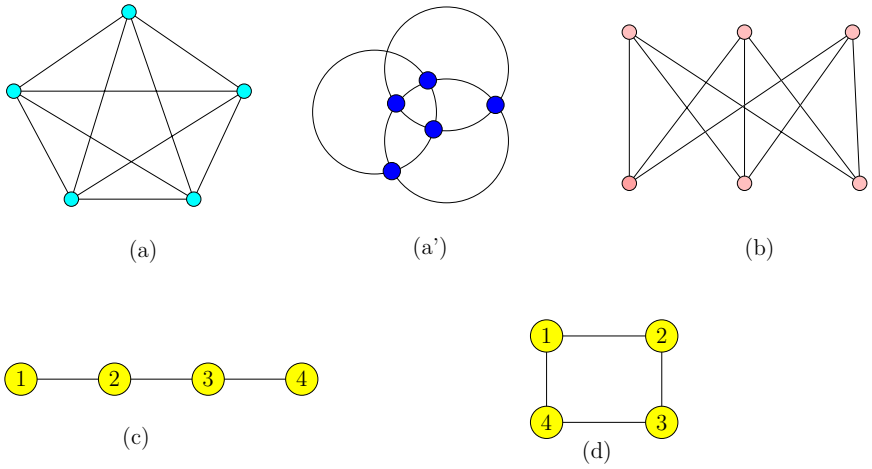


Figure 4: (a) K_5, (a') K_5, (b) $K_{3,3}$, (c) L_4, (d) C_4

Planar graphs are those bigraphs which can be embedded in the Euclidean plane such that no two edges cross. Informally, it means that we draw them on a piece of paper so that the curves representing edges do not intersect. 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 so-called Kuratowski graphs K_5 and $K_{3,3}$ in Figure 4(a,b). We have re-drawn K_5 in Figure 4(a'), this time to minimize the number of edge crossings. The graph $K_{3,3}$ is also known as the "utilities graph". The proof that these two graphs are nonplanar are found in Exercises (in this section, and also in Appendix of Chap. 1).

Why is $K_{3,3}$ so called?

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 of these.

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 isomorphism class K_4. Another example of two isomorphic graphs is the Kuratowski graph K_5, but represented differently as in Figure 4(a) and Figure 4(a'). We could sometimes avoid isomorphism classes by picking a canonical representative from the class. 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. Likewise, we define L_n and C_n above as graphs on the vertex set $\{1, 2, \ldots, n\}$ with edges i–$(i + 1)$ for $i = 1, \ldots, n - 1$ (and n–1 for C_n). Nevertheless, it should be understood that we intend to view L_n and C_n as an isomorphism class.

¶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) Convince the citizens of Königsberg that there is no way to traverse all seven bridges in Figure 1(a) without going over 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 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 $W = \{w \in V : w\text{–}v \in E, w\text{–}u \in E\}$. 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? Give exact values for $n \leq 5$. Give upper and lower bounds for general n. ♢

Exercise 1.5: Let $G = (V, E)$ be a hypergraph where $|e \cap e'| = 1$ for any two distinct hyperedges $e, e' \in E$. Also, the intersection of all the hyperedges in E is empty, $\cap E = \emptyset$. Show that $|E| \leq |V|$. ♢

Exercise 1.6: A hypergraph $G = (V, E)$ is **connected** if it can be written as a union to two non-empty hypergraphs, $G = G_0 \uplus G_1$ where the vertex sets of G_0, G_1 are disjoint. A **cycle** in G is a sequence $[u_0, e_1, u_1, e_2, u_2, \ldots, u_{k-1}, e_k]$ of alternating vertices u_i and hyperedges e_i such that $u_i \in e_i \cap e_{i+1}$ (assume $e_0 = e_k$). If G is connected, then G has no cycles iff

$$\sum_{e \in E} (|e| - 1) = |V| - 1.$$

♢

Exercise 1.7: Consider the decomposition of 2^V into **symmetric chains** $E_r \subset E_{r+1} \subset \cdots \subset E_{n-r}$ where each E_k is a subset of V of size k, and $|V| = n$. For instance, if $V = \{1, 2, 3\}$, then 2^V is decomposed into these 3 symmetric chains:

$$\emptyset \subset \{3\} \subset \{2, 3\} \subset \{1, 2, 3\}, \quad \{2\} \subset \{1, 2\}, \quad \{1\} \subset \{1, 3\}.$$

(a) Please give the decomposition for $V = \{1, 2, 3, 4\}$.
(b) Show that such a decomposition always exists. Use induction on n.
(c) How many symmetric chains are there in the decomposition? ♢

Exercise 1.8: (Sperner) Let $G = (V, E)$ be a hypergraph with $n = |V|$ vertices. Clearly, $|E| \leq 2^n$ and the upper bound is achievable. But suppose we require that no hyperedge is properly contained in another (we then say G is **Sperner**).
(a) Prove an upper bound on $|E|$ as a function of n in a Sperner hypergraph. HINT: If $E = \binom{V}{\lfloor n/2 \rfloor}$, then (V, E) is Sperner and $|E| = \binom{n}{\lfloor n/2 \rfloor}$. Try to use the symmetric decomposition in the previous Exercise.
(b) Characterize those graphs which attain your upper bound. ♢

Exercise 1.9: A "trigraph" $G = (V, E)$ is a hypergraph where $E \subseteq \binom{V}{3}$. These are also called 3-uniform hypergraphs. Each hyperedge $f \in E$ may also be called a **face**. 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. We say the trigraph G is **planar** if we can embed its vertices in the plane such that each face $\{a, b, c\}$ is represented by a simply region in the plane bounded by three arcs connecting the edges a–b, b–c and c–a. Show that G is planar iff its underlying bigraph is planar in the usual sense. ♢

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:

Adjacency is not *always symmetric!*

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 [4], the notation $p : v_0 \rightsquigarrow v_k$ (equivalently, $v_0 \overset{p}{\rightsquigarrow} v_k$) indicates that p is a path from v_0 to v_k. We denote the same by writing $p = (v_0 - \cdots - v_k)$. But more generally, we might also write $p = (u - v - \cdots - w - \cdots - z)$ to indicate that path p begins at u and v, passing through w and terminating at z.

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 **simple** if all its vertices, with the possible exception of $v_0 = v_k$, are distinct. The path p is **closed** if $v_0 = v_k$ and $k > 0$. Any trivial path is simple but not closed (because we require $k > 0$). Thus, a path can[5] only be closed if it has at least one edge. 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 un-generalized concept. The following is easy to see:

distance notation $\delta(u, v)$

- (Non-negativity) $\delta(u, v) \geq 0$, with equality iff $u = v$.
- (Triangular Inequality) $\delta(u, v) \leq \delta(u, w) + \delta(w, v)$.
- (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**.**

Suppose $(v_0 - v_1 - \cdots - v_k)$ is a **minimum link path** (sometimes called "shortest path") between v_0 and v_k. Thus, $\delta(v_0, v_k) = k$. Then we have the following basic property: for all $i = 0, 1, \ldots, k$, $\delta(v_0, v_i) = i$. This is also called the "dynamic programming principle" for minimum link paths (we will study dynamic programming in Lecture 7).

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

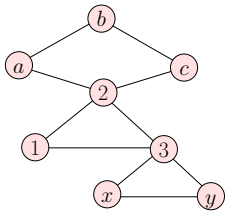
$$p = (v_0 - v_1 - \cdots - v_k), \quad q = (u_0 - u_1 - \cdots - u_\ell).$$

If the target of p equals the source of q, 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

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

[5] This technical decision will have consequences when we discuss cycles.

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 $p';p''$. E.g., if $p = (1{-}2{-}a{-}b{-}c{-}2{-}3{-}x{-}y{-}3{-}1)$ and



$$p' = (1{-}2), \quad q = (2{-}a{-}b{-}c{-}2), \quad p'' = (2{-}3{-}x{-}y{-}3{-}1).$$

then we can excise q to obtain $p';p'' = (1{-}2{-}3{-}x{-}y{-}3{-}1)$. Whenever we write a concatenation expression such as "$p;q$", it is assume that the operation is well-defined.

¶8. Cycles. Two closed 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. Note that cyclic equivalence is only applicable if p is a closed path because the expression "$r';r$" is only well-defined if the target of p (i.e., target of r') is equal to the source of p (i.e., source of r). By symmetry, we also see that q must be a closed path.

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

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

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 $p = (v_0{-}v_1{-}\cdots{-}v_k)$, $r = (v_0)$ and $r' = p$, then we see that $p = r';r = r;r' = q$. This proves that p is cyclic equivalent to itself. It is also clear that if $p \equiv q$ then $q \equiv p$. We may also check that transitivity holds (Exercise). Thus we conclude that cyclic equivalence is a mathematical equivalence relation. An equivalence class of closed paths is called a **cycle**. If the equivalence class of p is the cycle Z, we call p a **representative** of Z; if $p = (v_0{-}v_1{-}\cdots{-}v_k)$ then we write Z using square brackets $[\cdots]$ in any one of these forms:

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

For example, the cycle defined by the closed path in (3) can be denoted $[1{-}2{-}3{-}4]$ or $[2{-}3{-}4{-}2]$. In general, if p has $k+1$ vertices, then when we explicitly list the vertices of p in the cycle notation $[p]$, we only list k vertices since the last vertex may be omitted. The Exercise will explore the problem of detecting if two given paths p, q are cyclic equivalent, $[p] = [q]$.

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

- The **length** of Z is the length of p. E.g., $[1{-}2{-}3{-}4]$ or $[2{-}3{-}4{-}2]$ has length 4.
- 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

$$[1{-}2{-}a{-}b{-}c{-}2{-}3{-}x{-}y{-}3].$$

- The **reverse** of Z is the cycle which has the reverse of p as representative. E.g., the reverse of $[1{-}2{-}3{-}4]$ is $[4{-}3{-}2{-}1]$.

It is important to consider some "boundary" cases in our definitions. Are there are cycles of length 1? According to our definitions, this cycle has the form $[v_0]$, and hence it represents the equivalence class of the closed path $(v_0{-}v_0)$. This implies that $v_0{-}v_0$ is an edge (recall we called such edges loops). But loops, by definition, do not occur in bigraphs. Hence, bigraphs has no cycles of length 1. What about cycles of length 0? such a cycle would be denoted $[\,]$, and corresponds to some trivial closed path of the form (v_0). But according to our definitions, trivial paths are not closed. Hence, there are no cycles of length 0: In summary, *cycles in digraphs have length at least* 1*, and cycles in bigraphs have length at least* 2*.*

¶9. Cyclic Digraphs and Bigraphs. Intuitively, a "cyclic graph" is one that contains a cycle. For a digraph G, this is exactly the definition: G is **cyclic** if it contains any 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 $[u, v]$.

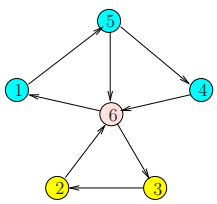
For bigraphs, we proceed as follows: first, define a closed path $p = (v_0{-}v_1{-}\cdots{-}v_{k-1}, v_0)$ to be **reducible** if $k \geq 2$ and for some $i = 1, \ldots, k$,

$$v_{i-1} = v_{i+1} \tag{4}$$

where subscript arithmetic are modulo k. In particular, when $i = k$, then (4) says $v_{k-1} = v_1$. Otherwise p is said to be **irreducible**. A cycle $Z = [p]$ is reducible iff any of its representative p is reducible. A bigraph is said to be **cyclic** if it contains some irreducible non-trivial cycle, otherwise it is **acyclic**. Each connected component of an acyclic bigraph is just a "free" tree (i.e., not rooted). Intuitively, irreducible paths contain is a bit of local memory: after we taking the edge $(u{-}v)$, we must remember not to take the "same" edge $(v{-}u)$ in the opposite direction. Recall that cycles in bigraphs have length at least 2. But cycles of length two have the form $[u, v]$, corresponding to the closed path $(u{-}v{-}u)$ that is reducible. Hence, irreducible cycles have length at least 3.

In physics, it is called "hysteresis".

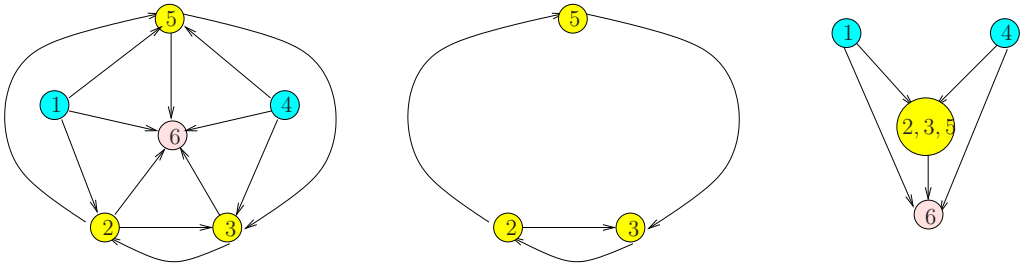
¶10. Strong Connectivity. Let $G = (V, E)$ be a graph (either di- or bigraph). Two vertices u, v in G are **connected** if there is a cycle containing both u and v. Note that we do not require the cycle to be simple, so it amounts to having a path from u to v and one from v to u. Equivalently, $\delta(u, v)$ and $\delta(v, u)$ are both finite. It is not hard to see that strong 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. Thus V is partitioned into disjoint components. If G has only one connected component, it is said to be **connected**. By definition, u and v are connected means there is a cycle Z that contains both of them. But we stress that Z need not be a simple cycle. For instance, the digraph in this margin is connected because every two vertices are connected. However, any cycle Z that contains both 1 and 2 is non-simple (Z must re-use vertex 6). The subgraph of G induced by C is called a **component graph** of G.



connected digraph

Note that in some literature, it is customary to add the qualifier "strong" when discussing components of digraphs; in that case "component" is reserved only for bigraphs. However, our definition of "component" covers both bi- and digraphs. Nevertheless, we might still use **strong components** for emphasis.

For example, the graph G_6 in Figure 5(a) has $C = \{2, 3, 5\}$ as a component. The component



(a) G_6 (b) (c)

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

graph corresponding to C is shown in Figure 5(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 $(C, C') \in E^c$ are 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 acyclic. 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.

¶11. 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.

Motto: "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: Let G be a bigraph. A Hamilton path of G is a simple path that passes through every vertex of G. A Hamilton circuit is a simple cycle that passes through every vertex of G. Show that $K_{3,5}$ has no Hamilton path or Hamilton circuit ♢

Exercise 2.3: 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 $N(m)$. ♢

Exercise 2.4: Give an algorithm which, given two cycles $p = [v_1 - \cdots - v_k]$ and $q = [u_1 - \cdots - u_\ell]$, determine whether they represent the same cycle. E.g., $p = [1, 2, 3, 4, 5]$ and $q = [3, 4, 5, 1, 2]$ are equivalent simple cycles, but p and $q' = [3, 4, 5, 2, 1]$ are not. Again, $p = [1, 2, 1, 3, 4, 5]$ and $q = [1, 3, 4, 5, 1, 2]$ are equivalent non-simple cycles. The complexity of your algorithm may be $O(k^2)$ in general, but should be $O(k)$ when q is a simple cycle. Note: Assume that vertices are integers, and the cycle $p = [v_1 - \cdots - v_k]$ is represented by an array of k integers. ♢

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

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

  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

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

  This is supposes to be the **list-of-lists form** of adjacency lists. Another variant where we assume the vertex set is $\{1, \ldots, n\}$ and we have an array $A[1..n]$ where $A[i]$ points to the adjacency list of vertex i. This is the **array-of-lists form**. In practice, it is much easier to program the array-of-lists form. Most of our examples will use this form of adjacency lists. But the two forms are inter-convertible in $O(n + m)$ time (Exercise).

- **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

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

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

¶12. Size Parameters. Two size parameters are used in measuring the input 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 $0 \leq m \leq n^2$. 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.

The $O(m+n)$ time bound is the "gold standard" for pure graph algorithms: aim for this bound whenever possible.

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) $m \leq 3n-6$ in such graphs (Exercise). Planar graphs are those that can be drawn on the plane without any crossing edges.

¶13. Arrays and Attributes. If A is an array, and $i \leq j$ 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.

¶14. 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:

$$\begin{array}{l} \texttt{white} \Rightarrow \texttt{gray} \Rightarrow \texttt{black}, \\ \texttt{unseen} \Rightarrow \texttt{seen} \Rightarrow \texttt{done} \\ 0 \Rightarrow 1 \Rightarrow 2. \end{array} \tag{5}$$

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}. You must show that your algorithm has the stated running time.

We want you to solve two versions of this problem:
(a) Assume an array representation of the adjacency linked list (i.e., the vertices is $V = \{1, 2, \ldots, n\}$ and you have a array of linked list.
(b) Assume a linked-list-of-linked-lists representation.

PROBLEM REQUIREMENT: Destructive or Conservative? Your algorithm may directly modify G into its reverse (this is the destructive form). Alternatively, you may preserve the input G (constructive form). Either version is accepted – do the simplest version you can. ♢

In `C++`*, you indicate that an argument G is to be conserved by tagging the argument with* `const`.

> This exercise raises a general question. Q: *If we need an library routine to transform an input graph G, which of the following is "better"? (i) Destructive Algorithm that directly modify the graph G. (ii) Conservative Algorithm that preserve G and returns a new graph representing the transformed G?*
> A: The answer to this question may depend on the application. But generally, I feel the destructive version is more useful in a library. You might object, saying you want to keep the original graph. My response is that you can first create a copy before calling the routine. On the other hand, if you do not care to keep the original input, then the conservative algorithm wastefully creates a new graph, forcing us to explicitly delete the old graph. At this point, the score is even: the destructive approach forces you to copy the graph first, and the conservative approach forces you to delete an unwanted copy. But a library routine with the freedom to destroy the input graph has opportunities for various optimization (like reusing nodes rather than creating new ones, which requires memory allocation and destruction).

Exercise 3.2: Let G is a connected planar bigraph. Let $E(G)$ be any embedding of G in the plane, but in such a way that the curves (representing edges) are pairwise disjoint. The plane is divided by these curves into connected regions called "faces" by this figure. Note that exactly one of these faces is an infinite face. For instance, the graph embedding in Figure 3(a) has 3 faces, while the embedding in Figure 3(b) (viewed as a bigraph for our

purposes) has 9 faces.
(a) Show that if an embedding of G has f faces, $v = |V|$ vertices and $e = |E|$ edges then the formula $v-e+f = 2$ holds. E.g., in Figure 3(a) (resp., Figure 3(b)) $v-e+f = 5-6+3 = 2$ (resp., $v - e + f = 6 - 13 + 9 = 2$). This proves that f is independent of the choice of embedding. HINT: use induction on e. Since G is connected, $e \geq v - 1$.
(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. Show this. ♢

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. Then counting the number of $(edge, face)$ incidences in two ways, from the viewpoint of edges, and from the viewpoint of faces. From this, obtain an upper bound on the number of faces, which should contradiction Euler's formula $v - e + f = 2$. ♢

Exercise 3.6: Give an $O(m + n)$ time algorithms to inter-convert between an array-of-lists version and a list-of-lists version of the Adjacency Graph representation. ♢

End Exercises

§4. Breadth First Search

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

A **graph traversal** is a systematic method to "visit" each vertex and each edge of a graph. In this section, we study two main traversal methods, known as Breadth First Search (BFS) and Depth First Search (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 (see [5, 6]). Such explorations is still the basis for some popular computer games.

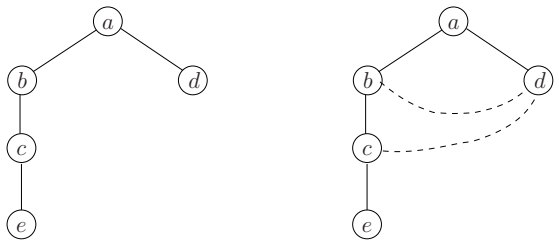
¶15. Generic Graph Traversal. The idea is to mark the vertices with two "colors", intuitively called `unseen` and `seen`:

```
Generic Graph Traversal:
Input: G = (V, E; s_0) where s_0 is any source node
    Color all vertices as initially unseen.
    Mark s_0 as seen, and insert into a container ADT Q
    While Q is non-empty
        u ← 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, not just those reachable from a single source s_0, we can use another driver routine which invokes this traversal routine with different choices for source nodes (see below). The set Q is represented by 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. These two algorithms are the main focus of this chapter.

Both traversal methods apply to digraphs and bigraphs. However, BFS is typically described for bigraphs only and DFS for digraphs only. We generally follow this tradition unless otherwise noted. 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".



(a) (b)

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

$$\delta(s_0, u) < \delta(s_0, v) < \infty. \tag{6}$$

If $\delta(s_0, u) = \infty$, then u will not be visited from s_0. The BFS algorithm does not explicitly compute the relation (6) to decide the next node to visit: below, we will prove that this is a consequence of using the queue data structure.

¶16. The BFS Shell. The key to the BFS algorithm is the **queue** data structure. This is an ADT that supports the insertion and deletion of items following the First-In First-Out (FIFO) discipline. If Q is a queue, we denote the insert and delete of a node u by

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

respectively. To keep track of the status of vertices we will use the color scheme (5) in the previous section. We could use three colors, but for our current purposes, two suffice: `unseen`/`seen`. Here is the BFS algorithm formulated as a shell program:

```
BFS Shell
    Input:   G = (V, E; s_0) a graph (bi- or di-).
    Output:  This is application specific.
    ▷ Initialization:
0       [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       [VISIT(s_0, nil)]   ◁ Visit s_0 as root
    ▷ Main Loop:
        while Q ≠ ∅ do
3           u ← Q.dequeue().   ◁ Begin processing u
4           for each v adjacent to u do   ◁ Process edge u−v
5               [PREVISIT(v, u)]   ◁ Previsit v from u
6               if v is unseen then
7                   Color v seen
8                   [VISIT(v, u)]   ◁ Visit v from u
9                   Q.enqueue(v).
10          [POSTVISIT(u)]
11      [CLEANUP(G)]
```

This BFS shell program contains the following shell macros

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

which will be application-specific. These macros may be assumed[6] to be null operations unless otherwise specified. The term "macro" here suggests only a small[7] and non-iterative piece code that can be executed in $\mathcal{O}(1)$ time. 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.

in computing, macro = small?

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

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

[7] Of course, "macro" literally means "big", and the terminology arises in assembly language programming where a macro is a block of code. But from the complexity theoretic viewpoint, $O(1)$ is small. Below, the Recursive DFS Shell will allow an exception for this $O(1)$ restrictions for macros.

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 which are VISITed if we run the BFS algorithm on $(G; s_0)$, and the vertices are printed in the order they are visited. 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

$$(a, b, d, c, e) \qquad \text{and} \qquad (a, d, b, c, e). \tag{8}$$

The particular BFS listing depends on how the adjacency list of each node is ordered. We can produce such a listing just by enumerating the vertices of the BFS tree in the order they are visited.

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

- Suppose you wish to print a list (without repetition) of all the vertices reachable from s_0. You can make VISIT(v, u) print some identifier (key, name, etc) associated with v. This would produce the BFS order at s_0. Alternatively, you can make POSTVISIT(u) print the identifier associated with u.

  Other macros can remain null operations. Intuitively, these two orderings correspond to preorder and postorder traversal of trees.

- 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. (As we will see, this depth has intrinsic meaning for the graph.) 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 interpret a node u to be unseen iff $d[u] = \infty$.

- Suppose you wish to detect cycles in an bigraph. Let us assume the input graph is connected. In PREVISIT(v, u), if v is seen, then you have detected a cycle, and you can immediately return "CYCLIC".

  You will only reach the final CLEANUP(G) (Step 11) if you did not return earlier through PREVISIT. So, CLEANUP simply returns "ACYCLIC".

¶18. 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. This notation will be justified shortly when we related $\delta(v)$ to link distance; but for now, it is just depth 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 instant 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:*

$$\delta(u_1) \leq \delta(u_2) \leq \cdots \leq \delta(u_k) \leq 1 + \delta(u_1). \tag{9}$$

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 (9) 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 (9). **Q.E.D.**

This lemma shows that $\delta(u_i)$ is monotone non-decreasing with increasing index i. 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 the first property about the BFS algorithm:

Lemma 3. *The depth $\delta(u)$ of a vertex u in the BFS tree is equal to the link distance from s_0 to u, i.e.,*

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

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 $\delta(u_i) \leq \delta(u_{i-1}) + 1$. By telescopy, we get $\delta(u_k) \leq k + \delta(u_0) = k$. 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 in the BFS tree. **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 has the property that $\delta(u_i) \leq \delta(u_j)$ for all $i < j$.

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.*

¶19. Classifying 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 b–d 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 c–d in Figure 6(b).
- **Unseen edges**: these are edges that are not used during the computation. Such edges involve only 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 (Classification of Bigraph Edges). *If G is a bigraph, the above classification of its edges is complete.*

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

¶20. Applications of Bigraph Edge Classification. Many basic properties of link distances can be deduced from our classification. We illustrate this by showing two consequences here.

Try proving them without the classification theorem!

1. Let T be a BFS tree rooted at v_0. Consider the DAG D obtained from T by adding all the cross-level edges. All the edges in G are given a direction which is directed away from v_0 (so each edge goes from some level $i \geq 0$ to level $i+1$). CLAIM: *Every minimum link path starting from v_0 appears as a path in the DAG D.* In proof, the classification theorem implies that each path in G is a minimum link path, as there are no edges that can skip a level.

2. Consider a bigraph G with n vertices and with a minimum link path $p = (v_0 – v_1 – \cdots – v_k)$. CLAIM: *If $k > n/2$ then there exists a vertex v_i ($i = 1, \ldots, k-1$) such that every path from v_0 to v_k must pass through v_i.* To see this, consider the BFS tree rooted at v_0. This has more than $n/2$ levels since $\delta(v_0, v_k) = k > n/2$. If there is a level i ($i = 1, \ldots, k-1$) with exactly one vertex, then this vertex must be v_i, and this v_i will verify our claim. Otherwise, each level i has at least two vertices for all $i = 1, \ldots, k-1$. Thus there are at least $2k = (k+1) + (k-1)$ vertices ($k+1$ vertices are in the path p and $k-1$ additional vertices in levels $1, \ldots, k-1$). But $k > n/2$ implies $2k > n$, contradiction.

¶21. Driver Program. In our BFS algorithm we are given a source vertex $s_0 \in V$. This guarantees that we visit precisely those 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 Shell
    Input:    G = (V, E) a graph.
    Output:   Application-dependent.
    ▷ Initialization:
1       Color all vertices as unseen.
2       DRIVER_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 DRIVER_INIT to our collection (7). Since each call to BFS produces a tree, the output of the BFS Driver is a **BFS forest** of the input graph G. It is clear that this is a spanning forest, i.e., every node of G occurs in this forest.

¶22. 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 $\Theta(n + m)$.

¶23. Application: Computing Connected Components. Suppose we wish to compute the connected components of a bigraph G. Assuming $V = \{1, \ldots, n\}$, we will use 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 DRIVER_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.

In some applications (e.g., Boruvka's algorithm in Chap.V), it is more convenient to assume that $C[u]$ is the index of a vertex in the connected component of u. We can easily modify the above algorithm to achieve this.

¶24. Application: Testing Bipartiteness. A graph $G = (V, E)$ is **bipartite** if V can be partitioned into $V = V_1 \uplus V_2$ such that if u–v is an edge then $u \in V_1$ iff $v \in V_2$. In the following we shall assume G is a bigraph, although the notion of bipartiteness applies to digraphs. 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 bipartite. We use the Driver Driver to 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 comprises 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 implementing the Bipartite Test above, and generally in our recursive routines, it is useful to be able to jump out of nested macro and subroutine calls. For this purpose, `Java`'s ability to **throw exceptions** and to **catch exceptions** is very useful. In our bipartite test, BFS can immediately throw an exception when it finds a level-edge. This exception can then be caught by the BFS Driver program.

Exercises

IMPORTANT: In this chapter, answers that that could be reduced to BFS (and later, DFS) should be solved using our shell programs. In other words, you only need to expand the various macros. The reason for this "straightjacket" approach is pragmatic — grading your solutions would be much easier. Otherwise, there are many trivial variations of the BFS and DFS programs (such as whether you change colors before or after visiting a node, etc).

Note for students!

Exercise 4.1: Prove Lemma 1 (p. 15, Lect. 6), showing 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: Modify the connected components algorithm for bigraphs so that the output array $C[1..n]$ has the property that each $C[u]$ refers to an actual vertex in the connected component of u. ♢

Exercise 4.4: Let $\delta(u)$ be the depth of u in a BFS tree rooted at s_0. If u–v, show:
(a) $\delta(v) \leq 1 + \delta(u)$.
(b) In bigraphs, $|\delta(u) - \delta(v)| \leq 1$.
(c) In digraphs, the inequality in (a) can be arbitrarily far from an equality. ♢

Exercise 4.5: Reorganize the BFS algorithm so that the coloring steps are folded into the shell macros of INIT, VISIT, etc. ♢

Exercise 4.6: Let us explore the following algorithm: in the BFS algorithm, suppose we use a stack instead of a queue. In ¶17, we gave several applications of BFS. Which of these applications will remain valid after this modification? ♢

Exercise 4.7: Fill in the shell macros so that the BFS Algorithm will correctly classify every edge of the input bigraph. ♢

Exercise 4.8: Recall that the edges of a bigraph is classified into four types by BFS: tree, cross, level, unseen. We now want to produce a BFS classification of the edges of a digraph. We keep the four types found in bigraphs, but we want to add two new types of edges. These will be back edges $u-v$ that goes from level i to level j with $i > j$. These back edges will be classified as **ancestor** or **non-ancestor** depending on whether v is an ancestor of u or not. (a) Prove that the above BFS classification of the edges of a digraph G into tree, cross, level, unseen, ancestor, non-ancestor is complete (i.e., no other edges arise) and minimal (i.e., each type of edge can arise).
(b) Now turn the classification of part(a) into a "computational classification". I.e., devise an algorithm to classify every edge of G according to (a). Recall that you must use shell programming, and try to be as efficient as possible.
(c) Analyze the complexity of your algorithm in part (b). ♢

Exercise 4.9: 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 4.10: 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 4.11: 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 $V_0 = \{i \in V : B[i] = \texttt{false}\}$ and $V_1 = \{i \in V : B[i] = \texttt{true}\}$ is a witness to the bipartiteness of G. In the case of NO, it returns an odd cycle. ♢

Exercise 4.12: 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 4.13: Let $k \geq 1$ 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, $c(u) \neq c(v)$. 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? ♢

Exercise 4.14: Let $G = (V, E)$ be a bigraph on n vertices. Suppose $n+1$ is not a multiple of 3. If there exists vertices $u, v \in G$ such that $\delta(u, v) > n/3$ then there exists two vertices whose removal will disconnect u and v, i.e., $\delta(u, v)$ will become ∞. ♢

End Exercises

§5. Nonrecursive Depth First Search

Depth First Search (DFS) turns out to be much more subtle than BFS. To appreciate the depth (no pun intended) of DFS, we take an unusual route of first presenting a non-recursive solution, based on the generic graph traversal framework of ¶15. We call this formulation the **Nonrecursive DFS algorithm**, to distinguish it from the **Standard DFS** algorithm which is a recursive one.

¶25. DFS Outline. Here is a general account of DFS: as in BFS, we want to visit all the vertices that are reachable from an initial source s_0.

> *Starting the search from the source s_0, the idea is to go as deeply as possible along any path without visiting any vertex twice. When it is no longer possible to continue a path (we reached a leaf), we backup towards the source s_0. We only backup enough for us to go forward in depth again.*

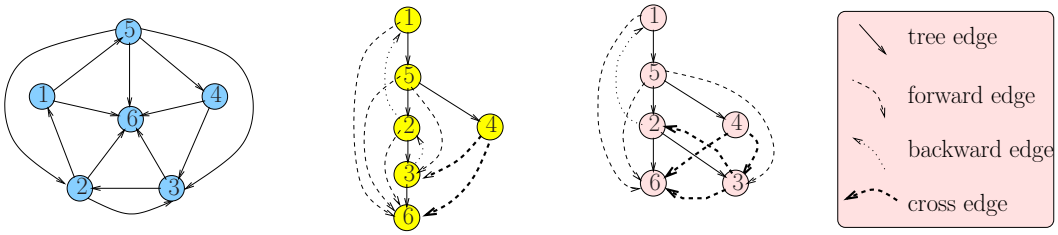
(DFS Outline)

The critical data structure derived from this informal outline is the so-called **DFS tree** underlying this computation. The edges of this tree are precisely those $u-v$ such that v is `seen` from u. This tree is not unique because it depends on the order in which we scan the adjacency list of a given vertex. The stack data structure is just right for for organizing this search.

In illustration, consider the[8] digraph G in Figure 7(i) Starting from the source vertex 1, one possible path to a leaf is $(1-5-2-3-6)$. From the leaf 6, we backup to vertex 2, from which point we can advance to vertex 3. Again we need to backup, and so on. The DFS tree is a trace of this search process, and is shown in Figure 7(ii). The non-tree edges of the graph are shown

[8] Reproduced from Figure 3(b), for convenience.

in various forms of dashed lines. For the same graph, if we visit adjacent vertices in a different order, we get a different DFS tree, as in Figure 7(iii). However, the DFS tree in Figure 7(ii) is the "canonical solution" if we follow our usual convention of visiting vertices with smaller indices first.



(i) digraph G_6 (ii) Canonical DFS tree (iii) Another DFS tree

Figure 7: Two DFS trees for digraph G_6.

¶* 26. Nonrecursive DFS. We describe a version of DFS that is parallel to BFS (¶16). The key difference is that BFS uses a queue data structure while DFS uses a stack data structure. Recall that a stack is an ADT that supports the insertion and deletion of items following a Last-in First-out (LIFO) discipline. Conceptually, we would like to derive the DFS algorithm just by replacing the queue in the BFS algorithm by a stack. Insertion and deletion from a stack S is denoted

$$S.\texttt{push}(u), \qquad u \leftarrow S.\texttt{pop}().$$

> The reader who wants to just learn the basic DFS Algorithm may go directly to the next section that describes a simpler, recursive formulation of DFS. We develop the nonrecursive DFS here in order to expose the subtleties that might be hidden in the standard formulation.

To motivate the following development, we must understand that in most applications of DFS, it is not just a matter of "traversing all nodes". We need to process the nodes in three distinct phases called PREVISITs, VISITs and POSTVISITs and these must to be correctly ordered if DFS is solve these applications. Here are the ground rules for this ordering:

- Each reachable node must be PREVISITed, VISITed and POSTVISITed, in this order.
- A node v is PREVISITed when it is reached (or seen) for the first time: that means we processed an edge of the form $(u{-}v)$, for some u.
- To VISIT a node u means, among other application-dependent things, the processing of the edges $(u{-}v)$, for each v adjacent to u.
- A node u is POSTVISITed after all its adjacent nodes v have been POSTVISITed (and hence VISITed).
- The entire process must take $O(n+m)$ time. In particular, we process each edge at most once and this entails the adjacency-list representation of graphs.

To fulfill these ground rules, we propose to color nodes using a four-color scheme:

$$\texttt{unseen} \rightarrow \texttt{seen} \rightarrow \texttt{visited} \rightarrow \texttt{done}.$$

There are three color transitions, and they correspond naturally to the events of PREVISITing, VISITings and POSTVISITing a node.

Let us consider stack events: these are basically pushing and popping of nodes. A node u is first pushed on the stack when we PREVISIT it. Intuitively, when this copy of u is popped later, we should VISIT it. Unfortunately, in between the pushing and popping of this u, other events may disrupt this simple scenario. There are two issues.

Recall VISITing u means we will examine each node v that is adjacent to u, PREVISITing v if it has not been PREVISITed. But how do know when to POSTVISIT u? The answer is to push u *just before* we visit any node adjacent to u. But we had just popped u! Thus this second push of u amounts to retaining u in the stack; call this the **retained copy** of u. This raises the first issue: when we pop a node u, how do we know whether we are to VISIT or POSTVISIT u? In other words, how do we distinguish between an original copy of u from the retained copy of u? Answer: check the current color of u to decide. If it is `seen` then we are to VISIT u and change its color to `visited`, and if it is `visited` then we POSTVISIT u and change its color to `done`. The second issue is this: after u has been PREVISITed, there may be other edges from which we will PREVISIT u again. *It is important to push a copy of u on the stack for each such PREVISIT of u.* The copies that are pushed earlier will be popped after the later copies according to the stack discipline. But once a copy of u is popped, all the earlier copies of u will be rendered obsolete (they eventually will be popped as `done` nodes and these have no effect).

We remark that color is a global property: when a node u is given a new color, each copy of u on the stack will automatically switch to the new color. Thus the color of a popped node might be different than the color it had when initially pushed on the stack. Now we are ready to present the DFS algorithm in pseudo-code:

```
Nonrecursive DFS Shell
    Input:   G = (V, E; s_0) a graph (bi- or di-)
    Output   Application dependent
    ▷ Initialization:
0       [INIT(G)]   ◁ If this is a standalone shell, then color all vertices as unseen
1       Color s_0 seen, and initialize the stack S with just s_0.
    ▷ Main Loop:
2       while S ≠ ∅ do:
3           u ← S.pop()
4           Switch (color(u)) :   ◁ three cases
5               CASE seen:
6                   Color u visited, and [VISIT(u)]
7                   S.push(u)   ◁ u is now a "retained copy"
8                   for each v adjacent to u do:
9                       If (v is unseen) then
10                          Color v seen, and [PREVISIT(v, u)];
11                      If (v is seen) then
12                          S.push(v).   ◁ The stack may have older copies of v already
13              CASE visited:
14                  Color u done, and [POSTVISIT(u)]
15              CASE done:
16                  No-Op (Do nothing)
17      [CLEANUP(G)]   ◁ Use if standalone shell .
```

the v's are popped in the reverse *order of their occurrence in u's adjacency list.*

This code has a switch-statement (Line 4) which activates one of three cases (Lines 5, 13, 15). The 3 cases corresponds[9] to the possible colors of the node u; the color `unseen` is not needed.

Like BFS, this DFS program has two main loops: an outer while-loop (Line 2) and an inner for each-loop (Line 8). Note that the color transitions `unseen` → `seen` → `visited` → `done` for node u are performed just before the appropriate events (Lines 10, 6, 14). The placement of the VISIT macros is also different from BFS: in BFS, we VISIT a vertex when it is first inserted into the queue; but in DFS, we VISIT a vertex after it is removed from the stack.

To understand this code, it is most interested to consider the CASE where u is `seen`, Line 5. We had just popped u, and in Line 7 we push u again! Hence we call this pushed copy of u the "retained copy". But note that the retained copy has the color `visited`, not `seen`. There may be other copies of u deeper in the stack, which we now consider as "dead". It is this retained copy, not the dead ones that will eventually be popped and executed under CASE `visited`. Why? Because no other copies of u will again be pushed on the stack – we see this in Lines 11-12 when vertices are pushed on the stack, and only `seen` vertices are pushed. Thereafter, any dead copy that is popped will be executed under CASE `done`, which is a no-op.

[9] Unlike the case statements in programming languages like `C` or `C++`, we assume that the case statements are insulated from each other (i.e., the execution of a case does not run into the next case). In these programming langauges, one must explicitly insert a "break" statement at the end of each case to achieve insulation.

More Powerful Stacks. Lines 11-12 is interesting because there may already be copies of v in the stack when we push v. This behavior is deliberate. Intuitively, the latest copy of v kills the older copies. Ideally, we should remove the older copies of v, but that would require a more powerful stack than we want to assume here. As consequence, the stack size is not bounded by n (number of vertices in the graph) but by m (the number of edges in the graph). We could use a more powerful stack with an operation called S.`delete`(u). It is not hard to implement this operation. Stacks operations are usually implemented efficiently in $O(1)$ time using an array. But to ensure $O(1)$ performance in the presence of S.`delete`(u), we can now use a linked-list.

Killing the older copies of v is the essence of DFS

¶27. DFS Driver. Finally, if we need to visit all vertices of the graph, we can use the following DFS Driver Program that calls Nonrecursive DFS repeatedly:

```
DFS Driver
    Input:   G = (V, E) a graph (bi- or di-)
    Output:  Application-specific
1       DRIVER_INIT(G)
2       Color each vertex in V as unseen.
3       for each v in V do
4           if v is unseen then
5               DFS(V, E; v)    ◁ Either Nonrecursive or Standard DFS
6               CLEANUP(V, E; v)
7       DRIVER_CLEANUP(G)
```

We view these algorithms as shell programs whose complete behavior depend on the specification of the embedded shell macros, which are presumed to be null unless otherwise specified:

$$\left.\begin{array}{l}\text{PREVISIT,}\quad \text{VISIT,}\quad \text{POSTVISIT,}\\ \text{INIT,}\quad \text{DRIVER\_INIT,}\quad \text{CLEANUP,}\quad \text{DRIVER\_CLEANUP.}\end{array}\right\} \tag{10}$$

¶28. Computing the 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). This tree can easily be constructed by appropriate definitions of INIT(G, s_0) and PREVISIT(v, u) in the above shells. Here are the details.

Let us represent this tree by a **parent array** $p[v \in V]$ where $p[v]$ denotes the parent of a node v in the DFS tree. Initially, $p[v] =$ `nil` for all $v \neq s_0$ and the root is indicated by $p[s_0] = s_0$. In the PREVISIT(v, u) macro, we will simply assign $p[v] \leftarrow u$. Note that $p[v]$ may be reassigned several times, since v can be pushed on the stack several times. The parent of v is the value obtained by the last assignment to $p[v]$.

We prove a basic fact about DFS. Although the lemma is intuitive, the correct proof is quite subtle, involving some non-obvious induction.

Lemma 6 (Unseen Path). *Let $u, v \in V$.*
Then v is a descendant of u in the DFS tree if and only if at the time instant that u was first seen, there is[10] *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 descendant 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 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 $k, \ell \geq 0$) are two distinct paths from u to v, we say $p <_{\texttt{dfs}} q$ if there is an m ($1 \leq m < \min\{k, \ell\}$) such that $u_1 = v_1, \ldots, u_m = v_m$ and $u_{m+1} <_{\texttt{dfs}} v_{m+1}$ 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

$$\pi : (u-u_1-\cdots-u_k-v) \tag{11}$$

where each vertex $u_1, \ldots, u_k, v$ 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 (11) 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 v' is a vertex 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, as we cycle through all the vertices adjacent to u, we will eventually reach v and color it `seen` from u, *i.e.*, $u-v$ is an edge of the DFS tree.

INDUCTIVE CASE: Suppose $k > 0$. Let π in (11) 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.**

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

[10] If we use the white-black coloring scheme, this may be called the "white path" as in [4].

- **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. E.g., edges 2−1 and 3−2 in Figure 7(iii).
- **Forward edges**: these are non-tree edges $u{-}v \in E$ where v is a descendant of u. E.g., edges 1−6 and 5−6 in Figure 7(iii).
- **Cross edges**: these are non-tree edges $u{-}v$ for which u and v are not related by ancestor/descendant relation. E.g., edges 4−6, 3−6 and 4−3 in Figure 7(iii).
- **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. Such edges cannot arise if we use a DFS Driver, which ensures that every edge will be seen.

Suppose we call the DFS Driver of a graph. Each call to DFS by the Driver produces a DFS tree. Thus the set of tree edges form a **DFS forest**. How does the above edge classification change?

- First of all, there are no unseen edges.
- We have a new kind of edge, **cross tree edge** that goes from one DFS tree to another.

¶30. Application of Unseen Path Lemma. We use the Unseen Path Lemma to prove that the existence of back edges in a DFS tree is equivalent to graphs being cyclic:

Lemma 7. *Consider the DFS forest of a digraph G:*
(i) If $u{-}v$ is a back edge in this forest then G has a unique simple cycle containing $u{-}v$.
(ii) If Z is a simple cycle of G then one of the edges of Z is a back edge in the DFS forest.

Proof. (i) is clear: given the back edge $u{-}v$, we construct the unique cycle comprising the path in the DFS forest from v to u, plus $u{-}v$. (ii) Conversely, for any simple cycle $Z = [v_1, v_2, \ldots, v_k]$, in the running of the DFS Driver program on G, there is a first instant when we see a vertex in Z. Wlog, let it be v_1. At this instant, there is an unseen path from v_1 to v_i (for $i = 2, \ldots, k$). By the Unseen Path Lemma, each v_i will become a descendant of v_1 in the DFS forest. Pick the i such that v_i be the last vertex to be seen. It will be Clearly, $v_i{-}v_{i+1}$ is a back edge in the forest. **Q.E.D.**

This lemma shows that detecting cycles in graphs can be reduced to detecting back edges. Detecting back edges is a special case of the general *computational* problem of classifying all the edges in a DFS forest. The next section will address this problem in its full generality; for now, we consider the simpler case of classifying edges when the graph is a bigraph.

¶31. Computational classification of bigraph edges by DFS. When DFS is applied to bigraphs, we can treat the bigraph as a special type of digraph. As usual we view a bigraph G as a digraph G' whose directed edges come in pairs: $u{-}v$ and $v{-}u$, one pair for each undirected edge $\{u, v\}$ of G. We call $u{-}v$ and $v{-}u$ **partners** of each other. So the above classification (¶29) is immediately applicable to these directed edges. This classification has special properties which are relatively easy to see (Exercise):

Lemma 8. *Let $u-v$ be an edge of G'.*
(a) $u-v$ is never a cross edge.
(b) $u-v$ is a back edge if and only if there is a path in the DFS tree from v to u. Thus, $u-v$ is a back edge iff the partner $v-u$ is either a tree edge or a forward edge.
(c) An edge $u-v$ is unseen iff its partner $v-u$ is unseen.

Using this lemma, we simplify the edge classification of bigraphs. After running DFS on G, we convert G into as a "hybrid graph" G' that has *both* directed and undirected edges. A seen edge of G can be a tree edge, a back edge or a forward edge. But there is redundancy here. We propose to omit one partner in each pair of partners so that no information is lost:

- The undirected edges of G' are precisely the unseen edges of G.
- If $u-v$ is a tree edge, we omit its partner $v-u$. In this way, the undirected edge $\{u, v\}$ becomes a directed edge (u, v).
- If $u-v$ is a forward edge, we omit it but retain its partner $v-u$ as a backward edge.

Thus, the hybrid graph G' just have three kinds of edges:

$$\textit{tree, back, unseen.} \tag{12}$$

When we speak of the DFS classification of the edges of a bigraph G, we are really referring to this classfication of G'.

Now that we know what the classification is, we address its computational problem. As usual, we encode tree edges by introducing a **parent array** $p[v \in V]$ where $p[v]$ is the parent of v in the DFS tree. Thus tree edges are precisely of the form $p[v]-v$. The root is the unique node v with the property $p[v] = v$.

In the shell of ¶26, we can detect the forward/back during $\boxed{\text{PREVISIT}(v, u)}$ (Line 10). There are two possibilities: if v is `unseen` then edge $u-v$ is a tree edge. Otherwise, it must be a back or forward (recall there are no cross edges). But we cannot distinguish between back and forward edges using only the colors!

The solution is to introduce "time". We plan to record the *time when we first see a node.* Then in PREVISIT(v, u), assuming v is `seen`, we know that $u-v$ is a back edge iff v was seen before u. To implement "time", we introduce a global counter `clock` that is initially 0. We introduce an array, `firstTime`$[v : v \in V]$ such that `firstTime`$[v]$ is set to the value of `clock` when we first see v (and the value of `clock` will be incremented). Thus the clock is just counting the number of "significant events". Later we will expand the notion of significant events. These operations are encoded in our macros:

This is no ordinary clock

$$\boxed{\text{INIT}(G, s_0)} : \texttt{clock} \leftarrow 0$$

```
PREVISIT(v, u) :
    If v is unseen,
        firstTime[v] ← clock++
        p[v] ← u    ◁ so p[v]−v is "tree-edge"
    elif (firstTime[u] > firstTime[v])
        u−v is "back edge"
    else
        u−v is "forward edge"
```

The above classification is evidently correct if we use the Standard (recursive) DFS in the next section. But this is not so clear in the non-recursive DGS.

¶32. Biconnectivity. When we introduced reduced graph earlier ¶10, we said that it is not a useful concept for bigraphs. We now introduce the appropriate analogue for bigraphs.

Let $G = (V, E)$ be a bigraph. A non-empty 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 vertices in C that contains u and v. Note we require the cycle be simple, in contrast to the definition of connectivity in ¶10. For instance, if there is an edge u−v, then $\{u, v\}$ is a biconnected set. That is because a closed path of the form u−v−u is considered a simple closed path; so its equivalence class $[u$−$v]$ is considered[11] a simple cycle. Any singleton $\{u\}$ is also a biconnected set, for trivial reasons (or by definition). If C is a biconnected set that is maximal with respect to biconnectedness, then we call C a **biconnected component**. If G has only one biconnected component, then G is called a **biconnected graph**. Biconnected components of sizes 1 or 2 are **trivial**. The lone vertex in a trivial biconnected component of size 1 is called an **isolated vertex**; the single edge in a trivial biconnected component of size 2 is called a **bridge**.

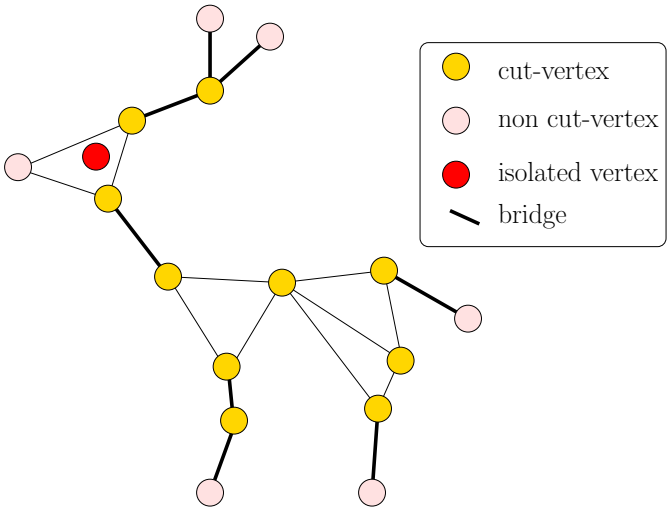


Figure 8: Reindeer graph with 3 non-trivial biconnected components and 8 bridges

E.g., the bigraph in Figure 3(a) has two biconnected components, $\{a, b, c, d\}$ and $\{d, e\}$. The vertex d is common to these biconnected components. Moreover, $\{a, b, c\}$ and $\{b, c, d\}$ are

[11] To some extent, $[u$−$v]$ is considered "simple" only by a technicality.

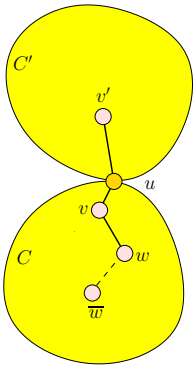
biconnected sets but $\{a, b, d\}$ is not. A more interesting graph is the reindeer graph in Figure 8 with 12 biconnected components, of which 3 are non-trivial.

Biconnectivity is clearly a strong notion of connectivity. Two biconnected components can share at most one common vertex, and such vertices are called **cut-vertices** (or "articulation points"). We give an alternative characterization of cut-vertices using connectivity instead of biconnectivity: vertex u is a cut-vertex iff the removal of u, and also all edges incident on u, will increase the number of connected components of resulting bigraph. This means there exist two vertices v, v' (both different from u) such that all paths from v to v' must pass through u. The absence of cut-vertices is *almost* equivalent to biconnectivity, as seen is the following easily verified facts:

Lemma 9.
(a) If G has a cut-vertex, then it is not biconnected.
(b) If G has no cut-vertices, and is connected, then it is biconnected.

The bridge is the edge analogue of cut-vertex: a bridge is an edge $u{-}v$ whose removal will increase the number of connected components of the resulting bigraph; note that the vertices u, v remain in the graph in this definition. 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 $c{-}e$; the graph in Figure 8 has 8 bridges and 9 cut-vertices.

Lemma 10 (Computational Characterization of Cut Vertex). *Assume G is connected, and T is a DFS tree of G. A vertex u in T is a cut-vertex iff one of the following conditions hold:*
(i) If u is the root of T and has two or more children.
(ii) If u is not the root, but it has a child v such that for any descendant w of v, if $\overline{w}$ is adjacent to w, then $\overline{w}$ is also a descendant of u. Note that the vertex w may be equal to v.



cut vertex u

Proof. Condition (i) implies u is a cut-vertex because there are no cross edges. Condition (ii) implies any path from the parent of u to w must pass through u, but there is no path from w to any ancestor of u; thus u is a cut-vertex.

Conversely, suppose u is a cut-vertex. Let C, C' be two distinct biconnected components containing u. If u is the root, then u must have a child $v \in C$ and a child $v' \in C'$. Thus u has more than one child, i.e., property (i) holds. Hence assume u is not the root. Then u has a parent v' and wlog, we may let C' denote the biconnected component of v'. Also one of the children v of u must belong to the other component C. Suppose there exists a descendant w of v such that w is adjacent to some vertex $\overline{w}$. where $\overline{w}$ not a descendant of u. Since the BFS tree has no cross edges, $\overline{w}$ must be an ancestor of u. So there is a path in the BFS tree from $\overline{w}$ to w. This path, together with the edge $w{-}\overline{w}$ forms a cycle Z that passes through v' and v. This contradicts the assumption that C, C' are distinct biconnected components. Thus, property (ii) holds. **Q.E.D.**

¶33. Biconnectivity Detection. We now present an algorithm to detect if a bigraph G is biconnected. Our algorithm either outputs a cut-vertex of G or report that G is biconnected. It is based on detecting any one of the two conditions in Lemma 10. Detecting condition (i) is easy: we introduce a global variable `numChildren` count the number of children of the root. Initially, `numChildren` $\leftarrow 0$. and each time the root spawns a new child, we update this number. If this number exceeds 1, we report the root to be a cut-vertex. To detect condition (ii), let `mft`$[u]$ denote the minimum value of `firstTime`$[w]$ where w ranges over the set $B(u)$ of vertices

for which there exists a back edge of the form $v{-}w$ and v is a descendent of u:

$$B(u) := \{w : (\exists v, \text{descendent of } u)[(v{-}w) \text{ is a back edge}\}\,.$$

Note that v need not be a proper descendant of u (i.e., we allow $v = u$). As usual, the minimum over an empty set is ∞, so $\texttt{mft}[u] = \infty$ iff $B(u)$ is empty. We now address three questions:

- What is the significance of $\texttt{mft}[u]$? Suppose u is not the root of the DFS tree. Claim: *u is a cut-vertex iff there exists a child v of u such that* $\texttt{mft}[v] \geq \texttt{firstTime}[u]$. In proof, if u is a cut-vertex, then condition (ii) provides a child v of u such that $\texttt{mft}[v] \geq \texttt{firstTime}[u]$. Conversely, suppose $\texttt{mft}[v] \geq \texttt{firstTime}[u]$. Take any path from v to $p[u]$. There is a first edge $w{-}w'$ in this path such that $\texttt{firstTime}[w] \geq \texttt{firstTime}[u] > \texttt{firstTime}[w']$. We claim that $w = u$. If not, then w is a descendant of v, and $w{-}w'$ is a back edge and so $w' \in B(v)$. Thus $\texttt{mft}[v] \leq \texttt{firstTime}[w'] < \texttt{firstTime}[u]$, contradiction. Thus every path connecting v and $p[u]$ must pass through u, i.e., u is a cut-vertex. This proves our claim.
- How do we maintain $\texttt{mft}[u]$? We initialize $\texttt{mft}[u]$ to ∞ when u is first seen. We subsequently update $\texttt{mft}[u]$ in two ways:
  (i) When we detected a back edge $u{-}v$, we will update $\texttt{mft}[u]$ with $\min\{\texttt{mft}[u], \texttt{firstTime}[v]\}$.
  (ii) When we POSTVISIT(v), and $p[v] = u$, we can update $\texttt{mft}[u]$ to $\min\{\texttt{mft}[u], \texttt{mft}[v]\}$.
  By the time we have POSTVISIT u, the value of $\texttt{mft}[u]$ would have been correctly computed because, inductively, it has been updated with the contributions of each of its children in the DFT, and also the contributions of each back edge originating from u.
- How do we use $\texttt{mft}[u]$ computationally? We can only use it to detect cut-vertices: in POSTVISIT(u), we check if $\texttt{mft}[u] \geq \texttt{firstTime}[p[u]]$, and $p[u]$ is not the root.

To summarize the algorithm, here are the shell macros:

```
INIT(G, s_0) :
    clock ← 0
    p[s_0] = s_0
    numChildren = 0
```

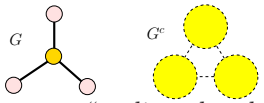
```
PREVISIT(v, u) :
    If (v is unseen),
        firstTime[v] ← clock++
        p[v] ← u                          ◁ (u−v) is "tree-edge"
        mft[v] ← ∞
        If p[u] = u                       ◁ u is root
            If (++numChildren > 1)
                Return("u is cut-vertex")
    elif (firstTime[u] > firstTime[v])    ◁ u−v is "back edge"
        mft[u] ← min {mft[u], firstTime[v]}
```

```
POSTVISIT(u) :
    If (mft[u] ≥ firstTime[p[u]]) and (p[u] ≠ p[p[u]])
        Return("p[u] is a cut-vertex")
```

"G^c" is a recycled notation for the "reduced graph" of a digraph G.

¶34. Reduced Bigraphs. Given a bigraph G, we define a bigraph $G^c = (V^c, E^c)$ that will be called the **reduced graph** of G. Intuitively, the elements of V^c should be the biconnected components of G, and the edges should comprise $C-C'$ whenever $C \cap C'$ is non-empty. The problem is this "G^c" could be cyclic shown in the margin. We prefer a definition where G^c is acyclic, in analogy to the digraph case.

If we only restrict V^c to non-trivial components, the result is acyclic, but we have have lost information about connectivity. Here then is our final definition: first, define a vertex to be **essential** if it does not belong to a non-trivial component. For instance, the reindeer graph Figure 8 has 7 essential vertices. Let V^c comprise all essential vertices as well as nontrivial components. Let E^c comprise $C-C'$ if either $C \cap C' \neq \emptyset$ or there is an edge connecting a vertex in C to one in C'. For instance, if C is an essential vertex, and C' is a non-trivial component, then $C-C'$ iff there is a cut-vertex v in C' such that $\{v, C\}$ is a bridge. We call G^c the **reduced graph** for G, as illustrated in Figure 9. In the Exercise, we ask you to extend the above biconnectivity detection algorithm to compute some representation of G^c.



"cyclic reduced bigraph"

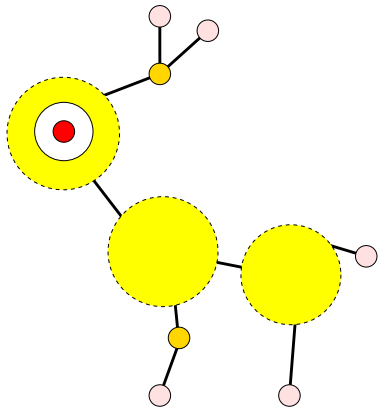


Figure 9: Reduced Reindeer Graph

Exercises

Exercise 5.1: True or False: Let C be a connected component C of a bigraph. Then C is a biconnected component iff C does not contain a cut-vertex or a bridge. ♢

Exercise 5.2:
(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 5.3: 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 sometimes 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 5.4: 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 5.5: 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 DFS classification of edges of a digraph is complete. Recall that each edge is classified as either tree, back, forward, cross, unseen. ♢

Exercise 5.7: Prove Lemma 8 concerning the DFS classification of the directed edges of a bigraph. ♢

Exercise 5.8: In the text, we gave an algorithm to detect if a bigraph is biconnected. Generalize this algorithm to compute all the biconnected components of the bigraph. ♢

Exercise 5.9: Extend our biconnected graph detection algorithm to compute a representation of the reduced graph G^c of a bigraph G. How should you represent G^c? First, we want to identify each vertex v as a cut-vertex or not. This can be represented by a Boolean array $CV[v \in V]$ where $CV[v] = \mathit{true}$ iff v is a cut-vertex. Second, we want to assign to each edge of G an integer called its "component number" (two edges have the same component number iff they belong to the same biconnected component of G). ♢

Exercise 5.10: 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 (link-distance) 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 radius$(G) \leq$ diameter$(G) \leq 2$radius(G).
(b) To see that part(a) is the best possible, show that for every natural number n, there are graphs G_n and H_n such that $n =$ radius$(G_n) =$ diameter(G_n) and diameter$(H_n) = n$ and radius$(H_n) = \lceil n/2 \rceil$.
(c) Show[12] that radius$(G) = \lceil$diameter$(G)/2 \rceil$ if G is acyclic. This shows that the graphs

[12] This was observed by Zhao (Joan) Jin in Fall 2011.

G_n in part(b) must be cyclic.
(d) Suppose G is a connected acyclic bigraph. Give an efficient algorithm to compute the diameter of G using the DFS shell. Prove the correctness of your algorithm. What is the complexity of your algorithm? HINT: write down a recursive formula for the diameter of a tree in terms of the diameter *and* height of its subtrees.
(e) How would you compute the radius of an acyclic connected bigraph? ♢

Exercise 5.11: Re-do the previous question (part (c)) to compute the diameter, but instead of using DGS, use BFS. ♢

Exercise 5.12: Prove that our nonrecursive DFS algorithm is equivalent to the recursive version. ♢

Exercise 5.13: Suppose we simply replace the queue data structure of BFS by the stack data structure. Do we get the DFS? Here is result, 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       INIT(G, s_0)    ◁ If standalone, make all vertices unseen except for s_0
    ▷ Main Loop:
        while S ≠ ∅ do
2           u ← 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. Which problems can still be solved by BDFS? Is there any conceivable advantage of DBFS? ♢

End Exercises

§6. Standard Depth First Search

¶35. Recursive DFS. The Nonrecursive DFS is simplified when formulated as a recursive algorithm. The simplification comes from the fact that the explicit stack is now hidden as the recursive stack. Indeed, this is the "standard" presentation of DFS:

```
Standard DFS
    Input:   G = (V, E; s_0) a graph (bi- or di-)
             The vertices in V are colored unseen or done; s_0 is unseen.
    Output   Application dependent
1       Color s_0 as seen, and [VISIT(s_0)]
2       for each v adjacent to s_0 do
3           [PREVISIT(v, s_0)]
4           if (v is unseen) then
6               Standard DFS((V, E; v))    ◁ Recursive call
7       Color s_0 done, and [POSTVISIT(s_0)].
```

Like the Nonrecursive DFS, the main macros in the Standard DFS Shell are PREVISIT, VISIT and POSTVISIT. But in contrast to the Nonrecursive DFS Shell, the Standard DFS does not have INIT or CLEANUP. This means the Standard DFS cannot be a stand-alone shell – it must be called by some driver function to initialize all vertices to `unseen`, and to call some CLEANUP macro to perform any post-processing of the graph. To visit every vertex of a digraph, we can keep the DFS Driver in ¶27, except that each DFS call refers to the Standard DFS.

Our placements of the [POSTVISIT](s_0) macro in Line 7 is intended to allow you to visit all the vertices adjacent to s_0 once more. This violates our normal injunction against non-constant work macros (see ¶16). Of course, this means doing another iteration of the loop of Line 2. That injunction is now modified to mean that, for each v adjacent to s_0, you should do $O(1)$ work in the POSTVISIT macro.

¶36. Computational classification of digraph edges by DFS. The result of calling this driver on G is the production of a DFS forest that spans G and a classification of every edge of G. But this classification is only conceptual so far — the purpose of this section is to achieve a computational classification of these edges. Previously we have only achieved this for the edges of a bigraph. Indeed, we can extend the solution method used for bigraphs: recall that we had time stamps and we maintained an array `firstTime`$[v \in V]$. We now introduce another array `lastTime`$[v \in V]$ to record the time of POSTVISIT'ing vertices.

Assume that `firstTime`$[v]$ and `lastTime`$[v]$ are both initialized to -1 in DRIVER_INIT(G). It is possible to avoid initialization of these arrays. That is because the color scheme `unseen`/`seen`/`done` can serve to detect initialization conditions. We will discuss this later.

In the Standard DFS, unlike the nonrecursive version, there is no INIT(G) step — that is because we do not want to initialize with each recursive call! Also, we perform VISIT(v) (Line 1) at the beginning of the recursive call to v (Line 6), but first ensuring that v is `unseen`. Finally, after recursively VISITing all the children of s_0, we POSTVISIT(s_0) (Line 7). This is done in a much smoother way than in the Nonrecursive DFS. Here are some macro definitions: we will maintain the usual parent array $p[v \in V]$.

- DRIVER_INIT$(G)\equiv$ `clock` $\leftarrow 0$; (for $v \in V$)[`firstTime`$[v] \leftarrow$ `lastTime`$[v] \leftarrow -1$].
- PREVISIT$(v, u)\equiv$ If v is `unseen`, then `firstTime`$[v] \leftarrow$ `clock++` and $p[v] \leftarrow u$.
- POSTVISIT$(v)\equiv$ `lastTime`$[v] \leftarrow$ `clock++`.

During the computation, a node v is `unseen` if `firstTime`$[v] < 0$; it is `seen` if `firstTime`$[v] >$ `lastTime`$[v]$; it is `done` if `firstTime`$[v] <$ `lastTime`$[v]$. In other words, we can avoid maintaining colors explicitly if we have the arrays `firstTime` and `lastTime`.

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 return to the computational classification of the edges of a digraph G relative to a DFS forest on G:

Lemma 11. *Assume that a digraph G has been searched using the DFS Driver, resulting in a complete classification of each edge of G. Let u−v be an edge of G.*

1. *u−v is a back edge iff* `active`$(u) \subset$ `active`(v).
2. *u−v is a cross edge iff* `active`$(v) <$ `active`(u).
3. *u−v is a tree edge iff $p[v] = u$.*
4. *u−v is a forward edge iff* fl$(p[v]) \subset$ fl(u).

This above classification of edges by active ranges is illustrated in Figure 10.

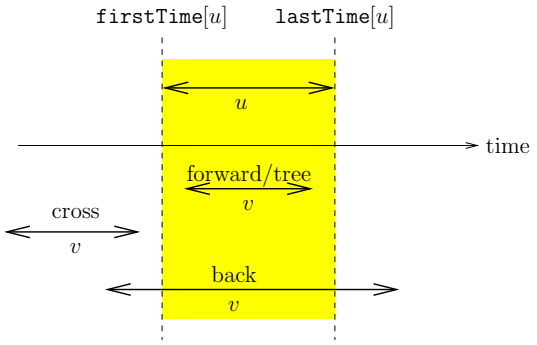


Figure 10: Relative positions of active ranges of u, v and the classification of edge $(u$−$v)$

These criteria can be used by the PREVISIT(v, u) macro to classify edges of G. Note that it is an elaboration of the previous PREVISIT:

```
PREVISIT(v, u)
▷ Visiting v, from u
    if (firstTime[v] = −1),
        mark u−v as "tree edge" (i.e., p[v] ← u)
        firstTime[v] ← clock++
    elif (firstTime[v] < lastTime[v]),
        mark u−v as "forward edge"
    elif (lastTime[v] = −1),
        mark u−v as "back edge"
    else
        mark u−v as "cross edge".
```

The correctness of this classification is a direct consequence of Lemma 11 (cf. Figure 10). If the arrays `firstTime`, `lastTime` are not initialized, we could replace the above code as follows: instead of the test `firstTime`$[v] = -1$, we could check if "v is `unseen`". Instead of the test `lastTime`$[v] = -1$, we could check if "v is `seen`" (thus not yet `done`).

¶37. Application of cycle detection. Cycle detection is a basic task in many applications. In operating systems, we have **processes** and **resources**: a process can **request** a resource, and the operating system can **grant** that request. We also say that the process has **acquired** the resource after it has been granted. Finally, a process can **release** a resource that it has acquired.

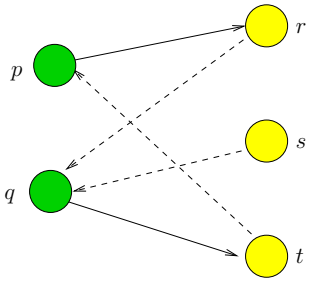


Figure 11: Process-resource Graph: $P = \{p, q\}, R = \{r, s, t\}$.

Let P be the set of processes and R the set of resources. We introduce a bipartite graph $G = (P, R, E)$ where $V = P \uplus R$ is the vertex set and $E \subseteq (P \times R) \cup (R \times P)$. See Figure 11 for an example with 2 processes and 3 resources. An edge $(p, r) \in E \cap P \times R$ means that process p has requested resource r but it has not yet been granted. An edge $(r, p) \in E \cap R \times P$ means r has been granted to p (subsequent to a request). A process p can also release any resource r it has acquired. While requests and releases are made by processes, the granting of resources to processes is made by the operating system. It is clear from this description that we view G as a dynamic graph where edges appear and disappear over time. Specifically, a process p can create a new edge of the form (p, r) or remove edges of the form (r, p); the operating system can transform an edge of the form (p, r) to (r, p). In operating systems (Holt 1971), G is called a **process-resource graph**.

Let us make some additional assumptions about how processes operates. As processes are executed, they issue requests on a set of one or more resources. For instance, to print a file, a process may need to acquire two resources, a file queue and a printer. We assume that the process will be blocked until each one of these requests it has required each of these resources. Sometime after it has acquired all the resources, the process will release *all* the acquired resources. The graph G is thus an instantaneous snapshot of the set of requests that are pending (p, r) or granted (r', p'). Under these assumptions, G represents a **deadlock** if there is a cycle $[p_1, r_1, p_2, r_2, \ldots, p_k, r_k]$ in G ($k \geq 2$) where p_i requests r_i but r_i has been granted to p_{i+1}. In particular, r_k has been granted to $p_{k+1} = p_1$. For instance, the graph in Figure 11 has a deadlock because of the cycle $[p, r, q, t]$. In this situation, the processes $p_1, \ldots, p_k$ could not make any progress. Thus our cycle detection algorithm can be used to detect this situation.

Exercises

Exercise 6.1: 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 6.2: In what sense is the Nonrecursive DFS (¶26) and the Standard DFS equivalent?
♢

Exercise 6.3: 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 cyclic 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 6.4: Give an efficient algorithm to detect a deadlock in the process-resource graph.
♢

Exercise 6.5: 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** $e = p_1 p_2 p_3 \ldots$ ($p_i \in V_P$) 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 6.6: 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).

¶38. Topological Sort. One motivation is the so-called[13] 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.

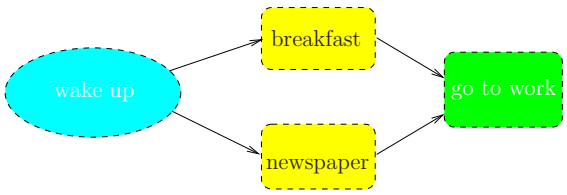


Figure 12: PERT graph

[13] 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.

Let

$$(v_1, v_2, \ldots, v_n) \tag{13}$$

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 interpreted 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 (13). Hence, topological sort amounts to computing this rank attribute of each vertex. We introduce an array `Rank`$[v \in V]$ for this purpose. Thus the goal of topological sort amounts to a "ranking algorithm" which fills in this array.

E.g., If our topological sort is the sequence (v_3, v_1, v_2, v_4), the corresponding rank array is `Rank`$[v_1, v_2, v_3, v_4] = [2, 3, 1, 4]$.

We use the DFS Driver to compute the rank attribute array. We must initialize the `Rank` array in the DRIVER_INIT.

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

Indeed, we need not use a separate color array, but simply interpret a `Rank` of -1 as `unseen`. How can we use DFS to assign a ranks to the vertices? If we reach a leaf v of the DFS tree, then we can clearly assign it with the largest available rank (initially, the largest available rank is n). To support this, we introduce a global counter R that is initialized to n. Each time a vertex v is to receive a rank, we use the current value of R, and then decrement R, thus:

$$\texttt{Rank}[v] \leftarrow R\texttt{--}. \tag{14}$$

Inductively, if all the proper descendants of v have received ranks, we can assign a rank to v. If all ranks are assigned as in (14), then it will be clear that the rank of v is less than the ranks of its descendants, which is what we want in the topological sort. Moreover, it is clear that the rank assignment (14) should be performed in POSTVISIT(v). Note that the rank function is just as the order of v according to `lastTime`$[v]$. So we could also perform (14) when we update the `lastTime` array.

It is easy to prove the correctness of this ranking procedure, provided the input graph is a DAG. What if G is not a DAG? There are two responses.

- First, we could say that our ranking algorithm should detect the situation when the input digraph G is not a DAG. This amounts to detecting the existence of back edges. When a back edge is detected, we abort and output "no topological sort".
- Second, it might turn out that the output of our ranking algorithm is still useful for a non-DAG. Indeed, this will be the case in our strong component algorithm below. For the strong component algorithm, it is more convenient to compute the **inverse** of `Rank`, i.e., an array `iRank`$[1..n]$ such that

  $$\texttt{iRank}[i] = v \quad \Longleftrightarrow \quad \texttt{Rank}[v] = i \tag{15}$$

  Thus we just have to replace (14) by

  $$\texttt{iRank}[R\texttt{--}] \leftarrow v. \tag{16}$$

  The topological sort (13) is then given by

  $$(\texttt{iRank}[1], \texttt{iRank}[2], \ldots, \texttt{iRank}[n]).$$

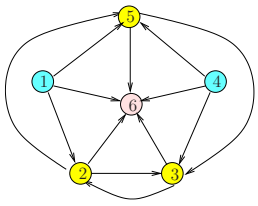
¶39. Strong Components. Computing the components of digraphs is somewhat more subtle than the corresponding problem of biconnected components for bigraphs. There are at least three distinct algorithms known for this problem. Here, we will develop the version based on "reverse graph search".

Recall that connected components of a digraph are also called "strong components". The strong components forms a partition of the vertex set; this is in contrast to biconnected components that may intersect at cut-vertices.

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

```
STRONG_COMPONENT_DRIVER(G, Per)
    INPUT: Digraph G and permutation Per[1..n].
    OUTPUT: DFS Spanning Forest of G.
▷ Initialization
1       For v = 1, ..., n,
            color[v] ←unseen.
▷ Main Loop
2       For v = 1, ..., n,
3           If (color[Per[v]] =unseen)
4               DFS(Per[v])    ◁ Outputs a DFS Tree rooted at v
```

This program is like the usual DFS Driver program, except that we use $\texttt{Per}[i]$ to determine the choice of the next vertex to visit. We assume that $DFS(i)$ will (1) change the color of every vertex that it visits, from `unseen` to `done`, 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.



digraph G_6

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

$$\begin{aligned} \texttt{Per}[1,2,3,4,5,6] &= [6,3,5,2,1,4] \\ &= [v_6, v_3, v_5, v_2, v_1, v_4]. \end{aligned} \tag{17}$$

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. On the other hand, if we use the "identity" permutation,

$$\texttt{Per}[1,2,3,4,5,6] = [1,2,3,4,5,6], \tag{18}$$

our STRONG_COMPONENT_DRIVER will first call to $DFS_1(\texttt{Per}[1])$. This produces a DFS tree containing the vertices $\{v_1, v_2, v_3, v_5, v_6\}$. Only one vertex 4 remain `unseen`, and so the driver will next call $DFS_1(\texttt{Per}[4])$ which produces a DFS tree containing $\{v_4\}$. Thus, the identity permutation does not lead to the correct output for strong components.

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 $\texttt{Per}[1..n]$ is said to be **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'*.

Clearly, 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 some form 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 $\texttt{Per}[1..n]$ (the inverse of the $\texttt{Rank}[1..n]$). The next lemma shows that $\texttt{Per}[1..n]$ almost has the properties we want. For any set $C \subseteq V$, we first define

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

Lemma 12. *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 $\texttt{Rank}[u_0] = \texttt{Rank}[C]$.
(b) If there is path from C to C' in the reduced graph of G, then $\texttt{Rank}[C] < \texttt{Rank}[C']$.

Proof. (a) By the Unseen Path Lemma, every node $v \in C$ will be a descendant of u_0 in the DFS tree. Hence, $\texttt{Rank}[u_0] \leq \texttt{Rank}[v]$, and the result follows since $\texttt{Rank}[C] = \min\{\texttt{Rank}[v] : v \in C\}$.
(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), $\texttt{Rank}[C] = \texttt{Rank}[u_0]$. Since there is a path from C to C', an application of the Unseen Path Lemma says that every vertex in C' will be descendants of u_0. Let u_1 be the first vertex of C' that is seen. Since u_1 is a descendant of u_0, $\texttt{Rank}[u_0] < \texttt{Rank}[u_1]$. By part(a), $\texttt{Rank}[u_1] = \texttt{Rank}[C']$. Thus $\texttt{Rank}[C] < \texttt{Rank}[C']$. (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, $\texttt{Rank}[C_0] < \texttt{Rank}[u_0]$. But $\texttt{Rank}[u_0] = \texttt{Rank}[C']$. **Q.E.D.**

Is the reverse "topological sort" ordering

$$[\texttt{iRank}[n], \texttt{iRank}[n-1], \ldots, \texttt{iRank}[1]] \tag{19}$$

is a good permutation?
Suppose there is path from strong component C to strong component C'. Then our lemma tells us that the rank of the *first seen vertex* v of C is less than the rank of the *first seen vertex* v' of C'. So v appears *after* v' in the list (19).
Unfortunately, what we need for a good ordering is that the *last seen vertex* u of C should appear after the *last seen vertex* u' of C' in (19). Why? Because u (and not v) is the first vertex of C to appear in the list (19).

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 13. *If $\texttt{iRank}[1..n]$ is the result of running topological sort on G^{rev} then $\texttt{iRank}$ 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 (19). That means that the first vertex of C is listed after the first vertex of C' in the listing $[\texttt{iRank}[1], \texttt{iRank}[2], \ldots, \texttt{iRank}[n]]$. This is good. **Q.E.D.**

We now have the complete algorithm:

```
Strong_Component_Algorithm(G)
    Input: Digraph G = (V, E), V = {1, 2, ..., 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 iRank[1..n].
3.      Call Strong_Component_Driver(G, iRank)
```

Remarks. Tarjan [7] gave the first 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 [6].

Exercises

Exercise 7.1: (a) Provide a self-contained algorithm (containing all the macros filled-in) to compute inverse Rank array $\texttt{iRank}[1..n]$.
(b) Code up this program in your favorite programming language. ♢

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 cyclic 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, ..., 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, ..., ⌊n/2⌋, do the swap Per[i] ↔ 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 descendants 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 descendant 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 descendants 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

§8. Games on Graphs

How do we know if a computer program has a given property? In industrial-strength software, especially in mission-critical applications, we seek strong assurances of certain properties. The controller for a rocket is such a mission-critical software. The area of computer science dealing with such questions is called **program verification**. We can use a graph to model salient properties of a program: the vertices represent **states** of the program, and edges represent possible **transitions** between states. Properties of the program is thereby transformed into graph properties. Here are two basic properties in verification:

- **Reachability** asks whether, starting from initial states from some A, we can reach some states in some set B. For example, if B is the set of terminal states, this amounts to the question of halting becomes a reachability question. Sometimes the property we seek is **non-reachability**: for instance, if C is the set of "forbidden states", then we want the states in C to be non-reachable from the initial states. Of course, in this simple form, DFS and BFS can check the reachability or non-reachability properties.
- **Fairness** asks if we can reach any state in some given set B infinitely often. Suppose the program is an operating system. If the states in B represent running a particular process, then we see why this property is regarded as "fairness" (no process is shut out by the process scheduler). Again, if state B represents the servicing of a print job at the printer queue, then fairness implies that the print job will eventually complete (assuming some minimum finite progress).

Alice and Bob

We introduce a new twist in the above reachability and fairness questions by introducing two opposing players, let us call them Alice and Bob. Alice represents a program, and is responsible for some transitions in the graph. Bob represents the external influences (sometimes called "nature") that determines other transitions in the graph. For instance, in our above example, Alice might send us into the state q which represents the servicing of a printer queue. But the transitions out of q might take us to states representing finished job, out-of-paper, paper jam, etc. It is Bob, not Alice, who determines these transitions.

¶40. Game Graphs. To model this, we introduce the concept of a **game graph** $G = (V_A, V_B, E)$ where $V_A \cap V_B = \emptyset$ and $(V_A \cup V_B, E)$ is a digraph in the usual sense. Note that G is not necessarily a bipartite graph — we do not assume $E \subseteq (V_A \times V_B) \cup (V_B \times V_A)$. The intuitive idea is that each $v \in V_A$ ($v \in V_B$) represents a state whose next transition is determined by Alice (Bob). A particular path through this graph $(v_1, v_2, \ldots)$ represents a run of the program, with the transition v_i–v_{i+1} determined by Alice (Bob) iff $v_i \in V_A$ ($v_i \in V_B$). We might think of the original (single player) reachability/fairness problems as operating in a graph in which V_B is the empty set. Clearly, the introduction of Bob captures new realities of an operating system. Reachability/Fairness is now defined to mean "reachable/fair in spite of Bob".

We next introduce a "game" on $G = (V_A, V_B, E)$ played by Alice and Bob (called the "players"). Let $V = V_A \cup V_B$, and for $v \in V$, let $Out(v) = \{u \in V : v\text{–}u \in E\}$ and $In(v) = \{u \in V : u\text{–}v \in E\}$. The elements of V are also called **states**. A **terminal state** is v such that $Out(v) = \emptyset$. There is a single token that resides at some state of V. At each step, this token is moved from its current state v to some new state $u \in Out(v)$. This move is determined by

Alice (Bob) if $v \in V_A$ ($v \in V_B$). In general, the moves of A or B can be non-deterministic, but for our basic questions, we may assume them to be deterministic. That is, the moves of Player X ($X \in \{A, B\}$) is determined by a function $\pi_X : V_X \to V$ such that $\pi_X(v) \in Out(v)$ We call π_X the **strategy** for Player X (X-strategy for short). Typically, we let α denote an A-strategy, and β denote a B-strategy. A **complete strategy** is a pair (α, β), which can be succinctly represented by a single function $\pi : V \to V$. From any $v_1 \in V$, the pair $\pi = (\alpha, \beta)$ determines a maximal path $(v_1, v_2, \ldots)$ where $v_{i+1} = \pi(v_i)$. This path is either finite (in which case the last state is terminal) or infinite. We may denote the path by $\omega(v_1, \alpha, \beta)$ or $\omega(v_1, \pi)$, and call it a **play**. Let $\Omega = \Omega(G)$ denote the set of all plays, ranging over all complete strategies and all initial states. We write "$u \in \omega$" to mean u occurs in the play ω. Also "$u \in_\infty \omega$" if u occurs infinitely often in ω (this implies ω is infinite). We may now define:

- Intuitively, Force(u) is the set of states from which Alice can force the system into state u. Formally:

$$\text{Force}(u) := \{v \in V : (\exists \alpha)(\forall \beta)[u \in \omega(v, \alpha, \beta)]\}.$$

- Intuitively, Fair(u) is the set of states from which Alice can force the system to enter state u infinitely often. Formally:

$$\text{Fair}(u) := \{v \in V : (\exists \alpha)(\forall \beta)[u \in_\infty \omega(v, \alpha, \beta)]\}.$$

For $U \subseteq V$, let $\text{Force}(U) = \cup_{u \in U} \text{Force}(u)$ and $\text{Fair}(U) = \cup_{u \in U} \text{Fair}(u)$. The set Fair($U$) is also called the **winning states** for a Büchi game with Büchi objective U. Such games originated in mathematical logic. We will design algorithms to compute the sets Force(U) and Fair(U) in times $O(n+m)$ and $O(mn)$. The exercises[14] will show how Fair(U) can be computed in $O(n^2)$ time.

¶41. Least Fixed Points (LFP). Inherent in these concepts is the important computing concept of least fixed points (LFP). Let us look at the basic properties of the set Force(U):

- $U \subseteq \text{Force}(U)$
- If $v \in V_A$ and $Out(v) \cap \text{Force}(U) \neq \emptyset$ then $v \in \text{Force}(U)$.
- If $v \in V_B$ and $Out(v) \subseteq \text{Force}(U)$ then $v \in \text{Force}(U)$.

Let us introduce an operator to capture these properties:

$$\mu_G = \mu : 2^V \to 2^V$$

such that for all $U \subseteq V$

$$v \in \mu(U) \Leftrightarrow \begin{cases} v \in U, \text{ or} & \text{[BASIS]} \\ v \in V_A \wedge (Out(v) \cap U \neq \emptyset), \text{ or} & \text{[INDUCT(A)]} \\ v \in V_B \wedge (Out(v) \subseteq U) & \text{[INDUCT(B)]} \end{cases} \tag{20}$$

For any $U \subseteq V$, there is a least $i \geq 0$ such that $\mu^{(i)}(U) = \mu^{(i+1)}(U)$; define $\mu^*(U)$ to be $\mu^{(i)}(U)$. We easily verify:

[14] From Krishnendu Chatterjee and Monika Henzinger (2011).

Lemma 14. *$\mu^*(U)$ is the* **least fixed point** *(LFP) of U under the operator μ):*

- *$\mu^*(U)$ is a fixed point of μ:*
$$\mu(\mu^*(U)) = \mu^*(U)).$$
- *$\mu^*(U)$ is the* least *fixed point of μ that contains U:*
$$(W \supseteq U) \wedge (W = \mu(W)) \quad \Rightarrow \quad \mu^*(U) \subseteq W.$$

Lemma 15. $\text{Force}(U)$ *is the least fixed point of U. In other words,* $\text{Force}(U) = \mu^*(U)$.

Proof. Clearly, $\mu^*(U) \subseteq \text{Force}(U)$. Conversely, suppose $u \in \text{Force}(U)$. By definition, there is a strategy α for Alice such that for all strategies β for Bob, if $\pi = (\alpha, \beta)$ then there exists a $k \geq 1$ such that $\pi^k(u) \in U$. This proves that $u \in \mu^k(U)$. **Q.E.D.**

¶42. Computing $\text{Force}(U)$**.** Given $U \subseteq V_A \cup V_B$, we now develop an algorithm to compute $\mu^*(U)$ in $O(m+n)$ time. It is assumed that the input game graph $G = (V_A, V_B, E)$ has the adjacency list representation. This implies that we can compute the reverse $G^r = (V_A, V_B, E^r)$ of G in time $O(m+n)$, where E^r simply reverses the direction of each edge in E. As we shall see, it is more convenient to use G^r than G.

The basic idea is to maintain a set W. Initially, $W \leftarrow U$ but it will grow monotonically until W is equal to $\mu^*(U)$. For each vertex $v \in V \setminus W$ it is easy to use the conditions in (20) to check whether $v \in \mu(W)$, and if so, add it to W. So the computability of $\mu(W)$ is not in question. But it may be a bit less obvious how to do this efficiently. The critical question is — *in what order should we examine the vertices v or the edges v–w?*

For efficiency, we want to examine edges of the form $(u–w) \in W' \times W$ where $W' = V \setminus W$. If we redirect this edge from what is known (W) to the unknown (W'), we get an w–u of G^r. So we imagine our algorithm as searching the edges of G^r. We maintain a queue Q containing those $w \in W$ for which the edges $Out(w)$ is yet unprocessed. Initially, $Q = U$, and at the end, Q is empty.

You will see that our algorithm is reminiscent of BFS or DFS, searching all graph edges under the control of a queue Q. The difference is that this queue is almost breadth-first, but has a certain built-in priority.

We now set up the main data structure, which is an array $C[1..n]$ of natural numbers. Assuming $V = \{1, \ldots, n\}$, we shall use C to encode the set W under the interpretation $i \in W$ iff $C[i] = 0$. Initially, we have

$$C[i] = \begin{cases} 0 & i \in U \\ 1 & i \in V_A \\ degree(i) & i \in V_B \end{cases}. \tag{21}$$

Here, the degree of vertex i is the number of edges leading out of v in G; it is just the length of the adjacency list of i. Actually, if the degree of i is 0 and $i \notin U$, we should set $C[i] = -1$, to avoid confusing i with an element of W.

It is now clear how we to update this array when processing an edge $(w–u) \in W \times W'$: if $C[u] = 0$, there is nothing to do (u is already in W). Else, we decrement $C[u]$. If $C[u]$ becomes

0 as a result of the decrement, it means u is now a member of W. Note that if $u \in V_A$, then this will happen with the very first decrement of $C[u]$; but if $u \in V_B$, we need to decrement $degree(u)$ times. We need to also take action in case $C[u]$ becomes 0 after decrement: we must now add u to Q. That completes the description of our algorithm, and it is summarized in this pseudo-code:

```
μ*(U):
Input: G^r = (V_A, V_B, E^r) and U ⊆ V = {1,...,n}
Output: Array C[1..n] representing μ*(U)
    ▷ Initialization
        Initialize array C[1..n] as in (21)
        Initialize queue Q ← U
    ▷ Main Loop
        while (Q ≠ ∅)
            w ← Q.pop()
            for each u adjacent to w in G^r
                If (C[u] > 0)
                    C[u]--
                    If C[u] == 0, Q.push(u)
        Return(C)
```

We leave the correctness of this algorithm to the reader. The complexity of this algorithm is $O(m+n)$ because each vertex u is added to Q at most once, and for each $u \in Q$, we process its adjacency list in $O(1)$ time.

¶43. Computing Fair(U). We next use this as a subroutine to compute Fair(U).

References

[1] C. Berge. *Hypergraphs*, volume 445 of *Mathematical Library*. North Holland, 1989.

[2] B. Bollobás. *Extremal Graph Theory*. Academic Press, New York, 1978.

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

[4] 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.

[5] S. Even. *Graph Algorithms*. Computer Science Press, 1979.

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

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