“Liesez Euler, liesez Euler, c’est notre maître à 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 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 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 Borůvka’s (1926) and Jarník (1930) minimum spanning tree algorithm, and Dijkstra’s shortest path algorithm (1959). Tarjan 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; for algorithmic treatments, see Even and Sedgewick.

![Figure 1: The 7 Bridges of Konigsberg](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 \( i \rightarrow j \) represents the

---

1. This former Prussian city is now in Russia, called Kaningsgrad. 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).
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 $v$.

![Figure 2: (a) Political map of 7 countries (b) Their adjacency relationship](image)

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 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 $u-v = v-u$ 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).

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].
4.1. 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
\[ V^k, \quad 2^V, \quad \binom{V}{k} \] denote, respectively, the $k$-fold Cartesian product of $V$, power set of $V$ and the set of $k$-subsets of $V$. The first two notations ($V^k$ 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:
\[
|V^k| = |V|^k, \quad |2^V| = 2^{|V|}, \quad \binom{|V|}{k} = \binom{|V|^k}{k}.
\]

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 $\binom{2}{2} = \binom{|V|^2}{2} = 1$. 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 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 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.

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.
§1. Varieties of Graphs

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.

In Figure 3(a), we display a bigraph $(V, E)$ where $V = \{a, b, c, d, e\}$ and $E = \{a-b, b-c, c-d, d-a, c-e, b-d\}$. In Figure 3(b), we display a digraph $(V, E)$ where $V = \{1, 2, \ldots, 6\}$ and $E = \{1-5, 5-4, 4-3, 3-2, 2-1, 1-6, 2-6, 3-6, 4-6, 5-6, 5-2, 5-3, 2-3\}$. We display a digraph edge $u-v$ by drawing an arrow from the start vertex $u$ to the stop vertex $v$. E.g., in Figure 3(b), vertex 6 is the stop vertex of each of the edges that it is incident on. So all these edges are “directed” towards vertex 6. In contrast, the curve segments in bigraphs are undirected (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 $a-a$ is called a loop. For bigraphs, a loop would correspond to a set $\{a, a\} = \{a\}$. 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 $S$ together with a function $\mu : S \rightarrow N$. We call $S$ the underlying set of $S$ and $\mu(x)$ is the multiplicity of $x \in S$. E.g., if $S = \{a, b, c\}$ 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 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 $K_3$ 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).

![Bipartite Graphs](image)

**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).

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 $W : E \rightarrow \mathbb{R}$. 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, $G = (V, E; \cdot \cdot \cdot)$. Alternatively, $G$ is a graph, and we want to add some additional data $d, d'$, we may also write $(G; d, d')$, etc.

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 $\geq 2$. Consider the set $W = \{w \in V : w - v \in E, w - u \in E\}$. What can you say about the set $W$?

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

Exercise 1.4:
(i) How many bigraphs, digraphs, hypergraphs are there on $n$ vertices?
(ii) How many non-isomorphic bigraphs, digraphs, hypergraphs are there on $n$ vertices? 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 $[u_0, e_1, u_1, e_2, u_2, \ldots, u_k, e_k]$ 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 $E \subset \binom{V}{3}$. 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 $\{u, v\} \subset f$ for some face $f$; in this case, we say $f$ is incident on $e$, and $e$ bounds $f$. 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 $a \rightarrow b$, $b \rightarrow c$ and $c \rightarrow a$. Show that $G$ is planar iff its underlying bigraph is planar in the usual sense.

§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 $u \rightarrow v$ 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:

```
```
Lecture IV

Define \( \delta \) 

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

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

In a bigraph, \( p \) is a path iff \( p^R \) is a path.

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

- (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 \( i = 0, 1, \ldots, k \), \( \delta(v_0,u_i) = i \). 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

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

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, then 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 assumed that the operation is well-defined.
8. Cycles. Two paths $p, q$ are cyclic equivalent if there exists paths $r, r'$ such that

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

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

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

$$(1\!-\!2\!-\!3\!-\!4\!-\!1) \equiv (2\!-\!3\!-\!4\!-\!1\!-\!2) \equiv (3\!-\!4\!-\!1\!-\!2\!-\!3) \equiv (4\!-\!1\!-\!2\!-\!3\!-\!4).$$  

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

$$(1\!-\!2\!-\!3\!-\!4\!-\!1) = (1\!-\!2\!-\!3); (3\!-\!4\!-\!1), \quad (3\!-\!4\!-\!1\!-\!2\!-\!3) = (3\!-\!4\!-\!1); (1\!-\!2\!-\!3).$$  

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

It is easily checked that cyclic equivalence is a mathematical equivalence relation. We define a cycle as an equivalence class of closed paths. If the equivalence class of $p$ is the cycle $Z$, we call $p$ a representative of $Z$; if $p = (v_0, v_1, \ldots, v_k)$ then we write $Z$ using square brackets $[\cdots]$ in one of the forms:

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

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

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

- The length of $Z$ is the length of $p$.
- 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.
- A cycle $Z = [p]$ is trivial if $p$ is a trivial path. So a trivial cycle is written $[(v_0)] = [].$

We now define the notion of a “cyclic graph”. For a digraph $G$, we say it is cyclic if it contains any nontrivial cycle. But for bigraphs, this simple definition will not do. To see why, we note that every edge $u\!-\!v$ in a bigraph gives rise to the nontrivial cycle $[u, v]$. Hence, to define cyclic bigraphs, we proceed as follows: first, define a closed path $p = (v_0\!-\!v_1\!-\!\cdots\!-\!v_{k-1}, v_0)$ to be reducible if $k \geq 2$ and for some $i = 1, \ldots, k$,

$$v_{i-1} = v_{i+1}$$  

Chee Yap  Fundamental Algorithms, Fall 2011: Basic Version  October 31, 2011
where subscript arithmetic are modulo $k$ (so $v_k = v_0$ and $v_{k+1} = v_1$). Otherwise $p$ is said to be irreducible. A cycle $Z = [p]$ is reducible iff any of its representative $p$ is reducible. Finally, a bigraph is said to be cyclic if it contains some irreducible non-trivial cycle.

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

\section{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 $v$. 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 $G$.

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.

![Connected Digraph](image)

Figure 5: (a) Digraph $G_6$, (b) Component graph of $C = \{2, 3, 5\}$, (c) Reduced graph $G'_6$

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

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 $G'$ 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 $G'$ will be re-interpreted using biconnectivity.

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

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 $p = [v_1, \ldots, v_k]$ and $q = [u_1, \ldots, u_\ell]$, 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 $p = [v_1, \ldots, v_k]$ 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
  \[
  \{(a : b, d), (b : a, d, e), (c : b, d, e), (d : a, b, c), (e : c)\}
  \]
  and
  \[
  \{(1 : 5, 6), (2 : 1, 3, 6), (3 : 2, 6), (4 : 3, 6), (5 : 2, 3, 4, 6), (6 :)\}
  \]
  This is supposed 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
  \[
  \begin{bmatrix}
  a & b & c & d & e \\
  a & 0 & 1 & 0 & 0 \\
  b & 0 & 1 & 1 & 0 \\
  c & 1 & 0 & 0 & 1 \\
  d & 0 & 1 & 0 & 0 \\
  e & 0 & 0 & 1 & 0 
  \end{bmatrix}, \quad
  \begin{bmatrix}
  1 & 0 & 0 & 0 & 0 \\
  2 & 0 & 1 & 0 & 0 \\
  3 & 0 & 1 & 0 & 0 \\
  4 & 0 & 0 & 0 & 1 \\
  5 & 0 & 0 & 0 & 0 \\
  6 & 1 & 2 & 3 & 4 
  \end{bmatrix}
  \]
  Note that the matrix for bigraphs is symmetric. The adjacency matrix can be generalized to store arbitrary values to represent weighted graphs.

### §11. Size Parameters.
Two size parameters are used in measuring the 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 \( T(n, m) = O(m + n) \). 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.

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.

The \( O(m + n) \) time bound is the “gold standard” for pure graph algorithms: try to achieve this bound whenever possible.
12. 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 have more efficient access and use less space than linked lists. For instance, arrays allow 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 vertex. We can use an array $A[1..n]$ where $A[i]$ is the value of the $A$-attribute of vertex $i$. For instance, if the attribute values are real numbers, we often call $A[i]$ the “weight” of vertex $i$. If the attribute values are elements of some finite set, we may call $A[i]$ the “color” of vertex $i$.

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

$$
\text{white } \Rightarrow \text{gray } \Rightarrow \text{black},
\text{unseen } \Rightarrow \text{seen } \Rightarrow \text{done}
$$

(3)

For instance, let the color status be represented by the integer array $\text{color}[1..n]$, 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 $\text{color}[i]++$. Sometimes, a two-color scheme is sufficient: in this case we omit the gray color or the done status.

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 $u\rightarrow v$ is an edge of $G$ iff $v\rightarrow u$ is an edge of $G^{rev}$. You must show that your algorithm has the stated running time.

PROBLEM REQUIREMENT: your algorithm should directly modify $G$ into its reverse. 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.
Q: In the above algorithm with an input graph $G$, which is better: (i) an algorithm to directly modify the graph $G$ into its reverse, or (ii) an algorithm that preserve $G$ and create a new graph to represent $G$?
A: Of course, in the above question, we asked you to do the former. More generally, is it better to have (i) a destructive algorithm, or (ii) a conservative algorithm? Note that in C++, you indicate that an argument is conserved by tagging the argument with const. The answer may depend on the application. But generally, I feel the destructive version is more useful. You might object, saying you want to keep the original graph. My response is that you can first create a copy before calling the algorithm. 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.

Exercise 3.2: Let $G$ be 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. ♦

---

§4. Breadth First Search

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.
14. 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
  - $u \leftarrow Q.Remove()$
  - For each vertex $v$ adjacent to $u$
    - If $v$ is unseen, color it as seen
    - $Q.insert(v)$

This algorithm will reach all nodes that are reachable from the source $s_0$. To visit all nodes, 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”.

![Figure 6: (a) BFS tree. (b) Non-tree edges.](image)

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

$$\delta(s_0, u) < \delta(s_0, v) < \infty.$$  \hspace{1cm} (4)

If $\delta(s_0, u) = \infty$, then $u$ will not be visited from $s_0$. The BFS algorithm does not explicitly compute the relation (4) to decide the next node to visit: below, we will prove that this is a consequence of using the
queue data structure.

\section*{15. The BFS Shell} The key to the BFS algorithm is the \textit{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

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

respectively. To keep track of the status of vertices we will use the color scheme (3) 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:

\begin{verbatim}
BFS SHELL
Input: $G = (V, E; s_0)$ a graph (bi- or di-).
Output: This is application specific.
\end{verbatim}

\begin{verbatim}
 Initialization:
0 INIT($G$, $s_0$) \small comment: 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) \small comment: Visit $s_0$ as root
\end{verbatim}

\begin{verbatim}
 Main Loop:
 while $Q \neq \emptyset$ do
3 $u \leftarrow Q.\text{dequeue}()$. \small comment: Begin processing $u$
4 \text{for each $v$ adjacent to $u$ do} \small comment: Process edge $u \rightarrow v$
5 \text{PREVISIT($v$, $u$)} \small comment: Previsit $v$ from $u$
6 \text{if $v$ is unseen then} \small comment: Color $v$ seen
7 \text{VISIT($v$, $u$)} \small comment: Visit $v$ from $u$
8 \text{Q.\text{enqueue}(v)}.$
9 \text{POSTVISIT($u$)}
10 \text{CLEANUP($G$)}
\end{verbatim}

This BFS shell program contains the following shell macros

\begin{verbatim}
INIT, PREVISIT, VISIT, POSTVISIT, CLEANUP
\end{verbatim}

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

Note that VISIT($v$, $u$) represents visiting $v$ from $u$; a similar interpretation holds for PREVISIT($v$, $u$). We set $u = \text{nil}$ 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.

$^5$ 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.

$^6$ Below, the Recursive DFS Shell will allow an exception.
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

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

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.

**16. 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 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 a 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”.

**17. 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 algorithm 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:

\[
\delta(u_1) \leq \delta(u_2) \leq \cdots \leq \delta(u_k) \leq 1 + \delta(u_1).
\]  

(7)

**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 (7) 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 (7).

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 \( \delta_j := \delta(u_{j+1}) - \delta(u_j) = 0 \) or 1 for all \( j = 1, \ldots, k - 1 \). Moreover, there is at most one \( j \) such that \( \delta_j = 1 \). 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 telescoping, 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 \( \pi \), 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.

§18. 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 \( c \rightarrow d \) 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.

**19. 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.

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 \( p = (v_0, v_1, \ldots, v_k) \). 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.

**20. 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 \( \text{unseen} \). Here is the driver program:

```
BFS DRIVER SHELL

Input: \( G = (V, E) \) a graph.
Output: Application-dependent.

Initialization:
1. Color all vertices as \( \text{unseen} \).
2. DRIVER_INIT(G)

Main Loop:
3. for each vertex \( v \) in \( V \) do
4.     if \( v \) is \( \text{unseen} \) then
5.         call BFS((V, E; v)).
```
Note that with the BFS Driver, we add another shell macro called DRIVER\_INIT to our collection (5). 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.

\section*{21. 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 $\Theta(n + m)$.

\section*{22. 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.

\section*{23. 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 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.
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 PRE-VISIT as well?

Exercise 4.3: 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 4.4: Reorganize the BFS algorithm so that the coloring steps are folded into the shell macros of INIT, VISIT, etc.

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

Exercise 4.6: (a) Give a classification of the edges of a digraph $G$ relative to the operations of running the BFS algorithm on $(G; s_0)$. We should see two new types of edges.
(b) Now turn your answer in 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.

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

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

Exercise 4.9: The text sketched an algorithm for testing if a graph is bipartite. We verify some of the assertions there:
(a) Prove that if a bigraph has no odd cycles, then it is bipartite.
Lecture IV

We call this formulation the Nonrecursive Depth First Search (Nonrecursive DFS) algorithm, to distinguish it from the Standard DFS algorithm which is a recursive one.

Here is a general account of DFS: as in BFS, we want to visit all the vertices that are reachable from an initial source $s_0$. We define a DFS tree underlying the DFS computation — the edges of this tree are precisely those $u \rightarrow v$ such that $v$ is seen from $u$. 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. The stack data structure turns out to be perfect for organizing this search.

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

Exercise 4.10: Let $G$ be a digraph. A global sink is a node $u$ such that for every node $v \in V$, there is a path from $v$ to $u$. A global source is a node $u$ such that for every node $v \in V$, there is path from $u$ to $v$.

(a) Assume $G$ is a DAG. Give a simple algorithm to detect if $G$ has a global sink and a global source. Your algorithm returns YES if both exists, and returns NO otherwise. Make sure that your algorithm takes $O(m + n)$ time.

(b) Does your algorithm work if $G$ is not a DAG? If not, give a counter example which makes your algorithm fail.

Exercise 4.11: Let $k \geq 1$ be an integer. A $k$-coloring of a bigraph $G \equiv (V, E)$ is a function $c : V \rightarrow \{1, 2, \ldots, k\}$ such that for all $u \rightarrow v \in E$, $c(u) \neq c(v)$. 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 $\leq 2$?

(b) What is the smallest graph which is not 3-colorable?

(c) The subdivision of an edge $u \rightarrow v$ is the operation where the edge is deleted and replaced by a path $u \rightarrow w \rightarrow v$ of length 2 and $w$ is a new vertex. Call $G'$ a subdivision of another graph $G$ if $G'$ is obtained from $G$ by 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 4.12: Let $G \equiv (V, E)$ 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 $\infty$.

§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 §4. We call this formulation the Nonrecursive DFS algorithm, to distinguish it from the Standard DFS algorithm which is a recursive one.

The Nonrecursive DFS algorithm will disconnect $u$ and $v$, i.e., $\delta(u, v)$ will become $\infty$.
In illustration, consider the digraph $G$ in Figure 7(i) Starting from the source vertex 1, one possible path to a leaf is $(1 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6)$. 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 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.

![Figure 7: Two DFS trees for digraph (i).](image)

### §24. Nonrecursive DFS.
We describe a version of DFS that is parallel to BFS (§15). 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 $S.push(u)$ and $u \leftarrow S.pop()$.

The reader who just wants to know about DFS may go directly to the next section that describes the simpler, recursive formulation of DFS. One motivation for looking at the nonrecursive DFS is to expose the subtleties that might otherwise be lost in the standard formulation.

It is not just a matter of “traversing all nodes”, but the careful ordering of VISITs, PREVISITs and POSTVISITS to nodes. These are essential if DFS is to do the tasks that it is normally called upon to solve. What does it mean to VISIT a vertex $u$? Among other (application-dependent) things, we shall associate VISITing a node $u$ with processing its adjacency list – it is important that processing is done at most once, to ensure an $O(m + n)$ time complexity. Those vertices in the adjacency list that turn into children of $u$ will be pushed on the stack. We also need to POSTVISIT $u$ after all its children have been VISITed. It turns out that in most applications of DFS, this POSTVISIT macro has a critical role. To implement POSTVISIT($u$) in our stack framework, we shall push a copy of $u$ in the stack so that when it is popped, we will invoke the POSTVISIT macro. To indicate that this copy of $u$ in the stack is meant for POSTVISIT (not for the usual VISIT), we will “mark” $u$ before pushing it (back) into the stack. When we pop a vertex from the stack, we can check if the vertex is marked or not.

We now use a tricolor scheme: initially a vertex is colored unseen. After it is pushed on the stack, they are colored seen. Remark that it may be pushed on the stack more than once. After it is VISITed they are colored done. We remark that the “marking” of vertices is independent of this coloring scheme.

---

7 Reproduced from Figure 3(b), for convenience.
Like BFS, this DFS program has two main loops: an outer while-loop (Line 2) and an inner for each-loop (Line 9). Note that the color changes unseen → seen → done for node $u$ are performed at the appropriate moments (when pushing $u$ onto the stack, and when VISITing $u$). One important point is that before we push $v$ on the stack (Line 12), we make sure that $v$ is not done, but it could be seen or unseen; if seen, it means that there is one or more copies of $v$ are already on the stack (these are now redundant). Correctness of the algorithm is unaffected by redundancy, but it means that stack size is bounded by $m$ (not $n$). If this efficiency issue is of concern, we could introduce "smart stacks" that can remove such redundancies. 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.

\[\textbf{25. DFS Driver.}\] Finally, if we need to visit all vertices of the graph, we can use the following DFS Driver Program that calls Nonrecursive DFS repeatedly:

\begin{verbatim}
DFS DRIVER
\textbf{Input:} $G = (V, E)$ a graph (bi- or di-)
\textbf{Output:} Application-specific
1 \hspace{1em} DRIVER\_INIT($G$)
2 Color each vertex in $V$ as unseen.
3 for each $v$ in $V$ do
4 \hspace{1em} if $v$ is unseen then
5 \hspace{2em} DFS($V, E; v$) \hspace{1em} \langle Either Nonrecursive or Standard DFS
6 DRIVER\_CLEANUP($G$)
\end{verbatim}
26. 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 VISIT($v, u$) and is left as an Exercise. We prove a basic fact about DFS:

**Lemma 6 (Unseen Path).** Let $u, v \in V$.

Then $v$ is a descendant of $u$ in the DFS tree if and only if at the time instant that $u$ was first seen, there is an "unseen path" from $u$ to $v$, i.e., a path $(u \leftarrow \cdots \leftarrow v)$ comprising only of unseen vertices.

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

(⇒) 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 $(u \rightarrow u_1 \rightarrow \cdots \rightarrow u_k \rightarrow v)$ 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 when 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.

(⇐) 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 $a <_{dfs} b$ (relative to $u$). If $p = (u \rightarrow u_1 \rightarrow \cdots \rightarrow u_k \rightarrow v)$ and $q = (u \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_l \rightarrow v)$ (where $k, \ell \geq 0$) are two distinct paths from $u$ to $v$, we say $p <_{dfs} q$ if there is an $m (1 \leq m < \min\{k, \ell\})$ such that $u_1 = v_1, \ldots, u_m = v_m$ and $u_{m+1} <_{dfs} v_{m+1}$ 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 $<_{dfs}$-least unseen path from $u$ to $v$ at time we first see $u$. By an unseen path from $u$ to $v$, we mean one

$$\pi : (u \rightarrow u_1 \rightarrow \cdots \rightarrow u_k \rightarrow v)$$

(9)

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 (9) is the $<_{dfs}$-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 $<_{dfs}$-least unseen path from $u$ to $v$ is just $(u \rightarrow v)$. So $v$ is adjacent to $u$. Suppose $v'$ is a vertex such that $v' <_{dfs} v$ (relative to $u$). Then there does not exist an unseen path $\pi'$ from $v'$ to $v$; otherwise, we get the contradiction that the path $(u \rightarrow v') \pi' <_{dfs}$ than than $(u \rightarrow v)$. 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., $u \rightarrow v$ is an edge of the DFS tree.

**Inductive Case:** Suppose $k > 0$. Let $\pi$ in (9) be the $<_{dfs}$-least unseen path of length $k + 1$ from $u$ to $v$. As before, if $v' <_{dfs} u_1$ then we will recursively visit $v'$, we will never color any of the

---

8 If we use the white-black coloring scheme, this may be called the “white path” as in [4].
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, \( \pi' \) remains the least unseen path from \( u_1 \) to \( v \) at time \( t_1 \). By IND\((k-1)\), the subpath \( \pi' \) is in the DFS tree. Hence the path \( \pi = (u-u_1); \pi' \) is in the DFS tree. Q.E.D.

\( \section{Classification of digraph edges by DFS.} \) First consider a digraph \( G \). Upon calling \( DFS(G, s_0) \), the edges of \( G \) becomes classified as follows (see Figure 7):

- **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). There are actually two possibilities: \( u, v \) may or may not share a common ancestor. Note that in Figure 7, we only have the case where \( u, v \) share a common ancestor, but it is easy to construct examples where this is not the case.
- **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.

Let us give a simple application of the Unseen Path Lemma:

**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 exactly 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 \( u-v \). 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 \( v_k \). By the Unseen Path Lemma, this implies that \( v_k \) will become a descendant of \( v_1 \) in the DFS forest. Clearly, \( v_k-v_1 \) is a back edge in the forest. Q.E.D.

Thus detecting cycles in graphs can be reduced to detecting back edges. More generally, we will address the computational classification of the edges of a DFS forest. Before we do this in full generality, we look at the simpler case of classifying bigraph edges.

\( \section{Computational classification of bigraph edges by DFS.} \) 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 \( G \). So the above classification (\( \section{27} \)) 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 its 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.

We can now simplify the classification of edges by regarding \( G \) (after running DFS on \( G \)) as a “hybrid graph”, with both directed and undirected edges. The undirected edges of \( G' \) are precisely the unseen edges of \( G \). But the seen edges of \( G \) are converted into directed edges as follows: tree edges are directed from parent to child, and back edges. Thus, the (modified) \( G \) just have three kinds of edges:

\[ \text{tree, back, unseen.} \]

We address the computational problem of classifying edges according to (10). We will 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 \( \S 24 \), we can detect the forward/back during \( \text{PREVISIT}(v, u) \) (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 without more information.

The solution is to introduce “time”. We plan to record the time when we first see a node. Then in \( \text{PREVISIT}(v, u) \), 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 \( \text{clock} \) that is initially 0. We introduce an array, \( \text{firstTime}[v : v \in V] \) such that \( \text{firstTime}[v] \) is set to the value of \( \text{clock} \) when we first see \( v \) (and the value of \( \text{clock} \) 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:

\[
\text{INIT}(G, s_0) : \text{clock} \leftarrow 0
\]

\[
\text{PREVISIT}(v, u) : \\
\quad \text{If } v \text{ is unseen,} \\
\quad \quad \text{firstTime}[v] \leftarrow \text{clock}++ \\
\quad \quad p[v] \leftarrow u \quad \text{So } u = p[v] \text{ is parent of } v, \text{ and } p[v]-v \text{ is “tree-edge”} \\
\quad \text{elseif } (\text{firstTime}[u] > \text{firstTime}[v]) \\
\quad \quad \text{NO-OP} \quad \text{a } u-v \text{ is “back edge”} \\
\quad \text{else} \\
\quad \quad \text{NO-OP} \quad a u-v \text{ is “forward edge”} \\
\]

\( \S 29. \) Biconnectivity. When we introduced reduced graph earlier, 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 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 \( v \). 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 a simple cycle.

\[ \text{One may feel } [u-v] \text{ is considered “simple” by a technicality.} \]
singleton \{ u \} is also a biconnected set, for a trivial reason. 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 \( \leq 2 \) are trivial. Trivial components are of two types: those of size 1 are called isolated components and those of size 2 are called bridges.

![Figure 8: Graph with 3 non-trivial biconnected components and 8 bridges](image)

E.g., the bigraph in Figure 3(a) has two biconnected components, \{a, b, c, d\} and \{c, e\}. Moreover, \{a, b, c\} is a biconnected set but \{a, e\} is not. A more interesting graph is Figure 8 which has 11 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 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.

There is an edge analogue of cut-vertex: an edge \( u - v \) is called a bridge-edge if the removal of this edge will increase the number of connected components of the resulting bigraph. The two endpoints of a bridge-edge constitute a bridge (a biconnected component of size 2). 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 9 bridges and 10 cut-vertices.

**Lemma 10.** 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 \( w \) is adjacent to \( w \), then \( w \) is also a descendant of \( u \). Note that a vertex \( w \) may be equal to \( v \).
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 $\bar{w}$, where $\bar{w}$ not a descendant of $u$. Since the BFS tree has no cross edges, $\bar{w}$ must be an ancestor of $u$. So there is a path in the BFS tree from $\bar{w}$ to $w$. This path, together with the edge $w-\bar{w}$ 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.

30. 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 $\text{numChildren}$ count the number of children of the root. Initially, $\text{numChildren} \leftarrow 0$, 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 $\text{mft}[u]$ denote the minimum value of $\text{firstTime}[w]$ 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 descendant of $u$. 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 $\infty$, so $\text{mft}[u] = \infty$ iff $B(u)$ is empty. We now address three questions:

- What is the significance of $\text{mft}[u]$? Suppose $u$ is not the root of the DFS tree. Claim: $u$ is a cut-vertex if there exists a child $v$ of $u$ such that $\text{mft}[v] \geq \text{firstTime}[u]$. In proof, if $u$ is a cut-vertex, then condition (ii) provides a child $v$ of $u$ such that $\text{mft}[v] \geq \text{firstTime}[u]$. Conversely, suppose $\text{mft}[v] \geq \text{firstTime}[u]$. Take any path from $v$ to $p[u]$. There is a first edge $w-w'$ in this path such that $\text{firstTime}[w] > \text{firstTime}[u] > \text{firstTime}[w']$. 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 $\text{mft}[v] \leq \text{firstTime}[w'] < \text{firstTime}[u]$, 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 $\text{mft}[u]$? We initialize $\text{mft}[u]$ to $\infty$ when $u$ is first seen. We subsequently update $\text{mft}[u]$ in two ways:
  (i) When we detected a back edge $u-v$, we will update $\text{mft}[u]$ with $\min\{\text{mft}[u], \text{firstTime}[v]\}$.
  (ii) When we POSTVISIT($v$), and $p[v] = u$, we can update $\text{mft}[u]$ to $\min\{\text{mft}[u], \text{mft}[v]\}$.

- How do we use $\text{mft}[u]$ computationally? We can only use it to detect cut-vertices: in POSTVISIT($u$), we check if $\text{mft}[u] \geq \text{firstTime}[p[u]]$, and $p[u]$ is not the root.

To summarize the algorithm, here are the shell macros:
5. NONRECURSIVE DEPTH FIRST SEARCH

\[ \text{INIT}(G, s_0) : \]
\[
\begin{align*}
\text{clock} & \leftarrow 0 \\
p[s_0] & = s_0 \\
\text{numChildren} & = 0
\end{align*}
\]

\[ \text{PREVISIT}(v, u) : \]
\[
\begin{align*}
\text{If } (v \text{ is unseen}), \\
\text{firstTime}[v] & \leftarrow \text{clock}++ \\
p[v] & \leftarrow u \quad \langle u-v \text{ is “tree-edge”} \rangle \\
\text{mft}[v] & \leftarrow \infty \\
\text{If } p[u] & = u \quad \langle u \text{ is root} \rangle \\
\text{If } (++\text{numChildren} > 1) \\
\text{Return} & (“u \text{ is cut-vertex”})
\end{align*}
\]
\[
\begin{align*}
\text{elif } (\text{firstTime}[u] > \text{firstTime}[v]) \\
\text{mft}[u] & \leftarrow \min \{\text{mft}[u], \text{firstTime}[v]\}
\end{align*}
\]

\[ \text{POSTVISIT}(u) : \]
\[
\begin{align*}
\text{If } (\text{mft}[u] \geq \text{firstTime}[p[u]]) \text{ and } (p[u] \neq p[p[u]]) \\
\text{Return} & (“p[u] \text{ is a cut-vertex”})
\end{align*}
\]

§31. Reduced Bigraphs. Given a bigraph \( G \), we define a bigraph \( G^c = (V^c, E^c) \) such that the elements of \( V^c \) are the biconnected components of \( G \), and \( (C, C') \in E^c \) iff \( C \cap C' \) is non-empty. It is easy to see that \( G^c \) is acyclic. We may call \( G^c \) the reduced graph of \( G \). In the Exercise, we ask you to extend the above biconnectivity detection algorithm to compute some representation of \( G^c \).

---

**EXERCISES**

**Exercise 5.1:** T or F: 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 \( \text{INIT}(G), \text{VISIT}(v, u) \) and \( \text{POSTVISIT}(u) \) 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 \( \text{VISIT}(u, v) \) to take two arguments. Show that we could have defined this it as \( \text{VISIT}(u) \), and not lost any functionality in our shell programs. HINT: take advantage of \( \text{PREVISIT}(u, v) \). 
(b) Give an example where it is useful for the Driver to call \( \text{CLEANUP}(u) \) after \( \text{DFS}(u) \).
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 $<_{dfs}$. 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 $CV[v] = true$ 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 $\text{radius}(G)$. Define the diameter $\text{diameter}(G)$ of $G$ to be the maximum value of distance$(u, v)$ where $u, v \in V$.
(a) Prove that $2 \cdot \text{radius}(G) \geq \text{diameter}(G) \geq \text{radius}(G)$.
(b) Show that for every natural number $n$, there are graphs $G_n$ and $H_n$ such that $n = \text{radius}(G_n) = \text{diameter}(G_n)$ and $\text{diameter}(H_n) = n$ and radius$(H_n) = \lceil n/2 \rceil$. This shows that the inequalities in (a) are the best possible.
(c) Using DFS, give an efficient algorithm to compute the diameter of a undirected tree (i.e., connected acyclic undirected graph). Please use shell programming. Prove the correctness of your algorithm. What is the complexity of your algorithm? HINT: write down a recursive formula for the diameter of a tree in terms of the diameter and height of its subtrees.
(d) Same as (c), but compute the radius instead of diameter.
(e,f) Same as (c) and (d) but using BFS instead of DFS.

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. \( \text{INIT}(G, s_0) \) // If standalone, make all vertices unseen except for \( s_0 \)

- **Main Loop:**
  while \( S \neq \emptyset \) do
    2. \( u \leftarrow S.\text{pop}() \).
    3. for each \( v \) adjacent to \( u \) do
      4. \( \text{PREVISIT}(v, u) \)
      5. if \( v \) is unseen then
        6. color \( v \) seen
        7. \( \text{VISIT}(v, u) \)
        8. \( S.\text{push}(u) \)
    9. \( \text{POSTVISIT}(u) \).

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?

---

### Standard Depth First Search

#### §6. Standard Depth First Search

### §32. 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-)
- **Output:** Application dependent

1. Color \( s_0 \) as seen, and \( \text{VISIT}(s_0) \)
2. for each \( v \) adjacent to \( s_0 \) do
3. \( \text{PREVISIT}(v, s_0) \)
4. if \( (v \) is unseen) then
5. Standard DFS((\( V, E; v \)) // Recursive call
6. Color \( s_0 \) done, and \( \text{POSTVISIT}(s_0) \)
7. \( \text{CLEANUP}(G) \).
To visit every vertex of a digraph, we invoke a DFS Driver on the graph. We can keep the DFS Driver in §25, except that each DFS call refers to the Standard DFS.

Our placements of the \texttt{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 §15). 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 \texttt{POSTVISIT} macro.

### §33. 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 a classification of each edge of $G$. Now we can extend the solution method used for bigraphs: recall that we had time stamps and we maintained a classification of every edge of $G$. Indeed, we can extend the solution method used for bigraphs: recall that we had time stamps and we maintained a classification of each edge of $G$. Let

\begin{itemize}
  \item \texttt{DRIVER\_INIT}(G)\equiv \texttt{clock} \leftarrow 0; (\text{for } v \in V)[\texttt{firstTime}[v] \leftarrow \texttt{lastTime}[v] \leftarrow -1].
  \item \texttt{PREVISIT}(v, u)\equiv \text{If } v \text{ is unseen, } \texttt{firstTime}[v] \leftarrow \texttt{clock}++.\]
  \item \texttt{POSTVISIT}(v)\equiv \texttt{lastTime}[v] \leftarrow \texttt{clock}++.\]
\end{itemize}

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

Let $\text{active}(u)$ denote the time interval $[\texttt{firstTime}[u], \texttt{lastTime}[u]]$, and we say $u$ is \textbf{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 $\text{active}(v) < \text{active}(u)$ if $\texttt{lastTime}[v] < \texttt{firstTime}[u]$. We return to the computational classification of the edges of a digraph $G$ relative to a DFS forest on $G$:

\begin{lemma}
Assume that a digraph $G$ has been searched using the DFS Driver, resulting in a complete classification of each edge of $G$. Let $u \rightarrow v$ be an edge of $G$.
\end{lemma}

\begin{enumerate}
  \item $u \rightarrow v$ is a back edge iff $\text{active}(u) \subseteq \text{active}(v)$.
  \item $u \rightarrow v$ is a cross edge iff $\text{active}(v) < \text{active}(u)$.
  \item $u \rightarrow v$ is a forward edge iff there exists some $w \in V \setminus \{u,v\}$ such that $\text{active}(v) \subseteq \text{active}(w) \subseteq \text{active}(u)$.
\end{enumerate}
4. \( u-v \) is a tree edge iff \( \text{active}(v) \subseteq \text{active}(u) \) but it is not a forward edge.

This above classification of edges by active ranges is illustrated in Figure 9.

![Figure 9: Relative positions of active ranges of \( u, v \) and the classification of edge \( (u-v) \)](image)

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

\begin{verbatim}
PREVISIT(v, u)
  \triangleright Visiting v, from u
  if (firstTime[v] = -1),
    mark \( u-v \) as “tree edge”
  elif (firstTime[v] > firstTime[u]),
    mark \( u-v \) as “forward edge”
  elif (lastTime[v] = -1),
    mark \( u-v \) as “back edge”
  else
    mark \( u-v \) as “cross edge”.
\end{verbatim}

The correctness of this classification is a direct consequence of Lemma 11 (cf. Figure 9). If the arrays \( \text{firstTime}, \text{lastTime} \) are not initialized, we could replace the above code as follows: instead of the test \( \text{firstTime}[v] = -1 \), we could check if “\( v \) is unseen”. Instead of the test \( \text{lastTime}[v] = -1 \), we could check if “\( v \) is seen” (thus not yet done).

\section*{34. 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.

Let \( P \) be the set of processes and \( R \) the set of resources. We introduce a bipartite graph \( G = (P, R, E) \) where \( V = P \cup R \) is the vertex set and \( E \subseteq (P \times R) \cup (R \times P) \). See Figure 10 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 10 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 (§24) and the Standard DFS equivalent?  

**Exercise 6.3:** Suppose \(G = (V, E; \lambda)\) is a strongly connected digraph in which \(\lambda : E \rightarrow \mathbb{R}_{>0}\). A **potential function** of \(G\) is \(\phi : V \rightarrow \mathbb{R}\) 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 ifff property \( P(G) \) holds.
(c) Give an algorithm to compute a potential function for \( G \) ifff \( P(G) \) 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 \( G_1 = (V_p \cup V_R, E_i) \) where \( E_0 = \emptyset \) (empty set) and for \( i \geq 1 \), if \( p_i \) is the \( j \)th occurrence of the process \( p_i \) in \( e \), then \( G_i \) is the result of \( p_i \) executing its \( j \)th instruction on \( G_{i-1} \). If \( p_i \) has no \( j \)th 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 \( i \)th 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.

---

\( \text{End Exercises} \)

---

§7. Further Applications of Graph Traversal

© Chee Yap

Fundamental Algorithms, Fall 2011: Basic Version

October 31, 2011
In the following, assume $G = (V, E)$ is a digraph with $V = \{1, 2, \ldots, n\}$. Let $\text{per}[1..n]$ be an integer array that represents a permutation of $V$ in the sense that $V = \{\text{per}[1], \text{per}[2], \ldots, \text{per}[n]\}$. This array can also be interpreted in other ways (e.g., a ranking of the vertices).

\subsection{Topological Sort.}

One motivation is the so-called\footnote{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 Critical Path Method (CPM) developed around the same time.} 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.

Let $\{v_1, v_2, \ldots, v_n\}$ be a listing of the vertices in $V$. We call it a \textbf{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 \textbf{reverse topological sort}.

If an edge $u - v$ is interpreted as saying “activity $u$ must precede activity $v$”, then a topological sort gives 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 \textbf{rank} $i$ in the topological sort (11). Hence, topological sort amounts to computing this rank attribute of each vertex. We introduce an array $\text{Rank}[v \in V]$ 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 $\text{Rank}[v_1, v_2, v_3, v_4] = [2, 3, 1, 4]$.

We use the DFS Driver to compute the rank attribute array. We must initialize the $\text{Rank}$ array in the DRIVER\_INIT.

\begin{align*}
\text{DRIVER\_INIT}(G) & \equiv (\text{for } v = 1 \text{ to } n, \text{Rank}[v] \leftarrow -1).
\end{align*}

Indeed, we need not use a separate color array, but simply interpret a Rank of $-1$ as \textbf{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:

\begin{align*}
\text{Rank}[v] & \leftarrow R-.-.
\end{align*}

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 (12), 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 (12) should...
Further Applications of Graph Traversal

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 \( \text{Rank} \), i.e., an array \( i\text{Rank}[1..n] \) such that

\[ i\text{Rank}[i] = v \iff \text{Rank}[v] = i \tag{13} \]

Thus we just have to replace (12) by

\[ i\text{Rank}[R--] \leftarrow v. \tag{14} \]

The topological sort (11) is then given by

\((i\text{Rank}[1], i\text{Rank}[2], \ldots, i\text{Rank}[n])\).

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

```plaintext
STRONG_COMPONENT_DRIVER(G, Per)
  INPUT: Digraph G and permutation Per[1..n].
  OUTPUT: DFS Spanning Forest of G.
  Initialization
  1. For \( v = 1, \ldots, n \), \( \text{color}[v] = \text{unseen} \).
  Main Loop
  2. For \( v = 1, \ldots, n \),
  3. If \( \text{color}[\text{Per}[v]] = \text{done} \)
  4. \( DFS_1(\text{Per}[v]) \) \( \triangleright \) Outputs a DFS Tree
```

This program is the usual DFS Driver program, except that we use \( \text{Per}[i] \) to determine the choice of the next vertex to visit, and it calls \( DFS_1 \), a variant of \( DFS \). We assume that \( DFS_1(i) \) will (1) change the color of every vertex that it visits, from \( \text{unseen} \) to \( \text{done} \), and (2) output the DFS tree rooted at \( i \). If \( \text{Per} \) is correctly chosen, we want each DFS tree that is output to correspond to a strong component of \( G \).
First, let us see how the above subroutine will perform on the digraph \( G_6 \) in Figure 5(a). Let us also assume that the permutation is

\[
\text{Per}[1, 2, 3, 4, 5, 6] = [6, 3, 5, 2, 1, 4] = [v_6, v_3, v_5, v_2, v_1, v_4].
\]

The output of \textsc{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,

\[
\text{Per}[1, 2, 3, 4, 5, 6] = [1, 2, 3, 4, 5, 6],
\]

our \textsc{Strong-Component-Driver} will first call \textit{DFS}_{1}(\text{Per}[1]). This produces a DFS tree containing the vertices \( \{1, 2, 3, 5, 6\} \). Only one vertex 4 remain unseen, and so the driver will next call \textit{DFS}_{1}(\text{Per}[4]) which produces a DFS tree containing \( \{4\} \). Thus, the identity permutation does not lead to the correct output for strong components.

It is 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 \( \text{Per}[1..n] \) is said to be \textbf{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 \( \text{Per}[1..n] \) (the inverse of the \( \text{Rank}[1..n] \)). The next lemma shows that \( \text{Per}[1..n] \) almost has the properties we want. For any set \( C \subseteq V \), we first define

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

\textbf{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 \( \text{Rank}[u_0] = \text{Rank}[C] \).

(b) If there is path from \( C \) to \( C' \) in the reduced graph of \( G \), then \( \text{Rank}[C] \leq \text{Rank}[C'] \).

\textbf{Proof.} (a) By the Unseen Path Lemma, every node \( v \in C \) will be a descendant of \( u_0 \) in the DFS tree. Hence, \( \text{Rank}[u_0] \leq \text{Rank}[v] \), and the result follows since \( \text{Rank}[C] = \min\{\text{Rank}[v] : v \in C\} \).

(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), \( \text{Rank}[C] = \text{Rank}[u_0] \). 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 \( u_0 \), \( \text{Rank}[u_0] \leq \text{Rank}[u_1] \). By part (a), \( \text{Rank}[u_1] = \text{Rank}[C'] \). Thus \( \text{Rank}[C] \leq \text{Rank}[C'] \). (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, \( \text{Rank}[C_0] < \text{Rank}[u_0] \). But \( \text{Rank}[u_0] = \text{Rank}[C'] \). \( \square \)
Is the reverse "topological sort" ordering

\[ [i\text{Rank}[n], i\text{Rank}[n - 1], \ldots, i\text{Rank}[1]] \]  (17)

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 (17).

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 (17). Why? Because \( u \) (and not \( v \)) is the first vertex of \( C \) to appear in the list (17).

We use another insight: consider the reverse graph \( G^{rev} \) (i.e., \( u \rightarrow v \) is an edge of \( G \) iff \( v \rightarrow u \) is an edge of \( G^{rev} \)). It is easy to see that \( C \) is a strong component of \( G^{rev} \) if \( C \) 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 \( i\text{Rank}[1..n] \) is the result of running topological sort on \( G^{rev} \) then \( i\text{Rank} \) 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 (17). That means that the first vertex of \( C \) is listed after the first vertex of \( C' \) in the listing \( [i\text{Rank}[1], i\text{Rank}[2], \ldots, i\text{Rank}[n]] \). This is good. \( \square \)

We now have the complete algorithm:

```
STRONG_COMPONENT_ALGORITHM(G)
   INPUT: Digraph \( G = (V, E), V = \{1, 2, \ldots, n\} \).
   OUTPUT: A list of strong components of \( G \).
1. Compute the reverse graph \( G^{rev} \).
2. Call topological sort on \( G^{rev} \).
   This returns a permutation array \( i\text{Rank}[1..n] \).
3. Call STRONG_COMPONENT_DRIVER(\( G, i\text{Rank} \))
```

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 \( i\text{Rank}[1..n] \).

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

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

Exercise 7.6: An edge $u \rightarrow v$ 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 $u \rightarrow v$ 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 $h_t(v) = t h_0(v) + (1 - t) h_1(v)$. 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 \( v \rightarrow w \) is an edge. Such an edge \( v \rightarrow w \) 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?

§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).

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.

Alice and Bob
\section*{Game Graphs.} To model this, we introduce the concept of a \textit{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 $v \in V_A$ 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 $v_i \rightarrow v_{i+1}$ determined by Alice (Bob) iff $v_i \in V_A, v_{i+1} \in V_B$. 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 $Out(v) = \{ u \in V : u-v \in E \}$ and $In(v) = \{ u \in V : u-v \in E \}$. The elements of $V$ are also called \textit{states}. A \textit{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 $\pi_X : V_X \rightarrow V$ such that $\pi_X(v) \in Out(v)$ We call $\pi_X$ the \textit{strategy} for Player $X$ ($X$-strategy for short). Typically, we let $\alpha$ denote an $A$-strategy, and $\beta$ denote a $B$-strategy. A \textit{complete strategy} is a pair $(\alpha, \beta)$, which can be succinctly represented by a single function $\pi : V \rightarrow V$. 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 \textit{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 $\omega$. Also “$u \in_{\omega} \omega$” if $u$ occurs infinitely often in $\omega$ (this implies $\omega$ is infinite). We may now define:

\begin{itemize}
  \item Intuitively, $\text{Force}(u)$ is the set of states from which Alice can force the system into state $u$. Formally:
  \[
  \text{Force}(u) := \{ v \in V : (\exists \alpha)(\forall \beta)[u \in \omega(v, \alpha, \beta)] \} .
  \]

  \item Intuitively, $\text{Fair}(u)$ is the set of states from which Alice can force the system to enter state $u$ infinitely often. Formally:
  \[
  \text{Fair}(u) := \{ v \in V : (\exists \alpha)(\forall \beta)[u \in_{\omega} \omega(v, \alpha, \beta)] \} .
  \]
\end{itemize}

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 $\text{Fair}(U)$ is also called the \textit{winning states} for a Büchi game with Büchi objective $U$. Such games originated in mathematical logic. We will design algorithms to compute the sets $\text{Force}(U)$ and $\text{Fair}(U)$ in times $O(n + m)$ and $O(mn)$. The exercises\footnote{From Krishnendu Chatterjee and Monika Henzinger (2011).} will show how $\text{Fair}(U)$ can be computed in $O(n^2)$ time.

\section*{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 $\text{Force}(U)$:

\begin{itemize}
  \item $U \subseteq \text{Force}(U)$
  \item If $v \in V_A$ and $Out(v) \cap \text{Force}(U) \neq \emptyset$ then $v \in \text{Force}(U)$.
  \item If $v \in V_B$ and $Out(v) \subseteq \text{Force}(U) + \text{Force}(U)$ then $v \in \text{Force}(U)$.
\end{itemize}
Let us introduce an operator to capture these properties:

\[ \mu_G = \mu : 2^V \to 2^V \]

such that for all \( U \subseteq V \)

\[ v \in \mu(U) \iff \begin{cases} v \in U, & \text{or} \\ v \in V_A \land (Out(v) \cap U \neq \emptyset), & \text{or} \\ v \in V_B \land (Out(v) \subseteq U) \end{cases} \]

[BASIS]

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:

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

- \( \mu^*(U) \) is a fixed point of \( \mu \):
  \[ \mu(\mu^*(U)) = \mu^*(U). \]

- \( \mu^*(U) \) is the least fixed point of \( \mu \) that contains \( U \):
  \[ (W \supseteq U) \land (W = \mu(W)) \Rightarrow \mu^*(U) \subseteq W. \]

**Lemma 15.** Force\((U)\) is the least fixed point of \( U \). In other words, 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 \( \alpha \) for Alice such that for all strategies \( \beta \) for Bob, if \( \pi = (\alpha, \beta) \) then there exists a \( k \geq 1 \) such that \( \pi^k(u) \in U \). This proves that \( u \in \mu^k(U) \).

\[ \text{Q.E.D.} \]

**§39. Computing** 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 (18) 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 \( v \rightleftharpoons w \)?

For efficiency, we want to examine edges of the form \((u \rightleftharpoons w) \in W^r \times W \) where \( W^r = V \setminus W \). If we redirect this edge from what is known \((W)\) to the unknown \((W')\), we get an \( w \rightleftharpoons u \) 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
\[
C[i] = \begin{cases} 
0 & i \in U \\
1 & i \in V_A \\
degree(i) & i \in V_B
\end{cases} \tag{19}
\]
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:

\[
\mu^*(U):
\begin{align*}
\text{Input: } & G^r = (V_A, V_B, E^r) \text{ and } U \subseteq V = \{1, \ldots, n\} \\
\text{Output: } & \text{Array } C[1..n] \text{ representing } \mu^*(U) \\
\text{ Initialization } & \text{Initialize array } C[1..n] \text{ as in (19)} \\
& \text{Initialize queue } Q \leftarrow U \\
\text{ Main Loop } & \text{while } (Q \neq \emptyset) \\
& \quad w \leftarrow Q.pop() \\
& \quad \text{for each } u \text{ adjacent to } w \text{ in } G^r \\
& \quad \quad \text{if } (C[u] > 0) \\
& \quad \quad \quad C[u]-- \\
& \quad \quad \quad \text{if } C[u] == 0 \text{, } Q.push(u) \\
\text{ Return } (C)
\end{align*}
\]

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.

§40. Computing Fair\((U)\). We next use this as a subroutine to compute Fair\((U)\).

References
