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

(Read Euler, read Euler, he is our master in everything)

— Pierre-Simon Laplace (1749–1827)

$\boxed{e^{\pi \boldsymbol{i}} + 1 = 0}$ (where $\boldsymbol{i}^2 = -1$)

— Leonhard Euler (1748)
(equation connecting the 5 basic numbers)

It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used.

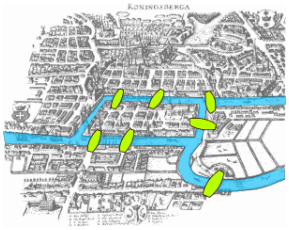
– Gottfried Wilhelm von Leibniz
(on his calculating machine "Step Reckoner" in 1685)

Let us calculate, without further ado, to see who is right.

– Gottfried Wilhelm von Leibniz (1646–1716)
(on his unfinished final project)

Lecture IV
PURE GRAPH ALGORITHMS

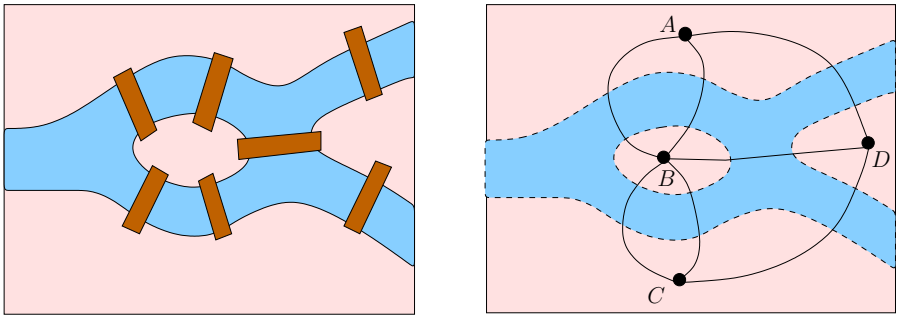
¶1. Graph Theory is said to have originated with Euler (1707–1783). The citizens of the city[1] of Königsberg asked him in 1736 to resolve their favorite pastime question: *is it possible to traverse all the 7 bridges joining two islands in the River Pregel and the mainland, without retracing any path?* See Figure 1(a) for a schematic layout of these bridges. Euler's answer for the citizens is negative (see Exercise below). Euler recognized[2] in this problem the essence of Leibnitz's earlier interest in founding a new kind of mathematics called *analysis situs*. This can be interpreted as topological or combinatorial analysis in modern language. A graph corresponding to the 7 bridges and their interconnections is shown in Figure 1(b). Computational graph theory has a relatively recent history. Among the earliest papers on graph algorithms are Boruvka's (1926) algorithm, and Dijkstra's shortest path algorithm (1959). Tarjan [12] 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 [5]; for algorithmic treatments, see Even [7] and Sedgewick [11].



The actual bridges
Credit: wikipedia

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

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



(a) (b)

Figure 1: The seven bridges of Königsberg

¶2. Real world graphs. Graphs are useful for modeling abstract mathematical relations in computer science as well as in many other disciplines. A graph is fundamentally one or more mathematical relations (called **incidence relations**) between two sets: a **vertex set** V and an **edge set** E. 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–j represents the "adjacency" relationship between countries i and j, i.e., they share a continuous stretch of common border. Clearly, i–j and j–i are considered the same. 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 i–j whenever there is a flight from airport i to another j. Note that i–j and j–i are distinct edges in this representation. Each i–j will typically have auxiliary data associated with it. For example, the data may be a triple of numbers (f, d, c) where f is the number of flights per week, d the flight distance, and c the average ticket price. Likewise, each $i \in V$ may have auxiliary data associated with airport i.

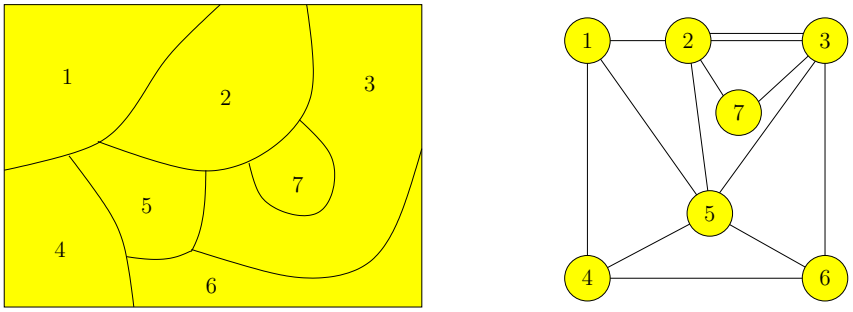
Hypertext Links In hypertext documents on the world wide web, a document will generally have links ("hyper-references") to other documents. We can represent these linkages by a graph whose vertices V represent individual documents, and each edge $(u, v) \in V \times V$ indicates that there is a link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

Relation or relationship? A relation R is a set $R \subseteq A \times B$ where A, B are sets. Each pair $(a, b) \in R$ is a relationship.

We said that a graph is characterized by one or more incidence relations between two set, V and E. An incidence relation I is a subset of $E \times V$, where $(e, v) \in I$ asserts that e is incident on v. Let us illustrate this using Königsberg graph in Figure 1(b): the vertex set is $V = \{A, B, C, D\}$ and there are the 7 edges, say $E = \{e_1, \ldots, e_7\}$. In this case, there is one incidence relation $I \subseteq E \times V$. We may compactly encode I as the set

$$I = \{e_1(A, B), e_2(A, B), e_3(B, C), e_4(B, C), e_5(A, D), e_6(B, D), e_7(C, D)\}$$

with the understanding that "$e(v_1, \ldots, v_m)$" asserts that edge e is incident on each v_i ($i = 1, \ldots, m$). It happens that in our graph, $m = 2$ for each edge. Note that e_1 and e_2 shares exactly the same incidences. We call them "parallel edges". Similarly, e_3 and e_4 are parallel edges. The political map of Figure 2(b) also give rise to a pair of parallel edges.



(a) (b)

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

Sometimes, we need more than one incidence relation. Let us modify the Königsberg example by making the edges "directed" so that two incidences in $e_1(A, B)$ need to be distinguished (say one end of the bridge is the entrance, the other is the exit). Then we need two incidence relation $I_{enter}, I_{exit} \subseteq E \times V$ where $e_1(A, B)$ means $(e_1, A) \in I_{enter}$ and $(e_1, B) \in I_{exit}$.

We shall be studying graphs *without* parallel edges, unlike the Königsberg graph or the political map. In this case, our notations simplifies greatly. If each edge e has two incidences, we can identify e with a pair of vertices $u, v \in V$. The pair can be unordered $e = \{u, v\}$ (if there is only one incidence relation) or ordered $e = (u, v)$ (if there are two incidence relations). We will call these kinds of graphs **bigraphs** and **digraphs**. These two kinds of graphs shares many common concepts and it is therefore desirable to unify them. We shall denote[3] such a pair by "u–v", and rely on context to determine whether an ordered or unordered edge is meant. For unordered edges, we have u–$v = v$–u; but for ordered edges, u–$v \neq v$–u unless $u = v$.

¶3. OVERVIEW OF CHAPTER. The title "Pure Graph Algorithms" of this Chapter emphasizes the fact that the algorithms in this chapter treat graphs that are "pure" in the sense of not having any auxiliary data such as edge labels or weights. The algorithmic issues for pure graphs revolve around 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). The student is already familiar with in tree traversals (§III.4). These traversals will be generalized in our study of BFS and DFS. Specifically, we can turn these graph traversals into algorithms for specific tasks, by inserting task-specific macros at critical junctures in DFS or BFS. These BFS/DFS algorithms form the core of this chapter.

"shell programming" again!

In applications of graph theory, a fundamental question of "graph decomposition", how to reduce a graph into simpler constituent subgraphs. Rather different concepts of decompositions arise for digraphs and bigraphs, giving rise to two important algorithms: the SCC algorithm (§6) for digraphs, and a biconnected component algorithm (§7) for bigraphs.

[3] We have taken this highly suggestive notation from Sedgewick's book [11].

Map Anomalies from the Real World: TIGER maps Political or geographical maps such as Figure 2(a) give rise to important applications of graph algorithms. Students interested in such algorithms may look into an extremely interesting map of the USA called the **TIGER Dataset**. It is freely available from the US Census Bureau [10]. This large data set, meant for census use, contains an underlying map of the entire USA. See Figure 3 for a visualization of the TIGER data.
The map is divided state maps, which are in turn divided into county maps. There are over 3200 counties in the US. Each county has a set of text files representing its map. County boundaries and roads are represented by polygonal lines. The roads, rivers and lakes in a county will subdivide a county into polygonal regions. But the union of all these polygons form the area of the county. *You might think that counties are connected as well as simply-connected.* As usual, the real world defy our expectations. Thus New York County (a.k.a. Manhattan) has at least three connected components because it owns the Statue of Liberty Island and part of Ellis Island. The water surrounding these two outposts of New York County belongs to Hudson County in New Jersey. It follows that Hudson county is not simply-connected (but it is connected). The reason Ellis Island only partially belongs to New York is that the island today is partly landfill. It originally belonged entirely to New York, but the landfill parts belong to New Jersey. The structure of this landfill seems quite complex on the map.
Another interesting anomaly is in Prince William County (tgr51153.zip) in Virginia has two holes corresponding to the town of Manassas (tgr51683.zip) and Manassas Park (tgr51685.zip). But Manassas itself contains a hole that belongs to its surrounding Prince William County. These odd facts arise in our visualization research [2, 14] using TIGER data.

Figure 3: Visualization of TIGER maps

§1. Varieties of Graphs

In this book, "graphs" refer to either directed graphs ("digraphs") or undirected graphs ("bigraphs"). We also use "vertex" and "nodes" interchangeably. Additional graph terminology may be found in Chapter I (Appendix A) for reference.

¶4. Set-Theoretic Notations for Simple Graphs. Many varieties of graph have been studied in the literature. Here, we describe three varieties: *directed graphs, undirected graphs* and *hypergraphs.* A graph G is represented by two sets, V and E called the **vertex set** and **edge set**, respectively. For any set V and integer $k \geqslant 0$, let

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

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

umbra = shade or shadow (in Latin)

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

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

$$\begin{array}{lcl} V^2 = \{(a,a),(a,b),(b,a),(b,b)\} & \Rightarrow & |V^2| = |V|^2 = 2^2 = 4 \\ 2^V = \{\varnothing, \{a\}, \{b\}, \{a,b\}\} & \Rightarrow & |2^V| = 2^{|V|} = 2^2 = 4 \\ \binom{V}{2} = \{\{a,b\}\} & \Rightarrow & \left|\binom{V}{2}\right| = \binom{|V|}{2} = \binom{2}{2} = 1. \end{array}$$

We now define our 3 varieties of graphs as follows:

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

The elements of V are called **vertices**, but the elements of E are called **directed edges** for digraphs, **undirected edges** for bigraphs, and **hyperedges** for hypergraphs. The terminology of "digraph" comes from the directed edges, and similarly, "bigraph" comes from the bi-directional nature of undirected edges.

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 [3] or Bollobás [4] is a basic reference on hypergraphs.

So u–v can mean (u, v) or $\{u, v\}$

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

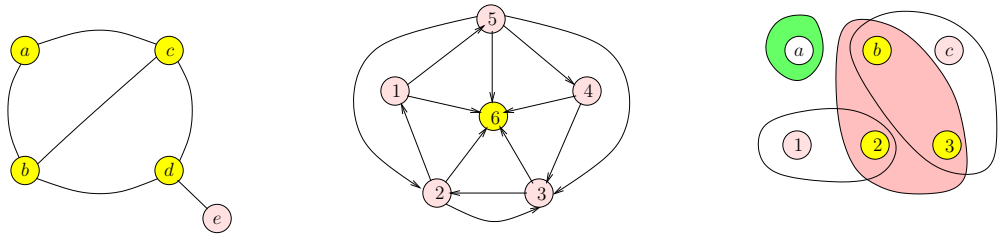
If $G = (V, E)$ and $G' = (V', E')$ are graphs such that $V \subseteq V'$ and $E \subseteq E'$ then we call G a **subgraph** of G'. When $E = E' \cap \binom{V}{2}$ or $E' \cap V^2$ or $E' \cap 2^V$, we call G the subgraph of G' that is **induced by** V.

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

[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.

plane and elements of E are represented by finite curve segments connecting these points. See Figure 4(a) and (b).

For hypergraphs, each hyperedge is represented by a simply-connected bounded of the plane that encloses precisely the vertices of the hyperedge. This **Venn diagram** representation is illustrated in Figure 4(c).



(a) bigraph (b) digraph G_6 (c) hypergraph

Figure 4: Three varieties of graphs

Example 1 In Figure 4(a), we display a bigraph (V, E) where $V = \{a, b, c, d, e\}$ and $E = \{a–b, b–c, c–d, d–b, c–a, d–e\}$. In Figure 4(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 4(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 4(c) we have a hypergraph on $V = \{a, b, c, 1, 2, 3\}$ with four hyperedges $\{a\}, \{1, 2\}, \{b, 2, 3\}$ and $\{b, c, 3\}$. ■

¶6. Non-Simple Graphs. In the literature, the concept of bigraphs, digraphs and hypergraphs might slightly vary from our definition. To distinguish among these variations, we designate our versions to be **simple**. Let us briefly see some "non-simple" versions of these graphs might look like:

- An edge of the form $u–u$ is called a **loop**. For bigraphs, a loop would correspond to a set $\{u, u\} = \{u\}$. But such edges are excluded by our definition of simple bigraph. To allow loops in bigraphs, we must define E as a subset of $\binom{V}{2} \cup \binom{V}{1}$.
- In contrast to simple bigraphs, our simple digraphs allow loops!
- In Figures 1(b) and 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. How can we allow multi-edges? The solution is to view E as a multiset: a **multiset** S is an ordinary set $\underline{S}$ together with a function $\mu : \underline{S} \to \mathbb{N}$. We call $\underline{S}$ the **underlying set** of S and $\mu(x)$ is the **multiplicity** of $x \in \underline{S}$.

  E.g., if $\underline{S} = \{a, b, c\}$ and $\mu(a) = 1, \mu(b) = 2, \mu(c) = 1$, then we could display S as $\{a, b, b, c\}$. So for multisets, $\{a\} \neq \{a, a\} \neq \{a, a, a\}$.

¶7. Some Special Graphs and Hypergraphs. In Chapter I (Appendix A), we introduced some special graphs such as acyclic graphs, trees, the complete graph K_n and the complete bipartite graph $K_{m,n}$. See Figure 5(a,b) for the cases of K_5 and $K_{3,3}$.

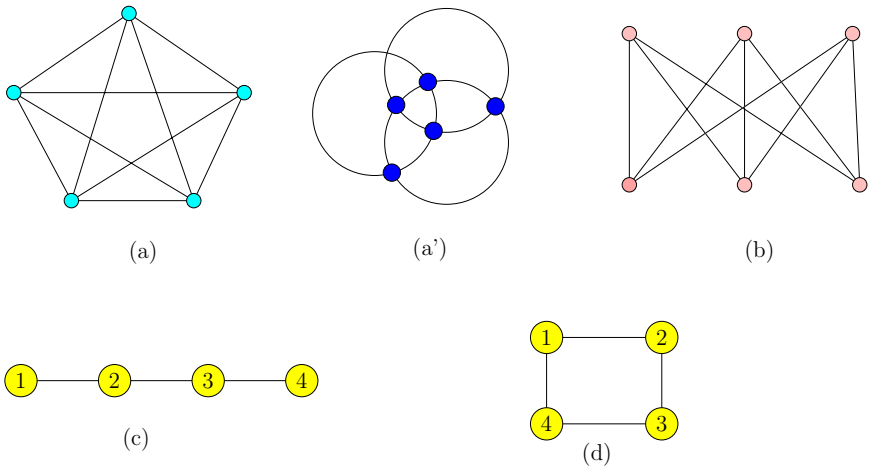


Figure 5: (a) K_5, (a') K_5, (b) $K_{3,3}$, (c) L_4, (d) C_4

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).

Planar graphs are those bigraphs which can be embedded in the Euclidean plane[5] such that no two edges cross each other. 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 (Exercise). Such graphs arise frequently in applications. For instance, in computational geometry, the Voronoi diagram of a set of planar points is a planar graph. The two smallest examples of non-planar bigraphs are the so-called **Kuratowski graphs**, K_5 and $K_{3,3}$ in Figure 5(a,b). We have re-drawn K_5 in Figure 5(a'), this time to minimize the number of edge crossings. The graph $K_{3,3}$ is also known as the "utilities graph". A famous theorem of Kuratowski (1930) says that G is nonplanar if and only if G contains a subgraph G' that is isomorphic to a subdivision[6] of K_5 or of $K_{3,3}$. In general, let K_n denotes the **complete graph** on n vertices and $K_{n,m}$ denotes the **complete bipartite graph** on two sets of vertices of sizes n and m, respectively. By "complete" we mean any possible edge for given set of vertices are included: thus K_n has $\binom{n}{2}$ edges and $K_{m,n}$ has mn edges.

How is $K_{3,3}$ the "utilities graph"?

The notion of planarity for hypergraphs is less straightforward: see four notions in [9, 13].

The Acyclic Hypergraph underlying any bigraph: Let $G = (V, E)$ be a bigraph. Consider the hypergraph $H = (V, A)$ where each set $S \in A$ corresponds to an acyclic subset of G, i.e., the induced subgraph $G|S$ does not contain any cycle. This hypergraph turns out to be a **matroid**, which we will encounter in Chapter 5.

We can also define the **line graphs** L_n whose nodes are $\{1, \ldots, n\}$, with edges i–$i+1$ for

[5] To embed a bigraph $G = (V, E)$ in the plane means the we view $u \in V$ as a point in $\mathbb{R}^2$, and u–$v \in E$ as a simple planar curve connecting u and v. The curves must not intersect each other except possibly at the end points.

[6] A bigraph K' is called a **subdivision** of another bigraph K if (base case) $K = K'$ or (recursively) K' is obtained from a subdivision K'' of K by replacing a single edge $(u$–$v) \in K''$ with two edges $(u$–$w), (w$–$v)$ for some new vertex w. E.g., the m-cycle is a subdivision of the n-cycle for all $m \geqslant n \geqslant 3$.

$i = 1, \ldots, n-1$. 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 5(c,d).

Graph Isomorphism. The concept of graph isomorphism (see Appendix, Chapter 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, Chapter 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 5(a) and Figure 5(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.

¶8. Graphs with Auxiliary Data. We often need to associate additional data with a graph. E.g., we may associate a real number $W(e)$ to each $e \in E$. If $W : E \to \mathbb{R}$ is the weight function, we may call $G = (V, E; W)$ a **weighted graph**. We can also give weights to vertices. Sometimes, we want to designate two vertices $s, t \in V$ as the **source** and **destination**. We may write this graph as $G = (V, E; s, t)$. Thus, in general, we can attach various auxiliary $d_1, d_2, \ldots$ to the graph $G = (V, E)$ and write $G = (V, E; d_1, d_2, \ldots)$. The algorithms in this Chapter will treat graphs without auxiliary data, and we call them "basic algorithms".

Graph Terms in the Literature. Graph vertices have several alternative names in the literature: *node*, *points*, and *dots* (this was used by Coxeter). Likewise, edges have alternative names: *arcs*, *links*, *branches*, *line segments*. These alternatives names are useful in contexts where two or more graph-like structures interact.

Exercises

Exercise 1.1: (Euler) Convince the citizens of Königsberg that there is no way to traverse all seven bridges in Figure 1(a) without going over any bridge twice. ♢

Exercise 1.2: Suppose we have a political map as in Figure 2(a), and its corresponding adjacency relation is a multigraph $G = (V, E)$ where E is a multiset whose underlying set is a subset of $\binom{V}{2}$. Assume that each $v \in V$ is a distinct country, there are no "unclaimed territories" in the map, and no bodies of water are represented. How would you describe a country v such that there exists a unique vertex u such that u–v is an edge. ♢

Exercise 1.3: (Kuratowski Graphs)

(a) Figure 6 shows the **cubic graphs** C_n for $n = 0, \ldots, 4$. The vertices of C_n correspond to vertices of the n-dimensional cube. Note that C_{n+1} is obtained from 2 copies of C_n (one blue, one red), with corresponding blue-red vertices connected by new edges. Clearly C_n are planar graphs for $n \leqslant 2$. Show that C_3 planar, and prove that C_4, also known as the **tessaract**, is non-planar.

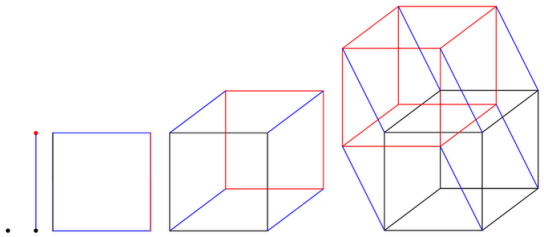


Figure 6: Cubic graphs C_n for $n = 0, 1, 2, 3, 4$. Can you detect a bug in C_4?

(b) Ex.I.10.10 (Appendix A, Chapter I). Show that K_5 is non-planar.

(c) Prove that $K_{3,3}$ is non-planar. HINT: Let the vertices of $K_{3,3}$ be $V = \{a, b, c, a', b', c'\}$ with edge set $\{a, b, c\} \times \{a', b', c'\}$. Consider an embedding of the cycle

$$C : a\text{–}a'\text{–}b\text{–}b'\text{–}c\text{–}c'\text{–}a$$

into the plane.

♢

Exercise 1.4: Prove or disprove: there exists a bigraph $G = (V, E)$ where $|V|$ is odd and the degree of each vertex is odd. ♢

Exercise 1.5:

(a) How many bigraphs, digraphs, hypergraphs are there on the vertex set $V = [1..n]$?

(b) It is much harder to determine the number of non-isomorphic bigraphs, digraphs, hypergraphs on n vertices? There are recursive formulas, but no simple closed formulas. Please enumerate the number non-isomorphic of bi-, di-, hyper-graphs for $n = 1, 2, 3, 4$.

♢

Exercise 1.6: 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, $\cap E = \varnothing$. Show that $|E| \leqslant |V|$. ♢

Exercise 1.7: A hypergraph $G = (V, E)$ is **connected** if it *cannot* be written as a union to two non-empty hypergraphs, $G = G_0 \uplus G_1$ where the vertex sets of G_0, G_1 are disjoint. A **cycle** in G is a sequence $[u_0, e_1, u_1, e_2, u_2, \ldots, u_{k-1}, e_k]$ of alternating vertices u_i and hyperedges e_i such that $u_i \in e_i \cap e_{i+1}$ (assume $e_0 = e_k$). If G is connected, then G has no cycles iff

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

♢

Exercise 1.8: 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 $|V| = n$. Decomposition means a partition 2^V into disjoint sets where each set forms a symmetric chain; the chain is symmetric because the the first and last set in the chain together has size n. For instance, if $V = \{1, 2, 3\}$, then 2^V is decomposed into these 3 symmetric chains:

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

(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 1.9: (Sperner) Let $G = (V, E)$ be a hypergraph with $n = |V|$ vertices. Clearly, $|E| \leqslant 2^n$ 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: Use the symmetric decomposition in the previous Exercise.
(b) Characterize those graphs which attain your upper bound. ♢

Exercise 1.10: A "trigraph" $G = (V, E)$ is a hypergraph where $E \subseteq \binom{V}{3}$. These are also called 3-uniform hypergraphs. Each hyperedge $f \in E$ may also be called a **face**. A pair $\{u, v\} \in \binom{V}{2}$ is called an **edge** provided $\{u, v\} \subseteq f$ for some face f; in this case, we say f is **incident** on e, and e **bound** f. We say the trigraph G is **planar** if we can embed its vertices in the plane such that each face $\{a, b, c\}$ is represented by a simply region in the plane bounded by three arcs connecting the edges a–b, b–c and c–a. Show that G is planar iff its underlying bigraph is planar in the usual sense. ♢

End Exercises

§2. Path Concepts

Most basic concepts of pure graphs revolve around the notion of a path. We expand upon the basic concepts given in an Appendix of Chapter 1.

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

"Adjacency" here is not symmetric!

```
u ← ...                          ◁ u is identified
for each (v adjacent to u)       ◁ loop over each v
      do  "...v...u..."
```

Let $p = (v_0, v_1, , \ldots, v_k)$, $(k \geqslant 0)$ be a sequence of vertices. We call p a **path** if v_i is adjacent to v_{i-1} for all $i = 1, 2, \ldots, k$. The authors [6] writes $p : v_0 \rightsquigarrow v_k$ (equivalently, $v_0 \overset{p}{\rightsquigarrow} v_k$) when

p is a path from v_0 to v_k. We have an alternative way to indicate the same information by extending our edge-notation,

$$p = (v_0 — \cdots — v_k) \quad \text{or} \quad (v_0 —^{p}— v_k)$$

239 Or write $p = (u — \cdots — v — \cdots — w)$ if path p goes from u to v to w.

The **edges** of a path $p = (v_0 — v_1 — \cdots — v_k)$ are $v_{i-1} — v_i$ for $i = 1, \ldots, k$. The **length** of this path is k (not $k+1$). So the length counts the number of edges, not vertices. The path is **trivial** if it has length 0: $p = (v_0)$. Call v_0 is the **source** and v_k the **target** of p. Both v_0 and v_k are **endpoints** of p. We also say p is a path **from** v_0 **to** v_k The path p is **simple** if all its vertices, with the possible exception of $v_0 = v_k$, are distinct. The path p is **closed** if $v_0 = v_k$ and $k > 0$. Any trivial path is simple but not closed (because we require $k > 0$). Thus, a path can[7] only be closed if it has at least one edge. The **reverse** of $p = (v_0 — v_1 — \cdots — v_k)$ is the path

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

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

241 **¶10. The Link Distance.** Define $\lambda^G(u, v)$, or simply $\lambda(u, v)$, to be the minimum length of
242 a path from u to v. If there is no path from u to v, then $\lambda(u, v) = \infty$. We also call $\lambda(u, v)$
243 the **link distance** from u to v. This terminology will be useful after $\lambda(u, v)$ is generalized to
244 weighted graphs, when we still need to refer to the un-generalized concept. The following is
245 easy to see:

distance notation $\lambda(u, v)$

246 - (Non-negativity) $\lambda(u, v) \geqslant 0$, with equality iff $u = v$.
247 - (Triangular Inequality) $\lambda(u, v) \leqslant \lambda(u, w) + \lambda(w, v)$.
248 - (Symmetry) When G is a bigraph, then $\lambda(u, v) = \lambda(v, u)$.

249 These three properties defines $\lambda(u, v)$ as a **metric** on V. Note that the third property restricts
250 G to a bigraph. If $\lambda(u, v) < \infty$, we say v is **reachable from** u**.**

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

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

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

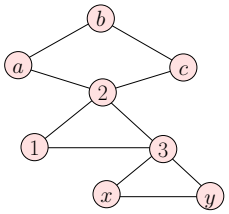
If the target of p equals the source of q, i.e., $v_k = u_0$, then the operation of **concatenation** is well-defined. The concatenation of p and q gives a new path, written

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

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

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

contains q as a subpath if $p = p'; q; p''$ for some p', p''. If in addition, q is a closed path, we can **excise** q from p to obtain the path $p'; p''$. E.g., if $p = (1{-}2{-}a{-}b{-}c{-}2{-}3{-}x{-}y{-}3{-}1)$ and



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

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

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

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

We write $p \equiv q$ in this case. Note that cyclic equivalence is only applicable if p is a closed path because the expression "$r'; r$" is only well-defined if the target of p (i.e., target of r') is equal to the source of p (i.e., source of r). By symmetry, we also see that q must be a closed path.

For instance, let

$$p = (1{-}2{-}3{-}4{-}1) = r; r' = (1{-}2{-}3); (3{-}4{-}1)$$

We see that

$$q = r'; r = (1{-}2{-}3{-}4{-}1) = (3{-}4{-}1{-}2{-}3).$$

Therefore, $p = (1{-}2{-}3{-}4{-}1) \equiv q = (3{-}4{-}1{-}2{-}3)$. The reader may verify that p is cyclic equivalent to these other closed paths (and there are no others):

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

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

We write "$[p]$" to denote the cycle that p is a representative of. Normally when we use the the square bracket notation with an explicit list of vertices,

$$[v_1{-}v_2{-}\cdots{-}v_k] \tag{4}$$

it is denotes the equivalence class of the closed path $(v_1{-}\cdots{-}v_k{-}v_1)$. In other words, we do not write "$[v_1{-}v_2{-}\cdots{-}v_k, v_1]$" in (4). Therefore, when we write "$[p]$", this should be regarded as a slight abuse of notation. Hopefully it is harmless. E.g., the cycle in (3) can be denoted by $[1{-}2{-}3{-}4]$ or $[2{-}3{-}4{-}1]$. However, the associated closed paths implied by $[1{-}2{-}3{-}4]$ and $[2{-}3{-}4{-}1]$ are distinct.

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 the closed path p. E.g., $[1{-}2{-}3{-}4] = [2{-}3{-}4{-}1]$ has length 4.
- Say Z is **simple** if p is simple.

- We may speak of subcycles of Z: if we excise zero or more closed subpaths from a closed path p, we obtain a closed subpath q; call $[q]$ a **subcycle** of $[p]$. In particular, the trivial cycle is a subcycle of Z. For instance, $[1{-}2{-}3]$ is a subcycle of

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

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

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

¶13. Cyclic Digraphs and Bigraphs. Intuitively, a "cyclic graph" is one that contains a cycle. For a digraph G, this is exactly the definition: G is **cyclic** if it contains any cycle. But for bigraphs, this simple definition will not do. To see why, we note that every edge $u{-}v$ in a bigraph gives rise to the cycle $[u, v]$. We shall call $[u, v]$ a "reducible cycle" in this case.

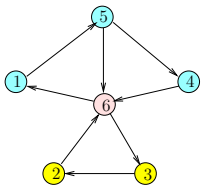
For bigraphs, we proceed as follows: first, define a closed path $p = (v_0{-}v_1{-}\cdots{-}v_k, v_0)$ to be **reducible** if for some $i = 1, \ldots, k$ $(k \geqslant 1)$

$$v_{i-1} = v_{i+1} \tag{5}$$

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

In physics, it is called "hysteresis".

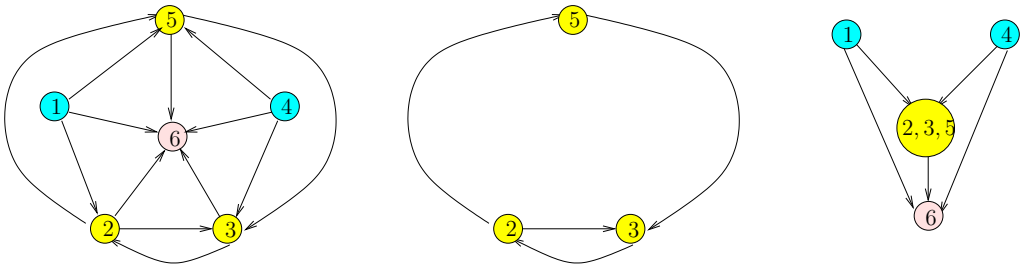
¶14. Strong Connectivity. Let $G = (V, E)$ be a graph. The following definitions applies to both digraphs and bigraphs. Two vertices u, v in G are **connected** if there is a cycle containing both u and v. Note that we do not require the cycle to be simple, so it amounts to having a path from u to v and one from v to u. For instance, the digraph in this margin is connected. However, any cycle Z that contains both vertices 1 and 2 is non-simple since Z must re-use vertex 6. 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**. The subgraph of G induced by C is called a **component graph** of G.



connected digraph

Note that in some literature, it is customary to add the qualifier "strong" when discussing components of digraphs, with the plain "component" is reserved for bigraphs. But our definition

of "component" makes no such distinction. Nevertheless, we might use **strong components** to emphasize the digraph context. Also the term **strongly connected components** (SCC) is commonly used for strong components.



(a) G_6 (b) (c)

Figure 7: (a) Digraph G_6, (b) Component graph of $C = \{2, 3, 5\}$, (c) Reduced graph G_6^c

For example, the graph[8] G_6 in Figure 7(a) has $C = \{2, 3, 5\}$ as a component. The component graph corresponding to C is shown in Figure 7(b). The other components of G are $\{1\}, \{4\}, \{6\}$, all trivial.

Suppose G is a digraph. Its **reduced graph** is the graph

$$G^c = (V^c, E^c)$$

where the vertices in V^c are the components of G, and the edges in E^c comprises $C-C'$ are such that there exists an edge from some vertex in component C to a vertex in C'. This is illustrated in Figure 7(c).

CLAIM: G^c is acyclic. We leave the proof as an exercise.

> *Why was the concept of reduced graph G^c restricted to digraphs?* We could have defined the concept for a bigraph G, but the resulting G^c would always be a trivial graph with no edges! In §7 on biconnectivity below, we will define a suitable notion of **reduced graph** G^c corresponding to any bigraph G.

¶15. 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 Chapter 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

[8] Not to be confused with the digraph in Figure 4.

directing each edge towards the root (so the edges are parent pointers). Given a tree rooted r, the ith **level** of the tree comprises those nodes u with link distance $\lambda(r,u) = i$. Thus, the 0th leven is $\{r\}$, and the 1st level is the set of nodes u such that r–u is a tree edge.

Exercises

Exercise 2.1: Prove that the "cyclic equivalence" relation $p \equiv q$ is an equivalence relation. ♢

Exercise 2.2: 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 2.3: (S. Even) 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 2.4: 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 2.5: Consider this decision problem: given two closed paths $p = [v_1 – \cdots – v_k]$ and $q = [u_1 – \cdots – u_\ell]$, decide whether they are cyclic equivalent. E.g., $p_1 = [1,2,3,4] \equiv p_2 = [3,4,1,2]$ but $p_1 \not\equiv p_3 = [3,4,2,1]$. Again, $p_4 = [1,2,1,3,4,5] \equiv p_5 = [1,3,4,5,1,2]$. Note p_4 and p_5 are *non-simple* closed paths.

(a) Give an algorithm (in pseudo-code) to decide cyclic equivalence. **ASSUME** that vertices are integers, and $p = [v_1 – \cdots – v_k]$ is represented by an array of k integers.
(b) Analyze the worst case complexity of your algorithm.

♢

End Exercises

§3. Graph Representation and Algorithms

¶16. Useful notations for graph. If m, n are integers, we write $[m..n]$ for the set $\{m, m+1, \ldots, n\}$ and $[m..n)$ for the set $\{m, m+1, \ldots, n-1\}$. E.g., when $|V| = n$, we usually represent V by one of two "standard sets" $V = [1..n]$ or "$V = [0..n)$. Many of our arrays are indexed by V. A ubiquitous one is the **parent array** p representing a tree or forest constructed by our graph algorithms. Instead of writing $p[1..n]$, we often write "$p[v \in V]$".

¶17. Three Graph Representations. 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 4 would be

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

and

$$(1\text{–}6, 2\text{–}1, 2\text{–}3, 2\text{–}6, 3\text{–}2, 3\text{–}6, 4\text{–}3, 4\text{–}6, 5\text{–}2, 5\text{–}3, 5\text{–}6).$$

- **Adjacency List**: for each vertex i of G, the **adjacency list** for i is a list of the form

$$(j_1, j_2, \ldots, j_k), \qquad (k \geqslant 0) \tag{6}$$

where $j_1, j_2, \ldots, j_k$ is a listing of the vertices adjacent to i. This list is empty if $k = 0$. We call (6) a **canonical list** if $j_1 < j_2 < \cdots < j_k$ assuming that the vertex set V is totally ordered by $<$. As a data structure, (6) is usually implemented as a singly-linked. Each node in this list has the format discussed in ¶III.4 (On nodes and attributes). In particular, a node u in this adjacency list has two attributes, u.`Next` to point to the next node, and u.`Vertex` $\in V$. The **adjacency list representation** of G will be an array $A[1..n]$ (or $A[V]$ if $V = \{1, \ldots, n\}$) such that each $A[i]$ points to the head of an adjacency list of i. Call A the **adjacency array**. If $A[i]$ points to a canonical adjacenct list then A is[9] the **canonical representation** of G. See Figure 8 for the canonical representation of bigraph G_5.

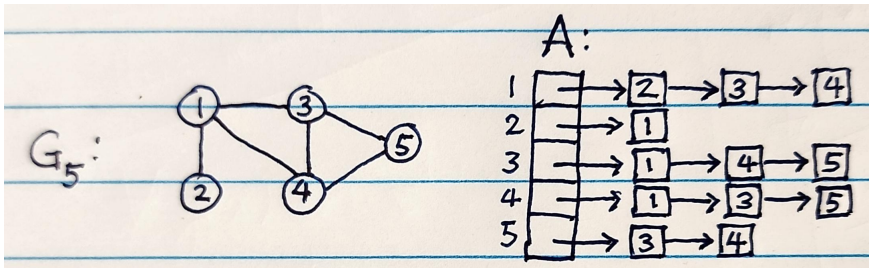


Figure 8: The canonical adjacency list representation for bigraph G_5.

[9] Canonical ordering serves a mainly pedagogical purpose, to ensure unique answers in homework or teaching examples. when students do hand simulations of graph algorithm, we expect them to use the canonical representation of graphs.

Variant Adjacency List Representations. For our biconnectivity algorithm (§7 below) we will use all of the following variations.

- Although $A[i]$ are usually singly-linked list, we may need a doubly-linked list. In that case, each node u of the list will have a extra field u.`Prev` to point to the previous node in the list.
- Sometime, we need to "classify" the edges of a graph. Observe that each node u in the adjacency list $A[i]$ represents the edge i–j where u.`Vertex` $= j$. To classify this edge, we introduce the field u.`Type` to store the classification.
- In general, $A[i]$ might be an empty list, i.e., $A[i]$ = `nil`. In **verbose variant** of the adjacency list $A[i]$, we insist that the head of the list represents i itself. i.e., $A[i]$.`Vertex` $= i$. Thus $A[i]$ is never null in the verbose variant. This verbose variant is strictly speaking unnecessary, on the grounds that we already "know" i if we can access $u = A[i]$. But when we need to manipulate the adjacency list, this verbose form can be useful.

Figure 22(a) (¶49) illustrates the verbose variant G_5 in Figure 8.

- **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 4 are

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

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

Note that the above discussions make sense because we are relying on the fact that the vertices are from $\{a, b, c, \ldots\}$ or $\{1, 2, 3, \ldots\}$ which are "well-known sets" with standard representations. For a general V, we need to first map V to one of these well-known sets by some convention.

¶18. Size Parameters. Two size parameters are used in measuring the input complexity of graph problems: $|V|$ and $|E|$. These are typically denoted by n and m. Thus the running time of graph algorithms are typically denoted by a function of the form $T(n, m)$. A linear time algorithm would have $T(n, m) = \mathcal{O}(m + n)$. It is clear that n, m are not independent, but satisfy the bounds $0 \leqslant m \leqslant n^2$. 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.

An $O(m + n)$ time complexity is considered optimal for graph algorithms.

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) $m \leqslant 3n - 6$ in such graphs.

401 **¶19. Arrays and Attributes.** If A is an array, and $i \leqslant j$ are integers, we write $A[i..j]$ to
402 refer to the subarray of A which are indexed from i to j. Sometimes, we may write "$A[u \in V]$"
403 to indicate that the array A are indexed by the vertices of V.

404 Often, we want to represent an **attribute** (or property) with each vertex. If $V = [1..n]$,
405 we can use an array $A[1..n]$ to store these attributes. For instance, if each vertex has a weight
406 attribute, then $A[i]$ is the weight of vertex i.

¶20. 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 white or unseen or 0. It is best to record this status or color information in an array `color`$[V]$. When we say "color vertex $i \in V$ as `seen`", it amounts to

$$\texttt{color}[i] \leftarrow \texttt{seen}.$$

407 The color transitions of each vertex are always in this order:

$$\begin{array}{ccccc} \texttt{white} & \Rightarrow & \texttt{gray} & \Rightarrow & \texttt{black}, \\ \texttt{unseen} & \Rightarrow & \texttt{seen} & \Rightarrow & \texttt{done} \\ 0 & \Rightarrow & 1 & \Rightarrow & 2. \end{array} \tag{7}$$

408 Sometimes, a two-color scheme[10] is sufficient, and we may omit the `gray` color or the `done`
409 status.

¶21. Destructive or Conservative Graph Routines? The following question is not only relevant for graphs, but for other data structures and throughout the book.
Q: *When designing an library routine to transform an input graph G to some other G', which of the following options is "better"?*
(i) Destructive Routine that directly modify the graph G. Upon return, we have G' but G has been destroyed.
(ii) Conservative Routine that returns G', but the original graph G is preserved.
A: Although the answer ultimately depends on the application, we suggest that the destructive version is better. If the original graph must be
410 preserved, then create a copy of the graph before calling the destructive routine. On the other hand, if you do not care to keep the original graph, then the conservative algorithm wastefully creates a new graph. Of course, you can explicitly delete the old graph. So far, the score seems to be even: the destructive routine may force you to first copy the graph; the conservative routine may force you to delete an unwanted copy at the end. Now a routine with the freedom to destroy the input graph can avoid initializations, and it has opportunities for optimization (such as reusing nodes rather than creating new ones via memory allocation). This seems to be a win for large graphs. See Exercise 3.1 for an concrete example for graph reversal.

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

[10] It is rare that we need a 4-color scheme. But see the exercise below, and also an an exercise in Chapter XIV.

¶22. Canonical Ordering of Edges. In some applications (e.g., Boruvka's Algorithm in Chapter 5), we need some total ordering of the edges. For our hand simulation, it is also useful to impose some ordering of edges so that we can have a canonical answer. We assume that the vertex set V has a total ordering. We extend them to the edge set E as follows:

- G is a digraph: we can totally order $V \times V$ using the lexicographic order: $(a{-}b) \leqslant_{\texttt{LEX}} (a'{-}b')$ iff $a < a'$ or $(a = a'$ and $b \leqslant b')$.
- G is a bigraph: we can totally order $\binom{V}{2}$ by mapping each edge $\{a, b\}$ to the pair $\mu(\{a, b\}) = (a, b)$ where $a < b$. Then the edge $\{a, b\} \leqslant \{a', b'\}$ iff $\mu(\{a, b\}) \leqslant_{\texttt{LEX}} \mu(\{a', b'\})$.
- G is a hypergraph: we can totally order 2^V by mapping each edge $S = \{a_1, \ldots, a_m\} \subseteq V$ to the sorted list $\mu(S) = (a_1, \ldots, a_m)$ where $a_1 < a_2 < \cdots < a_m$. Then $S \leqslant T$ iff $\mu(S) \leqslant_{\texttt{LEX}} \mu(T)$. In general, for two distinct sequences $\boldsymbol{a} \neq \boldsymbol{b}$, their lexicographical order is $\boldsymbol{a} = (a_1, \ldots, a_m) <_{\texttt{LEX}} (b_1, \ldots, b_p) = \boldsymbol{b}$ if there is some $i \leqslant \min(m, p)$ such that $a_j = b_j$ for $j = 1, \ldots, i$, and either $i = m < p$ or $i < \min(m, p)$ and $a_{i+1} < b_{i+1}$. Note that the our ordering on $\binom{V}{2}$ is a special case of this general ordering.

¶23. Shell Programming, again. In §III.4, we introduced shell programming for tree traversals. In shell programming, there is an overall program called the "shell". At critical points in the shell are macro calls (e.g., $\boxed{\text{VISIT}(u)}$ and $\boxed{\text{POSTVISIT}(u)}$). We solve our problem by programming these macros. The shells for tree traversals are generalized to graph traversals. We will introduce the two famous shells for graph traversals: BFS and DFS.

Note that shell programs rely global variables and/or data structures to communicate among the macros. Hence there are macros to initialize these global variables and structures. One limitation of our current paradigm is that our macros do not support "quick exit" from the shell. For example, an algorithm to detect if a graph is connected should be able to exit as soon as it is discovered that the graph has more than one component. This can be done, but it would have to be programmed into the shell. But we forgo such refinements in order to keep our shells simple and generic.

¶24. Simulation Conventions. Students will do hand simulations of our algorithms on graphs. To ensure a unique output, we will enforce a simple convention: whenever there are several choices for a vertex, we pick the one of smallest index that is legal or available. Typically our vertex set are $V = \{1, 2, \ldots, n\}$ or $V = \{a, b, c \ldots\}$ whose sorting order is clear. E.g., in Prim's algorithm or Dijkstra's algorithm, we need to start from a source node s. If this is not explicitly given, then we choose $s = 1$ or $s = a$. Similar, it adjacency lists, this convention will pick the (next) smallest adjacent vertex.

Exercises

Exercise 3.1:

> Graph reversal problem:
> INPUT: a digraph $G = (V, E)$ where $V = [1..n]$.
> OUTPUT: the reverse digraph G^{rev} (see ¶I.A.9).

Assume that the input and output graphs use the array-of-lists version of adjacency list representation (§IV.3). For the adjacency lists, assume each node u in a linked list has two fields, $u.Vertex \in [1..n]$ and $u.Next$ which points to another node (or to `nil`).

One goal of this exercise is to explore the relative powers of "destructive" versus "conservative" algorithms (see ¶IV.21). Use pseudo-code (¶I.A.11) to write your algorithms; for this problem, your pseudo-code should be at the low level of adjacency list manipulation.

(a) Give a conservative algorithm for graph reversal problem. Conservative means you must preserve the input graph G.

Your pseudo code must show how the linked lists are manipulated. To get a new node v, write "$v \leftarrow newNode()$" in the style of `Java` or `C++`. Similarly, use "$B \leftarrow newArray[1..n]$" to get a new array B.

(b) Repeat the part(a) but now, the algorithm is the destructive kind. *Your algorithm must exploit the fact that the input G need not be conserved.*

(c) In what scenario is the destructive algorithm better than the conservative one? Is there another scenario where the destructive algorithm is better?

(d) Show that your algorithms in (a) and (b) have running time $O(n + m)$ where n, m are the usual graph size parameters.

♢

Exercise 3.2: Let $G = (V, E)$ is a connected bigraph. Let $E(G)$ be any embedding of G in the plane. This means that each $u \in V$ is represented by a distinct point $E(u) \in \mathbb{R}^2$, and each edge u–$w \in E$ is represented by a continuous curve viewed as a set $E(u$–$w) \subseteq \mathbb{R}^2$ of points connecting $E(u)$ to $E(w)$. Call $E(G)$ a **planar embedding** if any two curves $E(u$–$w)$ and $(u'$–$w')$ are essentially-disjoint. By **essentially-disjoint** we mean the two curves do not intersect except possibly by sharing common endpoints (e.g., $E(u) = E(u')$). We say G is **planar** if there exists a planar embedding of G. For instance, in Figure 5(a) and (a') are two embeddings of the graph K_5. But neither embeddings are planar: Figure 5(a) has 5 pairs of edges whose embeddings are not essentially-disjoint. Figure 5(a') has only one pair of edges whose embeddings is not essentially-disjoint.

Suppose $E(G)$ is a planar embedding. Then the Euclidean plane is divided by these curves into connected regions called **faces**. All of these faces are bounded except for one, called the **infinite face**. E.g., the graph embedding in Figure 4(a) has 3 faces, while the graph embedding in Figure 4(b) (viewed as a bigraph) has 9 faces. In the following, assume that $E(G)$ is a planar embedding with f faces. Also let $v = |V|$ and $e = |E|$ be the number of vertices and edges respectively. Our goal is to prove Euler's formula $v - e + f = 2$ and derive some consequences.

(a) Show that $e \geqslant v - 1$ (recall that G is connected).

(b) Prove Euler's formula: $v - e + f = 2$.
Remark: this formula shows that the number f of faces is independent of the embedding.
HINT: use induction on e.

(c) Show that $2e \geqslant 3f$. HINT: Count the number of (edge-face) incidences in two ways: by summing over all edges, and by summing over all faces.

(d) Conclude that $e \leqslant 3v - 6$. When is equality attained? Illustrate by drawing an embedding $E(G)$ where $e = 3v - 6$ for some graph with $v = 5$.
Remark: this proves that planar graphs are sparse, in fact $m < 3n$ where m, n are the usual graph size parameters.

♢

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

♢

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

Exercise 3.5: Prove that $K_{3,3}$ is nonplanar. HINT: Use the fact that every face of an embedding of $K_{3,3}$ is incident on at least 4 edges. Then counting the number of $(edge, face)$ incidences in two ways, from the viewpoint of edges, and from the viewpoint of faces. From this, obtain an upper bound on the number of faces, which should contradiction Euler's formula $v - e + f = 2$. ♢

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

End Exercises

§4. Breadth First Search

¶25. Graph Traversals. A **graph traversal** is a systematic method to "visit" each vertex and each edge of a graph. 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[11] about threading through mazes and to Trémaux's cave exploration algorithm in the 19th Century (see [7, 11]). An early paper is E.F. Moore's "The Shortest Path through a Maze", Proceedings of the 1957 International Symposium on Switching Theory, Part II, Harvard U Press, 1959, pp. 285–292. Such explorations are the basis for some popular computer games.

Haven't we seen this before in trees?

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

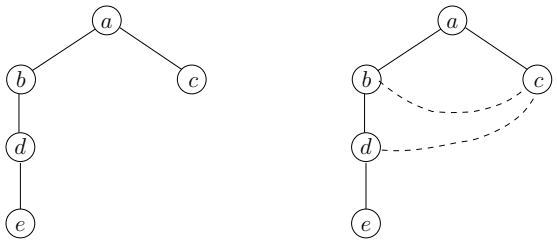
Generic Graph Traversal:
Input: $G = (V, E; s_0)$ where s_0 is any source node
Color all vertices as initially `unseen`.
Mark s_0 as `seen`, and insert into a container ADT Q
While Q is non-empty
    $u \leftarrow Q.Remove()$
    For each vertex v adjacent to u
        If v is `unseen`,
            color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

[11] Theseus was a Greek prince who entered the Labyrinth (maze) while leaving a trail formed by a red thread; upon killing the half-man half-bull Minotaur in the center of the Labyrinth, he escaped by retracing the thread. Of course, he carried off the princess Ariadne who gave him the thread.

This algorithm will reach all nodes that are reachable from the source s_0. Ostensibly, the goal is to "visit" all the nodes that can be reached from s_0. As in the case of tree traversal in §III.4, we introduce macros such as $\boxed{\text{VISIT}(v)}$. 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.

Both traversal methods apply to digraphs as well as bigraphs. However, BFS is typically described for bigraphs only and DFS for digraphs only. 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 4(a). From vertex a, we first visit the vertices b and c which are distance 1 from vertex a. Next, from vertex b, we find vertices c and d, but we only visit vertex d but not vertex c (which had already been visited). And so on. The trace of this search can be represented by a tree as shown in Figure 9(a). It is called the "BFS tree".



(a) (b)

Figure 9: (a) BFS tree. (b) Non-tree edges.

More precisely, recall that $\lambda(u, v)$ 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

$$\lambda(s_0, u) < \lambda(s_0, v) < \infty. \tag{8}$$

If $\lambda(s_0, u) = \infty$, then u will not be visited from s_0. The BFS algorithm does not explicitly compute the relation (8) to decide the next node to visit: below, we will prove that this is a consequence of using the queue data structure.

In this and the next sections we are going to design "shells" for BFS and DFS. By design, these shells explicitly indicate any color changes (`unseen` → `seen` → `done`, $0 \to 1 \to 2$, etc), but all other data manipulations are encapsulated in "boxed" macros calls (e.g., $\boxed{\text{MACRO}(\ldots)}$).

¶27. The BFS Shell. The key to the BFS algorithm is the **queue** data structure (§III.2) This is an ADT that supports the insertion and deletion of items following the First-In First-Out (FIFO) discipline. If Q is a queue, we denote the insert and delete of a node u by

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

respectively. To keep track of the status of vertices we use the color scheme (7) in the previous section. We could use three colors, but for now, two suffice: `unseen`/`seen`.

Here is the BFS algorithm formulated as a shell program.

```
    BFS(G, s_0)
     Input:   Graph G = (V, E) (bi- or di-) and s_0 ∈ V
     Output:  This is application specific.
     ▷ Initialization:
0        [INIT(G, s_0)]   ◁ If this is standalone, INIT should color all the vertices as unseen
1        Color s_0 as seen and initialize a FIFO queue Q ← {s_0}.
2        [VISIT(s_0)]   ◁ Visit the root
     ▷ Main Loop:
         While Q ≠ ∅ do
3            u ← Q.dequeue().   ◁ Begin processing u
4            for each v adjacent to u do   ◁ Process edge u–v
5                [PREVISIT(v, u)]   ◁ Previsit v from u
6                if v is unseen then
7                    Color v seen
8                    [VISIT(v)]
9                    Q.enqueue(v).
10           Color u done and [POSTVISIT(u)]
11       [CLEANUP(G)]
```

This BFS shell program contains the following shell macros

$$\text{INIT, PREVISIT, VISIT, POSTVISIT, CLEANUP} \tag{9}$$

which are application-specific. All these macros may be assumed[12] to be NO-OP operations unless otherwise specified. The term "macro" here suggests only a small[13] and non-iterative piece code that can be executed in $\mathcal{O}(1)$ time.

"macro = small"??

Color Transisions: Except for the initialization in INIT, all color transitions in BFS(V, E, s_0) are explicitly shown (not hidden in the macros). Initially, all vertices are `unseen`. A vertex becomes `seen` just before it enters the queue Q, and becomes `done` after it is removed from Q and we know that each vertex adjacent to it is no longer `unseen`. Let us define `Rank`$(u) = i$ (the **BFS rank** of u) if u is the ith vertex to be `seen`'ed by in the algorithm. Thus `Rank`$(s_0) = 1$, and one of the vertices adjacent to s_0 will have rank 2, etc. Assume `Rank`$(u) = \infty$ if u is not `seen`'ed.
The macro PREVISIT(v, u) has two arguments, and it should be read as "previsiting v **from** u". Alternatively, we can think of macro as "visiting the **edge** $u - v$". Clearly, the color transition of any node $v \neq s_0$ from `unseen` $\rightarrow$ `seen` to preceded by a previsit of v.

[12] 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.

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

The BFS Tree. 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 9(a)). A **BFS listing at** s_0 is a listing of all the vertices which are VISITed if we run the BFS algorithm on $(G; s_0)$. It is easy to see that this is the same as the order in which vertices are first seen (or enqueued). E.g., let G be the bigraph in Figure 4(a) and s_0 is vertex a. Then two possible BFS listing at a are

$$(a, b, c, d, e) \qquad \text{and} \qquad (a, c, b, d, e). \tag{10}$$

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.

¶28. BFS Driver Shell. In our BFS Shell, 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 need a "driver program" that repeatedly calls our BFS algorithm. We assume a global initialization which sets all vertices to `unseen`. Here is the driver program:

```
    BFS Driver(G)
    Input:   G = (V, E) a graph.
    Output:  Application-dependent.
    ▷ Initialization:
1       Color all vertices as unseen.
2       DRIVER_INIT(G)
    ▷ Main Loop:
3       for each vertex v in V do
4           if v is unseen then
5               call BFS((V, E; v)).
6       DRIVER_CLEANUP(G).
```

Note that with the BFS Driver, we add two shell macros: DRIVER_INIT and DRIVER_CLEANUP. 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.

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

(P1) *To print a list (without repetition) of all the vertices reachable from* s_0. You can make VISIT(v, u) print some identifier (key, name, etc) associated with v. This would produce the BFS order at s_0. Alternatively, you can make POSTVISIT(u) print the identifier associated with u. Other macros can remain null operations.

(P2) *Compute the BFS tree* T. Typically, T is represented by a global array $p[v \in V]$ where $p[v] \in V$ is the parent of v in the tree. We can have the convention that $p[v] = v$ iff v is the root. Then INIT(G, s_0) could initialize $p[s_0] \leftarrow s_0$, and for $v \neq s_0$, $p[v] \leftarrow$ `nil`. In PREVISIT(v, u), if v is `unseen` then we set $p[v] \leftarrow u$.

(P3) *Determine the depth $d[u]$ of each vertex u in the BFS tree.* As we will see, this depth has intrinsic meaning in the graph: the depth of u is equal to the link distance $\lambda(s_0, u)$. 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 PREVISIT(v, u), if v is `unseen`, then $d[v] \leftarrow 1 + d[u]$. 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 $d[u] = \infty$.

(P4) *Detecting 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" for the algorithm. Thus, you only reach the final CLEANUP(G) (Step 11) if you did not return earlier through PREVISIT. So, CLEANUP simply returns "ACYCLIC".

> Terence Kelly (ACMQueue, Vol.18, No.4, 2020) noted that for some problems, we can greatly improve the efficiency of BFS algorithm on certain inputs. For instance, in problems (P1-P3) above, we can halt the computation soon after we have visited n nodes. We only need to maintain a counter C to keep track of the number of visited nodes. In the appropriate macro, we can do the exit test: "if $(C = n)$ then HALT".
>
> One way to implement HALT in the `Java` language is to **throw an exception**. We can distinguish two kinds of exceptions – one is to be caught by the main loop of BFS, the other by the main loop of the BFS Driver. This can dramatically speed up the algorithm from $\Omega(n^2)$ to $O(n)$ for certain inputs such as the complete graph K_n.

¶30. BFS Analysis. We shall derive basic properties of the BFS algorithm. These results will apply to both bigraphs and digraphs unless otherwise noted. We have the following basic facts about BFS$(G(V, E; s_0))$:

Lemma 1
(i) The algorithm BFS$(G(V, E; s_0))$ terminate.
(ii) A node $v \in V$ will be VISIT'ed iff v is 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 $\lambda(v) \geqslant 0$ denote the **depth** of a vertex v in the BFS tree. This notation will be justified shortly when we show that $\lambda(v)$ is actually the link distance from s_0 to v. For now, it is just depth in the BFS tree. Note that if v is visited from u, then $\lambda(v) = \lambda(u) + 1$. 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 $k \geqslant 1$, with u_1 the earliest enqueued vertex and u_k the last enqueued vertex. The following invariant holds:*

$$\lambda(u_1) \leqslant \lambda(u_2) \leqslant \cdots \leqslant \lambda(u_k) \leqslant 1 + \lambda(u_1). \tag{11}$$

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 (11) 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

$$\lambda(u_2) \leqslant \lambda(u_3) \leqslant \cdots \leqslant \lambda(u_k) \leqslant 1 + \lambda(u_2).$$

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

$$\lambda(u_2) \leqslant \lambda(u_3) \leqslant \cdots \leqslant \lambda(u_k) \leqslant \lambda(v) \leqslant 1 + \lambda(u_2)$$

holds because $\lambda(v) = 1 + \lambda(u_1) \leqslant 1 + \lambda(u_2)$. In fact, every vertex v enqueued in this for-loop preserves this property. This proves the invariant (11). **Q.E.D.**

This lemma shows that $\lambda(u_i)$ is monotone non-decreasing with increasing index i. Indeed, $\lambda(u_i)$ will remain constant throughout the list, except possibly for a single jump to the next integer. Thus, we call this the "$0-1$ property" since $\lambda(u_{j+1}) - \lambda(u_j)$ is either 0 or 1 for all $j = 1, \ldots, k-1$.

From this lemma, we deduce the first property about the BFS algorithm:

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

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

Proof. By the definition of $\lambda(u)$ as the depth of u in the BFS tree, there is a path of length $\lambda(u)$ from s_0 to u. This proves

$$\lambda(u) \geqslant \lambda(s_0, u). \tag{12}$$

It remains to prove that

$$\lambda(u) \leqslant \lambda(s_0, u). \tag{13}$$

We use induction on $k := \lambda(s_0, u)$. The inequality is clearly true for $k = 0$. Assume $k \geqslant 1$ and let the shortest path from s_0 to u pass through v with $\lambda(s_0, v) = k - 1$. By induction hypothesis, $\lambda(v) \leqslant k - 1 = \lambda(s_0, v)$. By Lemma 1, there is a moment when v is at the front of the queue. Note that (12) implies $\lambda(u) \geqslant k$, and so u cannot appear before v in any BFS listing. Thus u must either be in the queue at this moment, or is still unseen. Now if u is in the queue, then the previous lemma implies $\lambda(u) \leqslant \lambda(v) + 1 = k$, as desired. So suppose u is still unseen. Then, after we pop v, the for-loop that processes all nodes adjacent to v will add u to the queue. This means u is a child of v in the BFS tree, and so $\lambda(u) = \lambda(v) + 1 = k$. Again this proves our inequality. **Q.E.D.**

We conclude that the BFS listing at s_0 gives a list of all the vertices reachable from s_0 *in non-decreasing distances from* s_0.

A useful structure theorem about the BFS tree is the following:

Lemma 4 *The BFS tree satisfies the followign property with respect to BFS Rank:*

- $\texttt{Rank}(s_0) = 1$
- *If v is a child of u in the tree, then* $\texttt{Rank}(v) > \texttt{Rank}(u)$.
- *The ranks of nodes in each level are consecutive integers.*

As a corollory, we have the following: say the depth of the BFS tree is $d \geqslant 0$ and the the size of level $i = 0, \ldots, d$ is ℓ_i. Then $\ell_0 = 1$ and $\ell_i \geqslant 1$. Moreover, the ranks in level i are precisely

$$[L_i, L_{i+1}) = \{L_i, L_i + 1, \ldots, L_{i-1} - 1\}$$

where $L_i = \ell_0 + \ell_1 + \cdots + \ell_i$.

¶31. 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 9(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–c in Figure 9(b).
- **Cross edges**: these are non-tree edges that connect vertices across two different levels. But note that the two levels differ by exactly one. E.g., edge c–d in Figure 9(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 9(b) for tree, level and cross 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.

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

Try proving them without the classification theorem!

1. Let T be a BFS tree rooted at v_0. Consider the DAG D obtained from T by adding all the cross edges. All the edges in G are given a direction which is directed away from v_0 (so each edge goes from some level $i \geqslant 0$ to level $i+1$). CLAIM: *Every minimum link path starting from v_0 appears as a path in the DAG D.* In proof, the classification theorem implies that each path in G is a minimum link path, as there are no edges that can skip a level.
2. Consider a bigraph G with n vertices and with a minimum link path $p = (v_0 – v_1 – \cdots – v_k)$. CLAIM: *If $k > n/2$ then there exists a vertex v_i ($i = 1, \ldots, k-1$) such that every path from v_0 to v_k must pass through v_i.* To see this, consider the BFS tree rooted at v_0. This has more

than $n/2$ levels since $\lambda(v_0, v_k) = k > n/2$. 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.

¶33. 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 $m = |E|$.

Summarizing, the macros in BFS and its Driver are as follows:

BFS Macros:	PREVISIT, VISIT, POSTVISIT, INIT, CLEANUP.
Driver Macros:	DRIVER_INIT, DRIVER_CLEANUP.

(14)

Theorem 6 (Complexity of BFS) *Let the input graph $G = (V, E)$ have parameters $|V| = n$ and $|E| = m$. Assume that the BFS macros takes $O(1)$ time, and the Driver macros take $O(n+m)$ time.*
Then the BFS Driver takes $O(n+m)$ time.

Proof. First consider the BFS algorithm: consider the INIT macro in BFS. If case BFS is a stand-alone program, this macro can take $O(n)$ time. But since it is called by the BFS Driver here, it is not stand-alone and its INIT macro takes only $O(1)$ time. The main loop is $\Theta(m')$ where $m' \leqslant m$ 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. Line 1 $O(n)$ 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 $\Theta(n+m)$. **Q.E.D.**

In other words, BFS has time linear in $n+m$, which is clearly optimal.

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

To accomplish this task, we assume a global variable called `count` that is initialized to 0 by DRIVER_INIT(G). Inside the BFS algorithm, the INIT(G, s_0) macro simply increments the

`count` variable. Finally, the VISIT(v, u) macro is simply the assignment, $C[v] \leftarrow$ `count`. The correctness of this algorithm should be clear. If we want to know the number of components in the graph, we can output the value of `count` at the end of the driver program.

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

¶35. Application: Testing Bipartiteness. A graph $G = (V, E)$ is **bipartite** if V can be partitioned into $V = V_1 \uplus V_2$ such that if u–v is an edge then $u \in V_1$ iff $v \in V_2$. In the following we shall assume G is a bigraph, although the notion of bipartiteness applies to digraphs. It is clear that all cycles in a bipartite graphs must be **even** (i.e., has an even number of edges). The converse is shown in an Exercise: if G has no **odd cycles** then G is bipartite. We use the Driver Driver to call BFS$(V, E; s)$ for various s. It is sufficient to show how to detect odd cycles in the component of s. If there is a level-edge (u, v), then we have found an odd cycle: this cycle comprises the tree path from the root to u, the edge (u–v), and the tree path from v back to the root. In the exercise, we ask you to show that all odd cycles is represented by such level-edges. It is now a simple matter to modify BFS to detect level-edges. Note that this Bipartite Test can benefit from the ability of macros to stop the overall computation as soon as non-bipartiteness is detected.

Exercises

Problems that could be reduced to BFS (also DFS in the next section) *must* be solved using our shell programs. In other words, you only need to expand the various macros. This "straightjacket" approach is also pragmatic — it simplifies the job of your graders. Otherwise, there would many trivial variations of BFS and DFS that is a grading headache.

Students, please note!

Exercise 4.1: Prove Lemma 1. ♢

Exercise 4.2: The BFS tree of a bigraph is shown in Figure 10.

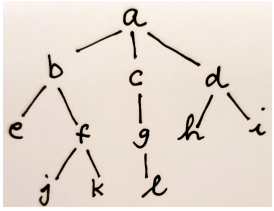


Figure 10: BFS tree

In each part, you are given an instantaneous **state** of the BFS queue Q during its execution. E.g., $Q = (u_0, u_1, \ldots, u_m)$ means that u_0 is the front of the queue and u_m is at the back. In each case, you are to choose YES or NO. YES means the queue could have

arisen during the execution, and NO means it is impossible. You must briefly justify your YES or NO answers. We do not presume any particular ordering within each adjacency list (e.g., the adjacency list of a could by any of the 6 permutations of its children b, c, d.)

(a) $Q = (a, b, c)$
(b) $Q = (b, d, g)$
(c) $Q = (c, e, h)$
(d) $Q = (c, e, f, h)$
(e) $Q = (b, g, h)$

♢

Exercise 4.3: In Figure 10, we assume that the vertices are entered into the BFS queue Q in the order $a, b, c, \ldots, j, k, \ell$. What is the largest size of Q encountered during the BFS algorithm? Please list the contents of Q at those moments of maximum size. ♢

Exercise 4.4: Show that each node is VISITed and POSTVISITed at most once by BFS. Is this true for PREVISIT as well? ♢

Exercise 4.5: In Problem (P1) to print each vertex reachable from s_0 without repetition, we said you could do this in VISIT or POSTVISIT. Do they produce the same listing? Can we print them using PREVISIT? ♢

Exercise 4.6: Level-Listing Problem: Output a list of sublists

$$(S_0, S_1, S_2, \ldots)$$

where S_i is a list of all vertices in depth i of the BFS tree. In particular, $S_0 = (s_0)$ has just the root of the BFS tree. E.g., for the BFS tree in Figure 9(a), we will output $((a), (b, d), (c), (e))$. This problem is a combination of Problems (P1) and P(3).
HINT: Allow the queue to contain a single dummy vertex that separates nodes of different levels. So this falls outside our shell programming paradigm. ♢

Exercise 4.7: Modify the connected components algorithm for bigraphs so that the output array $C[1..n]$ has the property that each $C[u]$ refers to an actual vertex in the connected component of u. ♢

Exercise 4.8: Let $\lambda(u)$ be the depth of u in a BFS tree rooted at s_0. If u–v is an edge, show:
(a) $\lambda(v) \leqslant 1 + \lambda(u)$.
(b) In bigraphs, $|\lambda(u) - \lambda(v)| \leqslant 1$.
(c) In digraphs, $|\lambda(u) - \lambda(v)|$ can be arbitrarily large. ♢

Exercise 4.9: Suppose the BFS queue is $Q = (u_1, \ldots, u_k)$ as in Lemma 2. Characterize the situations when the following is true:

$$\lambda(u_k) - \lambda(u_1) = 1.$$

♢

Exercise 4.10: Let us explore the following algorithm: in the BFS algorithm, suppose we use a stack instead of a queue. In ¶29, we gave several applications of BFS. Which of these applications will remain valid after this modification? ♢

Exercise 4.11: Fill in the shell macros so that the BFS Shell so that it correctly classifies every edge of the input graph G.
HINT: Each node u in the adjacency list of $A[i]$ represents an edge i–j where u.`Vertex` $= j$. As noted in the text, we can record the classification of edge i–j using the field u.`Type`. ♢

Exercise 4.12: Recall that the edges of a bigraph are classified into four types by BFS: tree, cross, level, unseen. We now want to produce a BFS classification of the edges of a digraph. We keep the four types found in bigraphs, but we want to add two new types of edges. These will be back edges u–v that goes from level i to level j with $i > j$. These back edges will be classified as **ancestor** or **non-ancestor** depending on whether v is an ancestor of u or not. (a) Prove that the above BFS classification of the edges of a digraph G into tree, cross, level, unseen, ancestor, non-ancestor is complete (i.e., no other edges arise) and minimal (i.e., each type of edge can arise).
(b) Now turn the classification of part(a) into a "computational classification". I.e., devise an algorithm to classify every edge of G according to (a). Recall that you must use shell programming, and try to be as efficient as possible.
(c) Analyze the complexity of your algorithm in part (b). ♢

Exercise 4.13: Let $G = (V, E; C)$ be a connected bigraph in which each vertex $v \in V$ has an associated value $C(v) \in \mathbb{R}$.
(a) Give an algorithm to compute the sum $\sum_{v \in V} C(v)$.
(b) Give an algorithm to label every edge $e \in E$ with the value $|C(u) - C(v)|$ where $e = u$–v. ♢

Exercise 4.14: Program the BFS shell to determines whether or not a bigraph $G = (V, E)$ contains a cycle. Assume G is connected. 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 4.15: We want an algorithm for testing if a graph is bipartite. Verify the following assertions:
(a) If a bigraph has no odd cycles, then it is bipartite.
(b) 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. Use shell programming. What is the complexity?
(d) Modify the algorithm in (c) so that, in case of YES, it return a Boolean array $B[1..n]$ such that $V_0 = \{i \in V : B[i] = \texttt{false}\}$ and $V_1 = \{i \in V : B[i] = \texttt{true}\}$ is a witness to the bipartiteness of G. In the case of NO, return an odd cycle. ♢

Exercise 4.16: Let G be a digraph. A **global sink** is a node u such that for every node $v \in V$, there is path from v to u. A **global source** is a node u such that for every node $v \in V$, there is path from u to v.

(a) Assume G is a DAG. Give a simple algorithm to detect if G has a global sink and a global source. Your algorithm returns YES if both exists, and returns NO otherwise. Make sure that your algorithm takes $O(m+n)$ time.
(b) Does your algorithm work if G is not a DAG? If not, give a counter example which makes your algorithm fail. ♢

Exercise 4.17: Give an algorithm to count the number of global sinks (as defined in the previous question). HINT: Use shell programming. We are happy with a simple solution with running time $O(n(n+m))$. In a later section, we will be able to solve this in $O(m+n)$. ♢

Exercise 4.18: Let $k \geqslant 1$ be an integer. A **k-coloring** of a bigraph $G = (V, E)$ is a function $c : V \to \{1, 2, \ldots, k\}$ such that for all u–v in E, $c(u) \neq c(v)$. We say G is **k-colorable** if G has a k-coloring. We say G is **k-chromatic** if it is k-colorable but not $(k-1)$-colorable. Thus, a graph is bipartite iff it is 2-colorable.
(a) How do you test the 3-colorability of bigraphs if every vertex has degree $\leqslant 2$?
(b) What is the smallest graph which is not 3-colorable?
(c) The **subdivision** of an edge $u-v$ is the operation where the edge is deleted and replaced by a path $u-w-v$ of length 2 and w is a new vertex. Call G' a subdivision of another graph G if G' is obtained from G be a finite sequence of edge subdivisions. Dirac (1952) shows that G is 4-chromatic, then it contains a subdivision of K_4. Is there a polynomial time to determine if a given connected bigraph G contains a subdivision of K_4? ♢

Exercise 4.19: Let $G = (V, E)$ be a bigraph on n vertices. Suppose $n+1$ is not a multiple of 3. If there exists vertices $u, v \in G$ such that $\lambda(u, v) > n/3$ then there exists two vertices whose removal will disconnect u and v, i.e., $\lambda(u, v)$ will become ∞. ♢

End Exercises

§5. Depth First Search

¶36. The Standard DFS Shell. In the framework of "generic graph traversal" ¶26, the Depth First Search (DFS) differs from BFS only in its use of a stack instead of a queue for visiting nodes. But this simple modification will have profound implications. The standard way to implement DFS does not explicitly use a stack, but uses recursion. Of course, the implementation of recursion implicitly uses a stack. In this recursive form, DFS is deceptively simple:

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

For simplicity, we just say "DFS" to refer to this Standard DFS. There are three things to note:

1. The Standard DFS is a recursive subroutine. It is a shell program that depends on 3 macros:

$$\boxed{\text{VISIT}(u)},\ \boxed{\text{PREVISIT}(v,u)},\ \text{and}\ \boxed{\text{POSTVISIT}(u)}.$$

   We may assume that these macros are No-Op's just to understand the flow of control of this algorithm. We need the macros to accomplish various task-dependent applications of DFS.

2. The vertices of input graph G are assumed to have one of three colors, `unseen`,`seen` or `done`. Equivalently, `white`, `gray`, `black`(see ¶20). The color transitions are correlated with VISIT and POSTVISIT macros:

$$\texttt{unseen} \overset{\text{VISIT}}{\longrightarrow} \texttt{seen} \overset{\text{POSTVISIT}}{\longrightarrow} \texttt{done}. \tag{15}$$

   After VISIT, the color becomes `seen`, and eventually it will be POSTVISITed, and color becomes `done`. Unlike the other 2 macros, PREVISIT(v,u) has two arguments: we say that "*we previsit v from the vertex u*". Note that vertex v may be PREVISITed many times, but VISITed and POSTVISITed at most once. Alternatively, we can also view PREVISIT(v,u) as "visiting the edge u–v".

3. Observe that the color transitions are explicit in the DFS shell, not hidden inside the macros. Otherwise, we could fold the `unseen` → `seen` transition in PREVISIT, and the `seen` → `done` transition in POSTVISIT. That is important because such transitions encode the essence of "DFS" and cannot be left to the macros.

4. Although G is usually a digraph, it works for bigraphs as well. The adjacency array representation is assumed. In case of bigraphs, an undirected edge $\{i,j\}$ is represented twice as i–j and j–i.

Don't by deceived by its simplicity! It is simplicity is hidden by recursion.

¶37. Computing the DFS Tree. We will discover much depth (no pun intended) in the properties of DFS. These properties are inevitably related to the DFS tree. It will also give us an initial insight into the operations of DFS.

The **DFS tree** $T(s_0)$ is associated with a particular execution of the routine DFS$(V,E;s_0)$. It is a tree rooted at s_0 which is a subgraph of $G(V,E)$. Its edges are precisely those i–j such

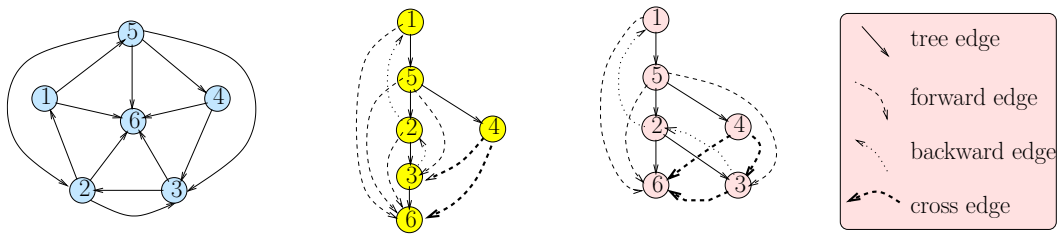
that j is VISIT'ed from i. The tree $T(s_0)$ is not unique for two reasons: it depends on the order in which we scan the adjacency lists, and also depends on the color status of the vertices. *Unless otherwise noted, we assume that all the vertices of V are* `unseen` *in the definition of* $T(s_0)$.

Our first DFS problem is to *construct a representation of* $T(s_0)$. We use a familiar data structure: let $p[i \in V]$ be a global array with the property that $p[i] \neq i$ implies $p[i] \in V$ is the parent of vertex i in $T(s_0)$. Hence $p[i \in V]$ is called **parent array**. As a stand-alone application, we can initialize this array with $p[i] \leftarrow i$ for all $i \neq V$. To compute $p[i \in V]$, we only need to program the PREVISIT macro:

PREVISIT(j, i):
    if (j is `unseen`),
        $p[j] \leftarrow i$.

In other words, if j is first seen from i, then $i-j$ becomes a tree edge, and j is a child of i. When DFS terminates, we obtain an array $p[i \in V]$ representing $T(s_0)$. In this final array, we have $p[s] = s$ iff $s = s_0$ or s is not unreachable from s_0.

Let us illustrate the stand-alone DFS on the digraph[14] G_6 in Figure 11(i). Starting from the source vertex $s_0 = 1$, one possible path to a leaf is (1–5–2–6) as shown in Figure 11(iii). 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 shown in Figure 11(iii). 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 different orders we get different DFS trees. In particular, Figure 11(ii) shows the "canonical DFS tree" if we follow our usual convention of visiting vertices with smaller indices first.



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

Figure 11: Two DFS trees for digraph G_6.

¶38. Two DFS Drivers. Recursive routines will often need to work with a non-recursive routine to set up some global variables before calling the recursive routine. We call the non-recursive routine the **driver** routine. The DFS subroutine is of this nature. In other words, it is not a standalone routine.[15] Moreover, depending on the application, we might call different drivers. Therefore, it is best to speak of a **DFS pair**, consisting of a recursive DFS subroutine *and* a driver routine.

[14] Reproduced from Figure 4(b), for convenience.

[15] For this the reason, we prefer to speak of the "DFS subroutine" rather than the "DFS algorithm". The DFS algorithm should refer to a particular DFS pair.

In the standard DFS, we specify that the graph (G, s_0) has already been colored with `unseen/seen/done`. To ensure this condition before calling DFS, we can provide the following driver:

Not to be confused with Microsoft's cloud storage space called OneDrive.

```
DFS OneDriver(G, s) :
  Input:   G = (V, E) a graph (bi- or di-), s ∈ V
  Output:  Application-specific
1     INIT(G, v)
2     Color each vertex in V as unseen.
3     DFS(G, v)
4     CLEANUP(G, v)
```

Note that OneDriver is a shell program with two macros, $\boxed{\text{INIT}(G, v)}$ and $\boxed{\text{CLEANUP}(G; v)}$ Taken together, we have the first example of a DFS pair:

$$\text{(Standard DFS, OneDriver)} \tag{16}$$

This particular pair is useful for graph problems involving a single source $s_0 \in V$ or one DFS tree rooted at s_0. E.g., if we want to know if there from s_0 to all the nodes in V, we can use this DFS pair. As an exercise, the reader may specify the macros in (16) to compute the π-array of the previous paragraph.

In other applications, we need to ensure that every node is visited, by repeatedly calls to DFS. This can be done by the following alternative driver:

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

This DFS Driver is somewhat more elaborate than the OneDriver. It has the INIT and CLEANUP macros of OneDriver, but needed two more macros DRIVER_INIT and DRIVER_CLEANUP. The DFS Driver has a loop over all the nodes of the graph, to ensure

that every node is visited (and therefore turn from `white` to `black` at the end). Since each call to DFS produces a tree, this driver will produce a DFS forest. We thus have our second example of a DFS pair

(Standard DFS, DFS Driver) (17)

There are a total of 7 macros in this pair, 3 recursive macros, and 4 non-recursive macros:

Recursive:	PREVISIT, VISIT, POSTVISIT
Nonrecursive:	INIT/CLEANUP, DRIVER_INIT/DRIVER_CLEANUP

(18)

The fundamental complexity result about the DFS Driver is captured by this theorem (cf. Theorem 6):

Theorem 7 (Complexity of DFS Driver)
Let the input graph $G = (V, E)$ have parameters $|V| = n$ and $|E| = m$. Assume that the recursive macros takes $O(1)$ time per invocation, and the nonrecursive macros take $O(n + m)$ time.
Then the DFS Driver takes $O(n + m)$ time, and the recursive stack uses $O(m)$ space.

Proof. The total time used by the ith DFS call is proportional to the number n_i of vertices and m_i of edges seen by this call to the DFS. Moreover $\sum_i n_i = n$ and $\sum_i m_i = n$. **Q.E.D.**

In other words, DFS Driver has linear time and linear space complexity. This is considered optimal.

Remark: in some applications, we may break the assumptions of this theorem about the complexity of the macros. E.g., we may allow $\boxed{\text{POSTVISIT}(u)}$ to take time proportional to the size of the adjacency list of u. Clearly, this extension not change the complexity bound in our theorem.

¶39. DFS Tree and the Unseen Path Lemma. We now prove a basic fact about DFS. This lemma answers the question: *when you run DFS(u_0), what is the set of nodes that you visit?* Let us rephrase this as follows: DFS(u_0) eventually produces a DFS tree rooted at u_0. This **DFS tree** is defined as follows: it is rooted at u_0, and its edges are those $u-v$ in which we "visit v from u" (i.e., we call $\boxed{\text{PREVISIT}(v, u)}$ when the node v is `unseen`.) Immediately after this, v would become `seen`. Let T_0 denote this DFS tree. Our question asks for a characterization of the nodes in T_0. The answer depends not only on graph $G = (V, E)$ and $u_0 \in V$, but also on the initial coloring c of the nodes V at the time we run DFS. Note that we call DFS recursively, this c will be different each time.

To discuss the evolving state of the DFS computation, we introduce a time variable t and assume time $t = 0$ when we call DFS(u_0). Let the evolving coloring function be denoted

$$c : V \to \{\texttt{unseen}, \texttt{seen}, \texttt{done}\}\,. \tag{19}$$

At time $t = 0$, we have $c(u_0) =$ `unseen`. Subsequently, the time ticks ($t \leftarrow t + 1$) whenever the color of some node $u \in V$ changes. For our purposes, we are only interested in the change from $c(u) =$ `unseen` to $c(u) =$ `seen`. Let $time(u)$ denote this time instant. If u is not `unseen` at time $t = 0$ or we never visit u, we may declare $time(u) = -1$. We can also view $time(u)$

as the time when the DFS first visits u. For instance, from the definition of DFS we see that $time(u_0) = 1$. Note that at $time(u)$, the recursive stack of the DFS subroutine contains a path $(u_0, u_1, \ldots, u_k)$ with $u_k = u$ at the top of the stack. We shall use this stack in discussing the property of DFS.

Lemma 8 (Unseen Path) *A node $v \in V$ belongs to the DFS tree T_0 if and only if, at the time $t = 0$, there is*[16] *an* `unseen` *path from u_0 to v, i.e., a path $(u_0 — \cdots — v)$ comprising only of unseen nodes.*

Although this lemma seems obvious, its correct proof[17] involves some unusual induction argument. We therefore begin by describing the inductive structure. If v, v' are two nodes adjacent to a nodes u and we visit[18] the edge u–a before u–b then we say

$$a <_{\texttt{dfs}} b \text{ (relative to } u).$$

This ordering is local, concerned only the adjacency list of u. We next turn $<_{\texttt{dfs}}$ into a global ordering on paths.

Let P_0 be the set of **simple `unseen` paths starting from node** u_0. Thus each $p \in P_0$ has the form $p = (u_0 — u_1 — \cdots — u_k)$ where $k \geqslant 0$, and $u_i \neq u_j$ for $i \neq j$. We define a strict total order $<_{\texttt{dfs}}$ on P_0 as follows: if

$$p = (u_0 — u_1 — u_2 — \cdots — u_k), \qquad q = (u_0 — v_1 — v_2 — \cdots — v_\ell)$$

are two distinct paths of P_0, then

$$p <_{\texttt{dfs}} q$$

if p is a prefix of q, or else

$$u_i <_{\texttt{dfs}} v_i \text{ (relative to } u_{i-1})$$

where i is the smallest index such that $u_i \neq v_i$ $(1 \leqslant i < \min\{k, \ell\})$. Thus $<_{\texttt{dfs}}$ is a kind of lexicographic ordering. It is clear that $\leqslant_{\texttt{dfs}}$ is a total ordering on P_0.

If p is a path, let $tar(p)$ denote the target of the path. Let us define two other sets, V_0 and P_0^*:

- $V_0 := tar(P_0) = \{tar(p) : p \in P_0\}$.
- $P_0^* := \{p^*(v) : v \in V_0\}$ where $p^*(v)$ denotes the $\leqslant_{\texttt{dfs}}$-minimal path in P_0 with target v. More precisely, $p^*(v)$ is the unique path satisfying

$$(\forall q \in P_0)[v = tar(q) \Rightarrow p^*(v) \leqslant_{\texttt{dfs}} q].$$

We must be careful to avoid circular arguments in our proofs: this means that we must clearly distinguish between properties about the *static* sets P_0^* and V_0 (defined at time $t = 0$) and the *dynamically constructed* DFS tree T_0. A key static property is the following: the **successor** of $p \in P_0$ is the path $q \in P_0$ such that q is the $\leqslant_{\texttt{dfs}}$-least path subject to $p <_{\texttt{dfs}} q$.

[16] If we use the white-grey-black coloring scheme, this may be called the "white path" as in [6].

[17] For instance, Baase and van Gelder [1] use induction on the length $\ell(v)$ of the shortest unseen path from u_0 to v. Unfortunately, DFS has its own order for visiting nodes that does not always respect the length $\ell(v)$.

[18] I.e., PREVISIT(v, u) is called before PREVISIT(v', u).

Lemma 9 (Basic Lemma) *Suppose $p = (u_0{-}u_1{-}\cdots{-}u_k)$. If q is the successor of p then*

$$q = (u_0{-}u_1{-}\cdots{-}u_i{-}v), \quad (i = 0, \ldots, k)$$

for some v that is adjacent to u_i. Moreover, if $i < k$ then

$$(u_{i+1} < v) \text{ (relative to } u_i).$$

Proof. If p is a prefix of q, then the Lemma is clear. Otherwise, since $p < q$, there is a maximal $i = 0, \ldots, k-1$ such that,

$$(u_0{-}u_1{-}\cdots{-}u_i), \quad (i = 0, \ldots, k)$$

is prefix of q. Let $q = (u_0{-}u_1{-}\cdots{-}u_1{-}v_1{-}\cdots{-}v_\ell)$. But since q is minimal among paths greater than p, we conclude that $\ell = 1$. **Q.E.D.**

The following lemma shows that a dynamic property – the order of visiting nodes in the DFS tree T_0 – agrees with the static "successor" relation. Of course the proof depends on the actual operations of the DFS subroutine:

Lemma 10 (Static characterization of DFS tree T_0)

(a) In DFS, each visited node v is the successor of the previously visited node u.

(b) The node set of the DFS tree T_0 is precisely V_0.

(c) The set of paths of T_0 from u_0 to any node is precisely P_0^*.

Proof. (a) Let the DFS stack be $(u_0, v_1, \ldots, v_k = u)$ when we visited u. Let v be the next visited node. Clearly, v is unseen at $time(u)$. Note that the list of potential successors of u are specified by Lemma 9. We must show that v is the minimum among these candidates. There are two cases: (i) If v is adjacent to u, then v must be the unseen node that is minimal in the local ordering of u. Indeed DFS will correctly choose this v. (ii) Otherwise, v must be adjacent to v_i $(i = 0, \ldots, k-1)$ and $v_{i+1} < v$ (relative to v_i). Again the DFS subroutine chooses the minimal v among these candidates. This proves that v is the successor of u.
(b) The first visited node in T_0 is $u_0 \in V_0$. If sorted list of all node in V_0 is $(u_0 < u_1 < \cdots < u_m)$ then part(a) implies that we visit a prefix of this list. Let the last visited node be u_i. To show that $i = k$, we must prove that u_i has no successor. Let the DFS stack at $time(u_i)$ be $(u_0, v_1, \ldots, v_\ell = u_i)$. According to the DFS algorithm, we terminate the algorithm when we pop off the entire stack because there are no unseen v such that $(v_{j+1} < v)$ (relative to v_j). Again, by Lemma 9, this means that u_i has no successor. This implies $u_i = u_m$.
(c) This follows immediately from part(b). **Q.E.D.**

The Unseen Path Lemma is an easy corollary of the above Characterization of T_0: by definition, there is an unseen path from u_0 to v at time $t = 0$ iff $v \in V_0$. By the Lemma 10(b), $v \in V_0$ iff v is a node of the DFS tree T_0.

¶40. Mathematical Classification of digraph edges. First consider a digraph G. Upon calling $DFS(G, s_0)$, the edges of G becomes classified into five types (see Figure 11):

(A) **Tree edges**: these are the edges belonging to the DFS tree as defined above.

(B) **Back edges**: these are edges u–$v \in E$ where v is an ancestor of u. Note that v might be the parent of u – such a back edge is also called a **parent edge**. E.g., edges 2–1 and 3–2 in Figure 11(iii).

(C) **Forward edges**: these are non-tree edges u–$v \in E$ where v is a descendant of u. A tree edge is not a forward edge, by definition.[19] E.g., edges 1–6 and 5–6 in Figure 11(iii).

(D) **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 11(iii).

(E) **Unseen edges**: all other edges are put in this category. These are edges u–v in which u is unseen at the end of the algorithm. Such edges cannot arise if we use a DFS Driver, which ensures that every edge will be seen.

Next, suppose we call the DFS Driver of a graph. Each call to DFS by the Driver produces a DFS tree. Thus the set of tree edges forms the DFS forest. How does the above edge classification change? First of all, there are no more Type (E) or unseen edges. All unseen edges will be turned into Type (A)-(D) edges, or a new kind of edge:

(F) **Forest edges**: these go from one DFS tree to another.

> Call any classification of the edges of G obtained by running the DFS Driver a **DFS classification of the edges** of G. There are exactly 5 types of edges in this classification (A)-(D),(F). Of course, the classification depends on s_0 and the adjacency list representation.

¶41. Applications of Edge Classification. Let us give a simple illustration of how the ability to classify edges can solve basic graph problems. In many applications, we need to decide if a given digraph is acyclic (i.e., has no cycles). For instance, in the allocation of resources in an operating system, we can construct digraphs whose acyclicity is essential for correctness of the operating system. Thus, the following statement has interest in such applications:

Theorem 11 *A digraph is acyclic iff its DFS forest has no back edges.*

Our theorem is an immediate consequence of the next lemma, which gives the connection between back edges and the cycles. Its proof uses the Unseen Path Lemma.

Lemma 12 *Consider the DFS forest of a digraph G:*
(i) If u–v is a back edge in this forest then G has a unique simple cycle containing u–v.
(ii) If Z is a simple cycle of G then one of the edges of Z is a back edge in the DFS forest.

[19] But we could call a tree edge an **improper** forward edge. Sometimes, it is useful to put consider the improper forward edges with the (proper) forward edges simultaneously, as in the discussion of biconnectivity below.

Proof. (i) is clear: given the back edge $u - v$, we construct the unique cycle comprising the path in the DFS forest from v to u, plus $u - v$. (ii) Conversely, for any simple cycle $Z = [v_1, v_2, \ldots, v_k]$, in the running of the DFS Driver program on G, there is a first instant when we see a node in Z. Wlog, let it be v_1. At this instant, there is an unseen path from v_1 to v_k, namely $(v_1 - \cdots - v_k)$. By the Unseen Path Lemma, each v_k will become a descendant of v_1 in the DFS forest. Then clearly $v_k - v_1$ is a back edge. **Q.E.D.**

Detecting back edges is a special case of the general *computational problem* of classifying all the edges in a DFS forest. We next turn to this problem.

¶42. TimeStamp DFS: algorithic classification of edges. The DFS edge classification in ¶40 is only mathematical (descriptive). We need a computational classification of these edges. The algorithm for this purpose will be called the **TimeStamp DFS**. It is basically the Driver DFS shell with some specific macros.

The output of the TimeStamp DFS is a DFS forest where each vertex v is annotated with a pair of integers $[\texttt{firstTime}(v), \texttt{lastTime}(v)]$. Intuitively, $\texttt{firstTime}(v)$ and $\texttt{lastTime}(v)$ are (respectively) the time of first PREVISIT'ing v and and time of POSTVISIT'ing v. For example, applied to the digraph G_6 in Figure 11(i), we obtain the annotated DFS forest in Figure 12. Of course, the forest is just a tree here.

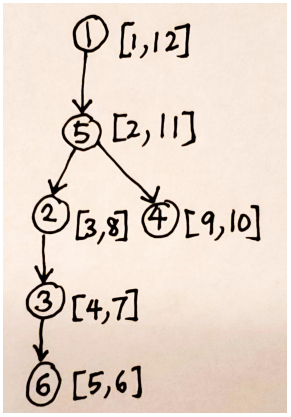


Figure 12: Canonical DFS tree for G_6 with time stamps.

Let us see how to produce these time stamps. We call the DFS Driver with suitable macros and some global data structures. There is a global clock variable `clock`, initialized to 1. We maintain time stamps in two global arrays, $\texttt{firstTime}[v \in V]$ and $\texttt{lastTime}[v \in V]$. Each time after we assign a time to $\texttt{firstTime}[v]$ or $\texttt{lastTime}[v]$, we immediately increment the clock. We also maintain the usual parent array $p[v \in V]$.

Assume that $\texttt{firstTime}[v]$ and $\texttt{lastTime}[v]$ are both initialized to 0 in DRIVER_INIT(G). Here are the macro definitions:

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

During the computation, we have this equivalence:

$$\begin{array}{lcl} v \text{ is } \texttt{unseen} & \Leftrightarrow & \texttt{firstTime}[v] = 0 \\ v \text{ is } \texttt{seen} & \Leftrightarrow & \texttt{firstTime}[v] > \texttt{lastTime}[v] (= 0) \\ v \text{ is } \texttt{done} & \Leftrightarrow & \texttt{firstTime}[v] < \texttt{lastTime}[v] \end{array}$$

¶43. Computational classification of digraph edges. It follows that we do not need to maintain colors explicitly if we have the arrays `firstTime` and `lastTime`. Alternatively, if we maintain colors, we can avoid initialization of these arrays since the color scheme `unseen`/`seen`/`done` can serve to detect initialization conditions.

Let `active`(u) denote the time interval [`firstTime`$[u]$, `lastTime`$[u]$], and we say u is **active** within this interval. It is clear from the nature of the recursion that two active intervals are either disjoint or has a containment relationship. In case of non-containment, we may write `active`(v) $<$ `active`(u) if `lastTime`$[v]$ $<$ `firstTime`$[u]$. We return to the computational classification of the edges of a digraph G relative to a DFS forest on G:

Lemma 13 *Assume that a digraph G has been searched using the DFS Driver, resulting in a complete classification of each edge of G. Let u–v be an edge of G.*

(i) *u–v is a back edge iff* `active`$(u) \subset$ `active`(v).

(ii) *u–v is a cross edge iff* `active`$(v) <$ `active`(u).

(iii) *u–v is a tree edge iff $p[v] = u$.*

(iv) *u–v is a forward edge iff* `active`$(v) \subset$ `active`(u).

The student is asked to verify why a cross edge $u-v$ is characterized as above, and not "`active`$(u) \cap$ `active`$(v) = \varnothing$".

This above classification of edges by active ranges is illustrated in Figure 13.

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

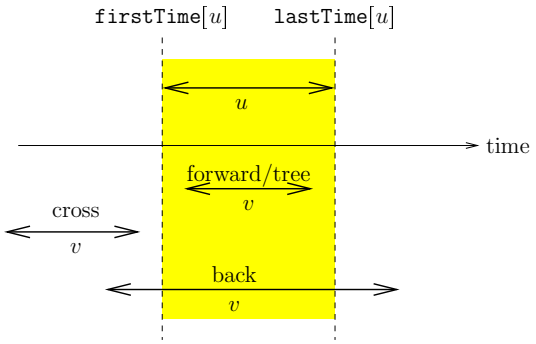


Figure 13: The 3 possible positions of `active`(v) relative to `active`(u), and its implication for classifying edge $(u$–$v)$

PREVISIT(v, u)
▷ *Visiting v, from u*
(T1) if (`firstTime`$[v] = 0$),
mark u–v as "tree edge" (i.e., $p[v] \leftarrow u$)
`firstTime`$[v] \leftarrow$ `clock++`.
(T2) elif (`firstTime`$[v] >$ `firstTime`$[u]$),
mark u–v as "forward edge"
(T3) elif (`lastTime`$[v] = 0$),
mark u–v as "back edge"
(T4) else
mark u–v as "cross edge".

Why is this correct? Intuitively, it is a direct consequence of Lemma 13 (cf. Figure 13). But that is not all: not that we conclude that u–v is forward just based on the test

$$\text{(T2) } \texttt{firstTime}[v] > \texttt{firstTime}[u]. \tag{20}$$

We did not check that `lastTime`$[v] <$ `lastTime`$[u]$ which is implicitly needed by Lemma 13(iv). But this check is unnecessary because we know that that the classification in Lemma 13 is complete (there are no other possibilities). Likewise, to distinguish between back edges from cross edges, the test (T3) suffices.

If the arrays `firstTime`, `lastTime` are not initialized, we could replace the above code as follows: instead of the test `firstTime`$[v] = 0$, we could check if "v is `unseen`". Instead of the test `lastTime`$[v] = 0$, we could check if "v is `seen`" (thus not yet `done`).

¶44. 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 $V = P \uplus R$ is the vertex set and $E \subseteq (P \times R) \cup (R \times P)$. See Figure 14 for an example with 2 processes and 3 resources. An edge $(p, r) \in E \cap P \times R$ means that process

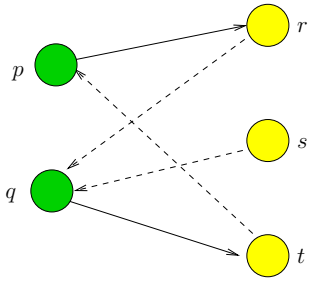


Figure 14: Process-resource Graph: $P = \{p, q\}, R = \{r, s, t\}$.

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**.

We make some additional assumptions about how processes operate. As processes are executed, they issue requests on a set of resources. For instance, to print a file, a process may need to acquire two resources, a file queue and a printer. We assume that a process will be blocked until it has acquired each of the requested resources. Sometime after it has acquired all the resources, the process will release all the acquired resources (in some order). 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 ($k \geqslant 2$) 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 14 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 5.1: (a) Hand simulate the TimeStamp DFS Algorithm on the graph $G10$ in Figure 15. You must start your DFS at node F (not the usual node A). Other than this, please use canonical rules for choosing the next node to visit. For output, imitate the style of Figure 12.

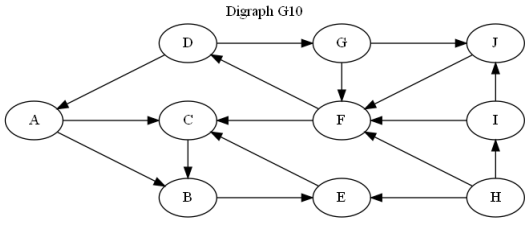


Figure 15: Digraph G10

(b) What is the relation between the TimeStamp DFS and Reckless Ranking Algorithm (¶IV.43)? Specifically, the TimeStamp DFS assigns to each node u the integer $\texttt{lastTime}[u]$ while the Reckless Ranking Algorithm assigns to u the integer $\texttt{Rank}[u]$. How are they related? You can state your claims without proofs, but illustrate with examples.

♢

Exercise 5.2: Why does the following variation of the recursive DFS fail?

```
    VARIANT 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               Variant DFS((V, E; v))
5       [POSTVISIT(s_0)].
6       Color s_0 as seen.
```

♢

Exercise 5.3: In our classification of edges into tree/back/forward/cross edges relative to a DFS search, we used two time stamps. Show that one time stamp suffices. HINT: exploit the fact that we have 3 colors in DFS. ♢

Exercise 5.4:

(a) (Poonyapat Sinpanyalert, 2024) Describe a 2-color DFS algorithm to check if an input digraph G is acyclic. Be sure to prove that your algorithm halts, and when it halts, it correctly outputs either "ACYCLIC" or "CYCLIC".

(b) Construct an input so that your algorithm in part(a) has complexity $\Omega(2^n)$ when G has n vertices.

♢

Exercise 5.5: Exercise 5.6: Let $G = (V, E)$ be a connected bigraph. For any vertex $v \in V$ define

$$\texttt{rad}(v, G) := \max_{u \in V} \delta(u, v)$$

where $\delta(u, v)$ is the length of the shortest (link-distance) path from u to v. The *center* of G is the vertex v_0 such that $\texttt{rad}(v_0, G)$ is minimized. We call $\texttt{rad}(v_0, G)$ the *radius* of G and denote it by $\texttt{rad}(G)$. Define the *diameter* $\texttt{diam}(G)$ of G to be the maximum value of $\delta(u, v)$ where $u, v \in V$.

(a) Prove that $\texttt{rad}(G) \leqslant \texttt{diam}(G) \leqslant 2\texttt{rad}(G)$.
REMARK: $\texttt{diag}(G) \leqslant 2\texttt{rad}(G)$ is equivalent to $\texttt{rad}(G) \geqslant \lceil \texttt{diam}(G)/2 \rceil$, by the Mixed Integer Inequalities of Chap.III.

(b) Show that for all $n \in \mathbb{N}$, there are graphs G_n and H_n such that

$$\begin{cases} n = \texttt{rad}(G_n) = \texttt{diam}(G_n), \\ n = \texttt{diam}(H_n) \text{ and } \texttt{rad}(H_n) = \lceil n/2 \rceil . \end{cases}$$

Thus, the inequalities in (a) are the best possible.

(c) Let G be an free tree (i.e., connected acyclic bigraph). Give an algorithm to compute the diameter of G.
HINT: Use DFS shell. For u any node in the DFS tree, let $u.diam$ and $u.ht$ be the diameter and height of the subtree T_u that is rooted at u. Let $u.ht2$ be the "next largest height" (if u has only one child $h.ht2 = 0$). Thus, $u.ht2 \leqslant u.ht$ (equality is possible). Give recursive formulas for $u.diam, u.ht, u.ht2$.

(d) Give an complexity analysis of your algorithm.

♢

♢

Exercise 5.7: In the previous question, we gave an algorithm to compute the diameter of an acyclic, connected bigraph, i.e., a free tree. Give an algorithm to compute its radius.

♢

Exercise 5.8: Suppose $G = (V, E; \lambda)$ is a strongly connected digraph in which $\lambda : E \to \mathbb{R}_{>0}$. A **potential function** of G is $\phi : V \to \mathbb{R}$ such that for all u–$v \in E$,

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

(a) Consider the cyclic graphs C_n (see Figure 5(d)). Show that if $G = (C_n; \lambda)$ then G does not have a potential function.
(b) Generalize the observation in part (a) to give an easy-to-check property $P(G)$ of G such that G has a potential function iff property $P(G)$ holds.
(c) Give an algorithm to compute a potential function for G iff $P(G)$ holds. You must prove that your algorithm is correct. EXTRA: modify your algorithm to output a "witness" in case $P(G)$ does not hold. ♢

Exercise 5.9: Give an efficient algorithm to detect a deadlock in the process-resource graph.

♢

Exercise 5.10: 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 $E' = E \backslash \{(r, p)\}$.

An **execution sequence** $e = p_1 p_2 p_3 \ldots$ ($p_i \in V_P$) is just a finite or infinite sequence of processes. The **computation path** of e is a sequence of process-resource graphs, $(G_0, G_1, G_2, \ldots)$, of the same length as e, defined as follows: let $G_i = (V_P \cup V_R, E_i)$ where $E_0 = \varnothing$ (empty set) and for $i \geqslant 1$, if p_i is the jth occurrence of the process p_i in e, then G_i is the result of p_i executing its jth instruction on G_{i-1}. If p_i has no jth instruction, we just define $G_i = G_{i-1}$. We say e (and its associated computation path) is **valid** if for each $i = 1, \ldots, m$, the process p_i is not blocked relative to G_{i-1}, and no process occurs

in e more times than the number of instructions in e. A process p is **terminated** in e if p has a finite number of instructions, and p occurs in e for exactly this many times. We say that a set V_P of processes **can deadlock** if some valid computation path contains a graph G_i with deadlock.
(a) Suppose each process in V_P has a finite number of instructions. Give an algorithm to decide if V_P can deadlock. That is, does there exist a valid computation path that contains a deadlock?
(b) A process is **cyclic** if it has an infinite number of instructions and there exists an integer $n > 0$ such that the ith instruction and the $(i + n)$th instruction are identical for all $i \geqslant 0$. Give an algorithm to decide if V_P can deadlock where V_P consists of two cyclic processes. ♢

Exercise 5.11: Continue with the previous model of processes and resources. In this question, we refine our concept of resources. With each resource r, we have a positive integer $N(r)$ which represents the number of copies of r. So when a process requests a resource r, the process does not block unless the outdegree of r is equal to $N(r)$. Redo the previous problem in this new setting. ♢

End Exercises

§6. Topological Sorting and Strong Components

We now address the fundamental problem of decomposing a graph G into simpler constituent parts, and the relationships among the constituent paths (usually represented as a reduced graph G^c). The constituents are known as "strong components" for digraphs, and as "biconnected components" for bigraphs. The latter turns out to be a lot more subtle, and will be separately treated in the next section (§7).

In this section, assume $G = (V, E)$ is a digraph with $V = \{1, 2, \ldots, n\}$. We address two fundamental problems: topological sorting of G and computing connected components of G. In fact, we will reduce connected components to a certain "reckless" topological sorting. Recall that the connected components of G are also called strongly connected components or SCC's.

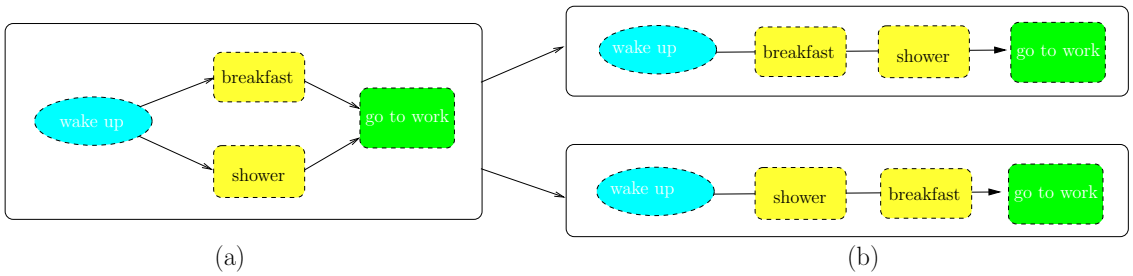


Figure 16: (a) PERT graph, (b) Two possible sorts

¶45. Topological Sort. One motivation is the so-called[20] PERT graphs: in their simplest form, these are DAG's where vertices represent activities. Figure 16 shows such a graph with four morning activities. An edge $u{-}v \in E$ means that activity u must be performed before activity v. For instance, in Figure 16, it is necessary to wake up before having breakfast or showering; and before going to work, you have to have showered and taken breakfast. By transitivity, if there is a path from u to v, then u must be performed before v. A topological sort of such a graph amounts to a feasible order of execution of all these activities. Figure 16(b) shows the two possible ways to topologically sort the four activities.

Really?

Another application arises in a famous Unix utility called the `Make program`. This program (or its many derivatives) is the basis of maintaining modern software libraries. Modern libraries have many components, and there is an acyclic dependency structure. The `Make program` is given these dependencies and knows how to automatically recompile any component whose dependencies have been updated.

Let

$$(v_1, v_2, \ldots, v_n) \tag{21}$$

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

If an edge $u{-}v$ is interpreted as saying "activity u must precede activity v", then a topological sort gives us one valid order for doing these activities.

The **ranking** of a graph $G = (V, E)$ refers to a bijection

$$\texttt{Rank} : V \to \{1, \ldots, n\}$$

where $|V| = n$. Of course, if $V = \{1, \ldots, n\}$ then `Rank` is just an n-permutation. Moreover, we will use an array $\texttt{Rank}[v \in V]$ to represent the ranking function. To reflect the semantic distinction between vertices (which is a symbol) and ranks (which is an integer), we usually use the variables u, v for vertices and variables i, j for ranks. E.g., $\texttt{Rank}[v] = i$.

the array $\texttt{Rank}[v \in V]$

The **inverse rank function** is $\texttt{Rank}^{-1} : \{1, \ldots, n\} \to V$. We use the array $\texttt{iRank}[1..n]$ to represent this inverse. Thus

$$\texttt{iRank}[i] = v \quad \Longleftrightarrow \quad \texttt{Rank}[v] = i \tag{22}$$

the array $\texttt{iRank}[1..n]$

We freely use either `Rank` and `iRank`, whichever is more convenient. This is justified since we can convert between the two arrays in $O(n)$ time. E.g., If the topological sort sequence (v_3, v_1, v_2, v_4) can be represented by either the array $\texttt{Rank}[v_1, v_2, v_3, v_4] = [2, 3, 1, 4]$ or the array $\texttt{iRank}[1, 2, 3, 4] = [v_3, v_1, v_2, v_4]$. **Careful**: do not confuse the inverse rank with the **reverse rank** which is defined by $\texttt{revRank}[v] = n + 1 - \texttt{Rank}[v]$ $(v \in V)$.

Note that any topological sort such as (21) induces a ranking of G where the vertex v_i in (21) is given the **rank** i. Such a rank function is called a **topological ranking**. Thus the topological sort amounts to computing a topological ranking.

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

¶46. Topological Ranking Algorithm. The algorithm to compute `Rank` simply calls the DFS Driver by defining two of macros as follows.

Note that we do not need to initialize the `Rank` array (since every vertex will eventually be visited). But we need to initialize a rank counter denoted R:

$$\boxed{\text{DRIVER\_INIT}(G)} \equiv \big[R \leftarrow n\big] \tag{23}$$

We assign a rank to vertxx v during the POSTVISIT of v: it gets the current value of the rank counter R.

$$\boxed{\text{POSTVISIT}(v)} \equiv \big[\texttt{Rank}[v] \leftarrow R\texttt{--}\ \big] \tag{24}$$

We immediately decrement R so that the next vertex will get a different rank. Inductively, by the time we assign a rank to v, all the proper descendants of v would have already received ranks. Then $\texttt{Rank}[v]$ would be less than those of v's descendants, as expected in topological sorting.

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

- **Responsible Ranking Algorithm**: as soon as the DFS Driver detects that G is not a DAG, it aborts and outputs "G is non-DAG". Recall that G is not a DAG iff it has a back edge, which we can easily modify the above algorithm to detect (¶43). Since our current shell framework does not allow our macros to abort, we can just set a flag to indicate this. But it is easy to implement the ability to abort.
- **Reckless Ranking Algorithm**: Do not attempt to detect whether G is a DAG. Run the DFS Driver to its logical conclusion, and output the `Rank` array. Although this array may not represent a topological sort, its output may nevertheless be useful, as we shall see next.

In our application of the Reckless Ranking Algorithm, we prefer to compute the `iRank` array instead of the `Rank` array. To do this, we modify the above POSTVISIT macro in (24) thus:

$$\boxed{\text{POSTVISIT}(v)} \equiv \big[\texttt{iRank}[R\texttt{--}] \leftarrow v\big] \tag{25}$$

Summary: topological sorting reduces to computing a `Rank` (equivalently, `iRank`) array. This array is computed by calling the DFS Driver program on the input digraph G, using the macros $\boxed{\text{DRIVER\_INIT}(G)}$ in (23) and $\boxed{\text{POSTVISIT}(v)}$ as in (24) or in (25).

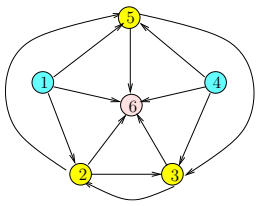
¶47. Strong Component Algorithm. We now address the problem of computing the strong components (SCC's) of a digraph. There are at least three distinct algorithms in the literature. Here, we will develop the version based on "reverse graph search" from Kosaraju and Sharir. The strong components form a partition of the vertex set.

Let $G = (V, E)$ be a digraph where $V = [1..n]$. Previously in ¶38, we defined a DFS Driver(G). We now define a variant DFS Driver that takes an `iRank` array as additional argument. Although $\texttt{iRank}[1..n]$ is meant to be an inverse ranking, we may temporarily think

of it as an arbitrary n-permutation. Our SCC Driver is like the usual DFS Driver program, except that we use $\texttt{iRank}[i]$ to determine the choice of the next vertex to visit:

```
SCC Driver(G, iRank)
 Input: digraph G = (V, E) with V = [1..n], and array iRank[1..n].
 Output: The array p[v ∈ V] representing a forest
▷ Driver Initialization
1       For v = 1, ..., n,
            color[v] ←unseen.
▷ Main Loop
2       For i = 1, ..., n,
3           If (iRank[i] is unseen)
4               DFS(G, iRank[i])    ◁ Computes DFS tree rooted at iRank[i]
```

Each call to $\text{DFS}(G, v)$ in the SCC Driver will produce in the $p[v \in V]$ array a DFS tree rooted at v. **Convention** *Assume that the root v of each DFS tree has the property that $p[v] = v$.* At the end of the Driver call, the p-array represents a forest. So far, the name "SCC Driver" has nothing to do with Strong Connected Components. But its intended application is that, if the array iRank is properly chosen, $p[1..n]$ represents a forest in which each tree represents exactly one SCC. We call such a forest a **SCC Forest**.



digraph G_6

Example 2 Let us see the operations of the SCC Driver on the digraph G_6 in Figure 7(a). Suppose the iRank array is

$$\begin{aligned} \texttt{iRank}[1,2,3,4,5,6] &= [6,2,3,5,4,1] \\ &= [v_6, v_2, v_3, v_5, v_4, v_1]. \end{aligned} \tag{26}$$

Then we see that the SCC Driver will call DFS four times. The (source) vertices given to DFS in the four calls are (in this order):

$$v_6, v_2, v_4, v_1$$

Successive calls will visit these sets of vertices (respectively):

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

Since these are the four strong components of G_6, the array $pi[v_1, \ldots, v_6]$ will be correctly computed.

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

$$\texttt{iRank}[1,2,3,4,5,6] = [1,2,3,4,5,6], \tag{27}$$

our SCC Driver will call DFS twice: first on the vertex v_1 and next on v_4. The resulting pi array represents a forest with two DFS trees. This is clearly not a SCC forest. ■

Relative to the SCC Driver, the central question becomes "Which iRank arrays lead to a SCC forest?". Let us focus on ranking rather than inverse ranking (they are equivalent, but ranking seems more natural). So, which ranking of G Rank leads to to an SCC forest? Call such a ranking **good**. To understand the issues, consider the case where G is a DAG. In this case, we know that the *reverse* of a topological ranking of G is good (Exercise).

In general, G is not a DAG. So we must use the Reckless Ranking Algorithm to produce a ranking, then reverse it. Does this really work? The correct answer turns out to be slightly unintuitive: we first reverse G, and then compute its ranking. We will prove that this ranking is good. Armed with this knowledge, we now have a SCC algorithm:

SCC Algorithm(G)
INPUT: Digraph $G = (V, E)$, $V = [1..n]$.
OUTPUT: Array $p[v \in V]$ represeing a SCC forest of G
1. Compute the reverse graph G^{rev}.
2. Call the Reckless Ranking Algorithm on G^{rev}.
This returns an inverse ranking $\texttt{iRank}[1..n]$.
3. Call SCC Driver(G, $\texttt{iRank}$)

Example 3 Let us continue with the graph G_6 of Figure 7(a) in our previouse example. We call the Reckless Ranking Algorithm on reverse of G_6: doing the canonical simulation, starting with with $DFS(v_1)$. Since v_1 is a sink in the reverse of G_6, and so the DFS tree has just one node v_1 with rank 6. Next, $DFS(v_2)$ will produce the DFS tree with nodes v_2, v_3, v_4, v_5. Their ranks will be $2, 3, 5, 4$ respectively. Finally, $DFS(v_6)$ will give v_6 a rank of 1. This yields the ranking and inverse ranking

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

This is precisely the inverse ranking in (26). We saw that it led to a correct output. ■

¶48. Analysis of the SCC Algorithm. The inverse ranking (26) is considered "good" since it allowed the SCC Driver to compute the SCC forest. Let us characterize this notion of a "good ranking" for any digraph G.

Let $\texttt{Rank}[v \in V]$ be any ranking of the vertices of a digraph $G = (V, E)$. If $C \subseteq V$ is a strong component of G, define

$$\texttt{Rank}(C) := \min\{Rank(v) : v \in C\}$$

Clearly, this induces a ranking of the reduced graph $G^c = (V^c, E^c)$ where V_c are the strong components of G. E.g., if $V^c = \{C_1, C_2, C_3\}$ and

$$\texttt{Rank}(C_1) = 3, \texttt{Rank}(C_2) = 18, \texttt{Rank}(C_3) = 7$$

then we can "renumber" the ranks so that

$$\texttt{Rank}(C_1) = 1, \texttt{Rank}(C_2) = 3, \texttt{Rank}(C_3) = 2.$$

We say that $\texttt{Rank}[v \in V]$ is **good** if, for any two strong components C, C' of G, if there is a path from C to C', then

$$\texttt{Rank}(C) > \texttt{Rank}(C').$$

Lemma 14 *A good ranking function for a digraph G induces a reverse topological ranking on its reduced graph $G^c = (V^c, E^c)$.*

Proof. Let G^c have m components,

$$C_1, C_2, \ldots, C_m \tag{28}$$

listed in increasing order of their induced ranking: $\texttt{Rank}(C_1) < \texttt{Rank}(C_2) < \texttt{Rank}(C_3) < \ldots$. Since Rank is a "good ranking", for any edge $C_i - C_j$, we have $\texttt{Rank}(C_i) > \texttt{Rank}(C_j)$. That means $i > j$. Thus (28) is a reverse topological sort of G^c. **Q.E.D.**

Clearly, our SCC Driver gives the correct output iff the given permutation is good. This is because it will visit C' before it comes to C.

Lemma 15 *Let C, C' be two distinct strong components of G, and* Rank *be a good ranking of G.*
(a) If $u_0 \in C$ is the first vertex in C that is seen, then $\texttt{Rank}[u_0] = \texttt{Rank}[C]$.
(b) If there is path from C to C' in the reduced graph of G, then $\texttt{Rank}[C] < \texttt{Rank}[C']$.

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

History. Tarjan [12] gave the first linear time algorithm for strong components. R. Kosaraju (unpublished, 1978) and M. Sharir (1981) independently discovered the reverse graph search method described here. The reverse graph search is conceptually simple. But since it requires the computation of the reverse graph and 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 its history, see Sedgewick [11].

Exercises

Exercise 6.1: (a) Using the DFS Driver shell provide macros to *simultaneously* compute both the Rank and inverse Rank arrays: $\texttt{Rank}[1..n]$ and $\texttt{iRank}[1..n]$.
NOTE: you must *not* compute only one of the arrays, and then compute the second array using a second pass.
(b) Code up this shell program in your favorite programming language. ♢

Exercise 6.2: (a) List the strong connected components (SCC's) of the digraph G10 in Figure 15.

(b) Run the TimeStamp DFS algorithm on the graph G10 in Figure 15. Draw the **canonical** DFS forest produced by the algorithm. For each node u, you must write `firstTime`$[u]$ and `lastTime`$[u]$ inside u.

HINT: Please begin `firstTime` with 1 (not 0). "Canonical" means we always pick the lowest numbered node among the available nodes. Thus we begin the TimeStamp DFS at node A, next visit B, etc.

(c) Classify the non-tree edges of part(b).

♢

Exercise 6.3: Repeat the previous question, using digraph G10b in Figure 17.

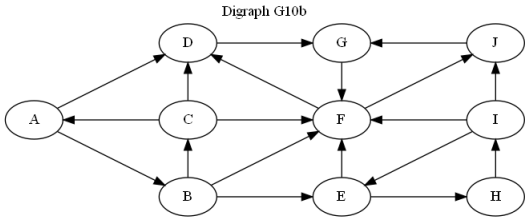


Figure 17: Digraph G10b

♢

Exercise 6.4: A **ranking** for digraph $G = (V, E)$ is a bijection `Rank` : $V \to [1..n]$. Since it is a bijection, it is equivalent to the inverse Ranking function `iRank` : $[1..n] \to V$ (see ¶IV.45, page 45). The ranking is **good** if the algorithm SCC(G, `iRank`) (a.k.a. DFS Driver(G, `iRank`)) outputs a DFS forest in which each DFS tree is a strong component of G.

How many good rankings does the graph $G10b$ in Figure 17 have? Justify your calculations. ♢

Exercise 6.5: Hand simulation of the SCC algorithm on the graph $G9$ in Figure 18.

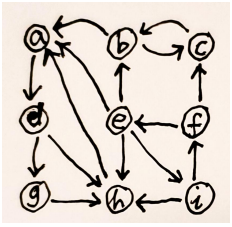


Figure 18: Digraph $G9$

(a) Draw the reduced graph $G9^c$ of $G9$. Recall for any digraph G, its reduced graph is denoted G^c (see ¶14).

(b) Run the Strong Component algorithm on $G9$. There is a unique output, based on our convention for choosing which node to visit next. Please show intermediate results: (i) The reverse graph, (ii) Output of Reckless Ranking algorithm, (iii) The DFS forest (you may ignore forward and back edges, but don't omit the forest edges).

♢

Exercise 6.6: A matrix A of order $n \geqslant 1$ is said to be **reducible** iff $A = P^{-1}BP$ where P is a permutation matrix and $B = \begin{bmatrix} B_{11} & B_{12} \\ \mathbf{0} & B_{22} \end{bmatrix}$ where B_{ii} are square non-zero matrices of order $n_i \geqslant 1$ and $n_1 + n_2 = n$. Otherwise, A is said to be **irreducible**.

Let $G_A = (V, E)$ be the digraph with $V = [1..n]$ and i–$j \in E$ iff $A_{ij} \neq 0$. Prove that A is irreducible iff G_A is strongly connected.

♢

Exercise 6.7: Give an algorithm to compute the number $N[v]$ of maximal distinct paths originating from each vertex v of a DAG. Thus $N[v] = 1$ if v is a sink, and if u–v is an edge, $N[u] \geqslant N[v]$. ♢

Exercise 6.8: Consider this proposed SCC algorithm, which has the advantage of not having to compute G^{rev}:

```
    Faster SCC Algorithm(G)
     Input: Digraph G = (V, E), V = [1..n].
     Output: A SCC forest represented by the p array.
1.       Compute the topological rank of G
                ◁ Returns Rank : V → [1..n]
2.       ▷ Compute the inverse of the reverse ranking:
         for v ∈ V,
              iRevRank[n − Rank[v] + 1] ← v.
                ◁ Since revRank[v] = n − Rank[v] + 1
3.       Call SCC Driver(G, iRevRank)
```

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

Exercise 6.9: Our SCC Driver on input $(G, \texttt{iRank})$ produces an array $pi[v \in V]$ that is a spanning forest for G. If `iRank` is nice, then this is a SCC Forest. However, to represent the SCC graph G^c, we need to know the edges between each SCC. Modify our SCC Driver to compute this extra information.

The question requires a representation of the graph G^c. Please describe your representation and as usual, program the shell macros to compute them. ♢

Exercise 6.10: ♢

Exercise 6.11: Consider the following alternative for computing strong components. The input digraph is G and we will also need its reverse graph G^r. We use an auxiliary

stack S; but this is a "powerful stack" in which we can delete items from S that are not necessarily at the top of stack. Such an S is easily implemented using a doubly-linked list.

```
0.  Initially, color all nodes of G as unseen.
1.  While there is an unseen node u in G
2.      Do DFS on (G, u), and push nodes into S at finish time
3.  While S is non-empty
4.      Pop u from S and run DFS on (G^r, u).
5.      Each vertex visited by DFS is removed from stack
6.          and output as the current strong component.
```

(a) Run this algorithm on the graph in Figure 19, starting with the vertex 1.
(b) Prove the correctness of this algorithm.
(c) What is the relation between this algorithm and the one in the text?

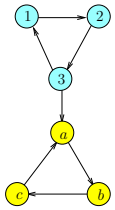


Figure 19: Digraph with two strong components

♢

Exercise 6.12: Suppose we apply our concept of biconnected components to digraphs. What is the corresponding notion of reduced digraph? ♢

Exercise 6.13: An edge u–v is **inessential** if there exists a $w \in V \setminus \{u, v\}$ such that there is a path from u to w and a path from w to v. Otherwise, we say the edge is **essential**. Give an algorithm to compute the essential edges of a DAG. ♢

Exercise 6.14: 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 $h(v) \geqslant 0$, and each pair u, v of adjacent vertices of M gives rise to a directed edge u–v if $h(u) > h(v)$. Note that if the heights are all distinct, the resulting graph is a DAG. If $h(u) = h(v)$, we can arbitrarily pick one direction for the edge, as long as the graph remain a DAG. This is the DAG G_0 in our problem above. Suppose now we have two height functions h_0 and h_1, and we want to interpolate them: for each $t \in [0, 1]$, let $h_t(v) = th_0(v) + (1 - t)h_1(v)$. We want to represent the transformation from h_0 to h_1 by a sequence of graphs, where each successive graph is obtained by changing the direction of one edge. ♢

Exercise 6.15: 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 6.16: 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, but $O(m+n)$ is possible. ♢

Exercise 6.17: In the previous problem, we defined bottlenecks. Now we want to classify these bottlenecks into "real" and "apparent" bottlenecks. A bottleneck u is "apparent" if there exists an ancestor v ($\neq u$) and a descendant w ($\neq u$) such that v–w is an edge. Such an edge v–w is called a by-pass for u. Give an efficient algorithm to detect all real bottlenecks of a DAG G. HINT: This can be done in $O(n + m \log n)$ time. ♢

Exercise 6.18: Given a DAG G, let $D[u]$ denote the number of descendants of u. Can we compute $D[u]$ for all $u \in V$ in $o((m+n)n)$ time, i.e., faster than the obvious solution? ♢

End Exercises

§7. Biconnectivity

We previously defined the reduced digraph G^c where G is a digraph (¶14). Our goal in this section is to introduce a corresponding concept for bigraphs. But first we make a simplifying assumption.

> *In this section, assume the bigraph G is connected.* This is without loss of generality since there are no edges between distinct connected components of bigraphs, and so we can just focus on each component. When G is connected, the reduced bigraph (still) denoted G^c that gives the "fine structure" of G, based on the concept of biconnectivity. Biconnectivity is also known as 2-**connectivity** in the literature. We follow the approach of Jens M.Schmidt (Information Processing Letters, 113:7, pp. 241–244, 2013). This approach reveals new structural properties of biconnectivity that are are hidden in previous algorithms.

Example 4 **Reindeer graph G_{deer}.**
Refer to Figure 20. The bigraph G_{deer} is not connected (temporarily violating our assumption about connectivity). That is in order to illustrate an isolated vertex component. This bigraph has 10 biconnected components: 3 "blocks" (head, front and back torso), and 7 "singular components" (eye, antler, neck, front and back legs, nexus, tail). These concepts will be defined below, but it is clear this "fine structure" of bigraphs is non-obvious. We initially want to define the notion of G^c, the reduced graph corresponding to any bigraph. For our example, G^c_{deer} is shown in Figure 23 below. ■

¶49. Classification of bigraph edges by DFS. Begin with the naive view of a bigraph $G = (V, E)$ as a special digraph $G^+ = (V, E^+)$ where E^+ is obtained from E by replacing each

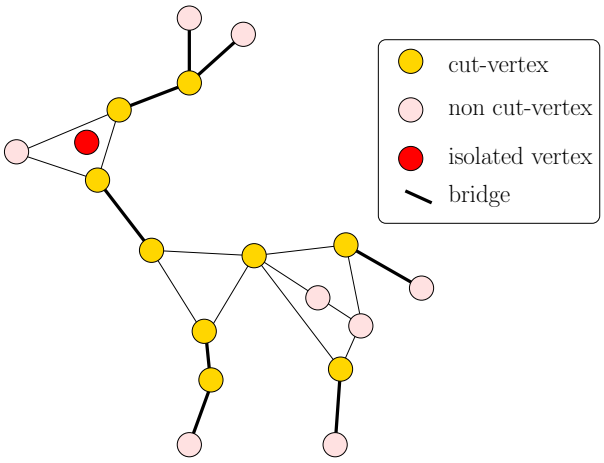


Figure 20: Reindeer bigraph with 3 nontrivial biconnected components and 8 bridges.

undirected edge u–v by two directed edges $u \to v$ and $v \to u$ (called **partners** of each other). By definition, G and G^+ have the same adjacency list representation! So we can apply the digraph edge classification (¶40) to G^+. Recall that the edges of digraphs fall under one of the following types:

$$\text{tree, back, forward, cross, unseen.}$$

Since G is connected, there are no unseen edges in G^+. Since every edge of G^+ has a partner, there are no cross edges:

Lemma 16 *Consider any DFS classification of the edges of G^+.*

(a) There are no cross edges or unseen edges. Thus, we only have tree, back or forward edges.

(b) Edge u–v is a back edge iff its partner v–u is a forward edge or a tree edge.

We leave the proof as an exercise. One unsatisfactory aspect of using digraph G^+ instead of G is its redundancy: the edges comes in pairs:
(1) Call the partner of a tree edge a **parent edge**. A tree edge is an **improper** forward edge; likewise, a parent edge is an **improper** back edge.
(2) We are left with **proper** forward or back edges, and they come in pairs.

We propose to only keep the parent edges and proper forward edges. Thus the number of directed edges is halved! Moreover, we see that every proper forward edge i–j defines a unique cycle of the form

$$[i\text{–}j\text{–}p(j)\text{–}p^2(j)\text{–}\cdots\text{–}p^k(j)], \quad (k \geqslant 2) \tag{29}$$

where $p^{k+1}(j) = i$. Call this an **elementary cycle** of i–j. Recall that $[u_1\text{–}u_2\text{–}\cdots\text{–}u_k]$ is the cycle corresponding to a closed path $(u_1\text{–}u_2\text{–}\cdots\text{–}u_k\text{–}u_1)$. So each forward edge determines a unique elementary cycle. In Figure 21, there are 4 forward edges, and hence 4 elementary cycles.

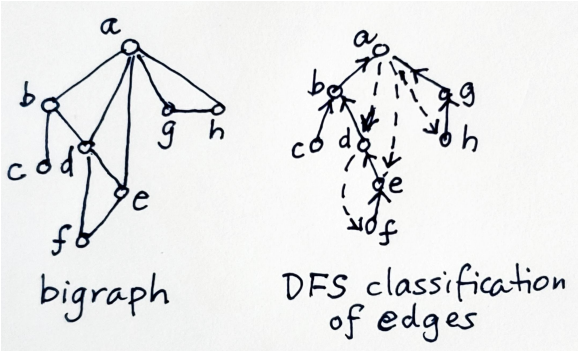


Figure 21: Bigraph with $n = 8, m = 11$ and $4 = m - (n-1)$ forward edges.

Henceforth, by the **edge classification** of a bigraph G, we mean that each original undirected edge of G is turned into a directed edge labeled

parent or forward. (30)

Moreover, the set of parent edges defines a rooted spanning tree of G. See Figure 21.

Now that we understand the DFS edge classification of bigraphs, how do we represent it? We will use two data structures, both of which are familiar:

(B1) The first data structure is to represent the edges of the DFS tree. We use the standard parent array $p[i \in V]$ (see ¶37 above) where $p[i] \in V$ is the parent of i. If i is the root, $p[i] = \texttt{nil}$.

(B2) The second data structure is analogous to the adjacency list data structure, consisting of an adjacency list array $A[1..n]$ where each $A[i]$ points to a linked list. We now denote the array as $F[1..n]$, where each $F[i]$ is the head of a list of vertices j such that i–j is a forward edge. We will call $F[1..n]$ the **forward list array** (or "forward array" for short).

To better support our algorithms, we assume that each $F[i]$ represents a doubly-linked list. Each node u in such a list has 3 fields,

$$u.\texttt{Vertex}, \quad u.\texttt{Next}, \quad u.\texttt{Prev}.$$

Moreover the node $F[i]$ represents i (i.e., $F[i].\texttt{Vertex} = i$) (recall that this is the "verbose" adjacency lists in ¶17, page 17. The array $F[1..n]$ is illustrated in Figure 22(b).

How can we compute $F[i]$? A simple approach is to remove vertices from $A[i]$ to convert it to $F[i]$. We will use a self-modifying DFS algorithm that converts the input $A[i]$ into $F[i]$. For instance, the adjacency array $A[1..5]$ of the bigraph G_5 in Figure 22(a) will be converted to the forward list array $F[1..5]$ Figure 22(b).

Example 5 To see what needs to be done to transform $A[i]$ to $F[i]$, consider the vertex v_5 in the bigraph G_{10} in Figure 24(b). For example, for the vertex v_5, its adjacency list is transformed as

$$(v_5, v_2, v_3, v_6, v_9, v_{10}) \to (v_5, v_{10}). \tag{31}$$

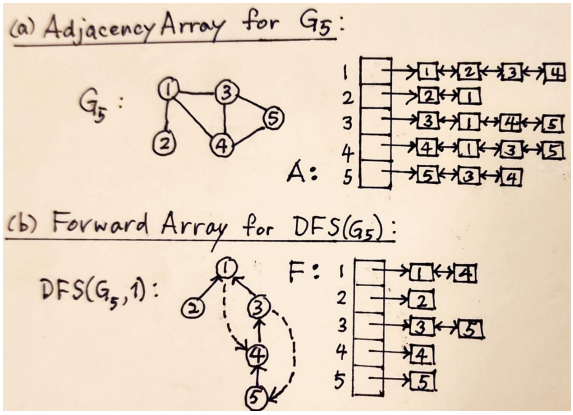


Figure 22: (a) Adjacency Array of G_5 (verbose form); (b) Forward Array of $DFS(G_5)$

In other words, we must eliminate the vertices v_2, v_3, v_6, v_9 from the adjacency list. These eliminations are justified as follows:

- We eliminate v_2 because v_5–v_2 is a back edge;
- v_3 because v_5–v_3 is a parent edge;
- v_6 and v_9 because v_5–v_6 and v_5–v_9 are tree edges.

■

We transformation from $A[i]$ to $F[i]$ is justified the following lemma:

Lemma 17 *Let v be a node in $A[i]$, with v.*`Vertex` *$= j$. Then v must be removed iff i–j is either a back edge or a tree edge.*

The $A[i] \to F[i]$ transformation can be achieved by programming the PREVISIT(v, u) macro of DFS. Since we want to manipulate the adjacency lists, the arguments u, v of PREVISIT are not just vertices of V but nodes from the adjacency lists:

```
PREVISIT(v, u):    ◁ visiting node v from node u
    If (v.Vertex is unseen),    ◁ u–v represents a tree-edge
        p[v.Vertex] ← u.Vertex    ◁ update the parent array
    If (v.Vertex is not done),    ◁ i.e., v.Vertex is seen or unseen.
        v.Prev.Next ← v.Next    ◁ remove v
```

Theorem 18 *The DFS shell with the preceding PREVISIT macro correctly computes the parent array $p[i \in V]$ and forward array $F[i \in V]$ representing an edge classification of G.*

Proof. The parent array construction in PREVISIT is standard. But the destructive transformation of the adjacent list $A[i]$ (renaming it $F[i]$) needs a bit of justification: suppose we removed node v from $A[i]$. If v.`Vertex` $= j$, it means that the color of j is not `done`. That means that j is either `unseen` or `seen`. If j is `unseen`, it means i–j is a tree edge. If j is `seen`, it means i–j is a back edge. In either case, this removal is precisely what we need.

We should examine the DFS code to verify that even if i–j is a tree edge, the removal of j does not impact the ability of DFS to recursively call DFS$(V, E; j)$ in the next step. **Q.E.D.**

¶50. Biconnected Components. Let $G = (V, E)$ be a bigraph. A non-empty subset $C \subseteq V$ is a **biconnected set** of G if for every pair u, v of distinct vertices in C, there is a simple cycle of vertices in C that contains u and v. *Note we require the cycle to be simple*, in contrast to the general definition of connectivity in ¶14. In the literature, biconnected sets are also known as **2-connected sets**. Some consequences of this definition:

- 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.
- Any singleton $\{u\}$ is also a biconnected set, for trivial reasons.

We come to our key definition: If C is a biconnected set that of maximal size, then we call C a **biconnected component**. When there is no ambiguity, we may simply say "component" instead of biconnected component. If G has only one component, then G is called a **biconnected graph**, also known as a 2-connected graph.[21]

Example 6 **Biconnected components may not be disjoint.**
Consider the bigraph $G = (V, E)$ where $V = \{a, b, c\}$ and $E = \{a$–b, b–$c\}$. Then V is not a biconnected set since there is no simple cycle containing both a and c. Since $C_1 = \{a, b\}$ is a biconnected set and it is maximal, we conclude that C_1 is a component. By the same token, $C_2 = \{b, c\}$ is a component. This is most interesting: $C_1 \cap C_2$ is non-empty, but equals $\{b\}$. We call[22] b a **cut vertex**. ■

Lemma 19 (Cut Vertex Lemma) *If C_1, C_2 are distinct biconnected components of a bigraph G, then $|C_1 \cap C_2| \leqslant 1$.*

If $C_1 \cap C_2 = \{v\}$, then we call v a **cut vertex**. So the above lemma says that two components are either disjoint or shares a unique cut vertex.

A component C where $|C| \leqslant 2$ is called a **trivial** component. Non-trivial components where $|C| \geqslant 3$ are called[23] **blocks**. For example, a biconnected graph G of size $\geqslant 3$ is a single block.

[21] But usually, a k-connected graph is required to have size $|V| \geqslant k$. But we allow biconnected graphs of size 1.

[22] Alternatively a cut vertex is a node such that, if we remove it and all edges incident to it, the number of connected compoents of the graph increases. Cut vertices are also known as **articulation points**.

[23] The block terminology is from Harary and Prins. Even [7] calls such a set "inseparable".

- A vertex v is called an **isolated vertex** if $C = \{v\}$ is a trivial component.
- An edge u–v is called a **bridge** if $C = \{u, v\}$ is a trivial component.
- If C is a block, then the restricted graph $G|C$ contains at least one non-trivial cycle of size $\geqslant 3$.

¶51. What is a Reduced Bigraph? We want to define the "reduced bigraph of G", denoted G^c. This turns out to be subtle. An obvious definition, in analogy to the case for digraphs is this: let $G^c = (V^c, E^c)$ be the bigraph where V^c comprises the biconnected components of G, and C–$C' \in E^c$ iff $C \cap C'$ is non-empty. This is a possible approach, but we want to condense this structure even more, by introducing the concept of a "singular component" which are basically the maximally connected set of bridges.

The superscript in G^c refers to "component"

Suppose u–v is a bridge. If u is not a cut vertex (i.e., not shared with another component), then we call it an **end vertex**. E.g., the reindeer graph in Figure 20 contains exactly 5 end vertices (two on the antler, two attached to the legs and one attached to the tail).

We come to a key definition: a **singular vertex** refers to one of three possibilities: a cut vertex or an isolated vertex or an end vertex. Then **singular component** is defined as a maximal non-empty set $S \subseteq V$ of singular vertices such that the restricted graph $G|S$ is connected. A singular component S is **trivial** if $|S| \leqslant 2$. These are possibilities for trivial singular components:

- If $S = \{u\}$, then u is either an isolated vertex or there may be two blocks C_1, C_2 such that $C_1 \cap C_2 = \{u\}$. We call such a u the **nexus** of C_1 and C_2.
- If $C = \{u, v\}$, then u–v must be a bridge.

E.g., the reindeer graph in Figure 20 has 2 non-trivial singular components (antler, fore leg), and 5 trivial singular components (eye, neck, tail, hind leg, and a nexus connecting the front and back torso).

Finally, we define the **reduced bigraph** G^c of G to be a bipartite graph $G^c = (V^b, V^s; E^c)$ where

- V^b is comprised of the blocks and V^s is comprised of singular components.
- E^c is comprised of edges of the form S–C where $S \in V^s$, $C \in V^c$, and $C \cap C' \neq \emptyset$.

The reduced bigraph of the reindeer graph is shown in Figure 23 in which V^c has 3 blocks and 7 singular components. We have the proof of the following as an Exercise:

Lemma 20 (Reduced bigraph)
Let $G^c = (V^b, V^s; E^c)$ be the reduced bigraph of G.

- *For $S \in V^s$, the induced subgraph $G|S$ is a free tree with $|S| - 1$ edges. All these edges are bridges.*

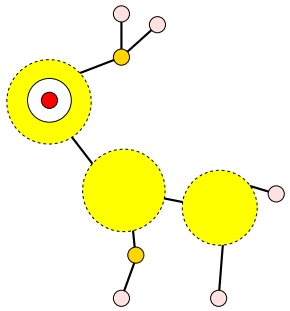


Figure 23: Reduced Reindeer graph (error in figure: the front and back torsos are connect by a nexus, not a bridge.

1573 • *G^c is a acyclic (contains no cycles).*

> Harary and Prins [8] define a closely-related concept of the **block-cutpoint tree** $T(G)$. Their "blocks" include bridges, as there is no independent concept of a bridge. Then $T(G) = (V_b \cup V_c, E_{bc})$ is a bi-
1574 partite graph where V_b is the set of "blocks", V_c the set of cut vertices and isolated vertices. If $B \in V_b$ and $u \in V_c$, then $B-u$ is an edge of E_{bc} iff $u \in B$. The relationship between $T(G)$ and G^c is explored in an Exercise.

¶52. Handle Decomposition of Connected Bigraphs. We will prove a structure theorem about biconnected sets, based on the "handle decomposition" of Jens Schmidt. Recall that from the connected bigraph $G = (V, E)$ we can call DFS to compute two arrays:

$$p[i \in V] \quad \text{and} \quad F[i \in V]$$

1575 which classify each edge of E as a parent or forward edge. We can use this data structure to
1576 compute a certain decomposition of G which we now explain.

Let $G = (V, E)$ and $H = (V', E')$ be bigraphs. If $V \cap V' = \{a, b\}$, $a \neq b$ and H is a path a—$\cdots$—b, then we call H a **handle** for G. Moreover, the bigraph

$$G + H := (V \cup V', E \cup E')$$

1577 is called the extension of G by handle H.

1578 **Theorem 21 (Handle Theorem)** *Let $G = (V, E)$ be a nontrivial biconnected graph.*

1579 *(i) For any three distinct vertices $a, b, c \in V$, G has a simple path of the form a—$\cdots$—b—$\cdots$—c.*

1580 *(ii) Let $H = (V(H), E(H))$ be a handle for G with $V \cap V(H) = \{a, c\}$. then $G + H$ is*
1581 *biconnected.*

(iii) If G is not a cycle, then we can express G as

$$G = G' + H$$

1582 *for some biconnected graph G' with handle H.*

Proof. (i) By biconnectivity of G, there is a simple cycle of the form $p_{bc} = (b\text{–}\cdots\text{–}c\text{–}\cdots\text{–}b)$. There is also a path $p_{ab} = (a\text{–}\cdots\text{–}b)$. If p_{bc} and p_{ab} are disjoint except for sharing vertex b, then clearly we can construct the desired path $a\text{–}\cdots\text{–}b\text{–}\cdots\text{–}c$. Otherwise, let consider the shortest prefix of p_{ab}, say $p_{ad} = (a\text{–}\cdots\text{–}d)$ where d is a new vertex in p_{bc}. Then, the cycle p_{bc} contains a subpath of the form $p_{dc} = (d\text{–}\cdots\text{–}b\text{–}\cdots\text{–}c)$. The desired path is obtained by the concatenation

$$p_{ad}; p_{dc}.$$

(ii) To show that $G + H$ is biconnected, suppose b, b' are two vertices of $G + H$. We must show that $G + H$ contains a simple cycle Z containing b and b'. There are 3 possibilities for b, b': (1) If $b, b' \in V$, the existence of Z follows from the biconnectivity of G. (2) If $b, b' \in V(H)$, this Z is the concatenation of H and a path $a\text{–}\cdots\text{–}c$ in G. (3) Finally, if $b \in V$ and $b' \in V(H)$, by part(i), there exists a simple path $P = (a\text{–}\cdots\text{–}b\text{–}\cdots\text{–}c)$ in G. We construct the cycle Z by concatenating P with P.
(iii) Since G is not a cycle, there exists a vertex $u_0 \in V$ of degree > 2. For any edge $u_0\text{–}u_1$, there is a unique path $H = (u_0\text{–}u_1\text{–}\cdots\text{–}u_k)$ $(k \geqslant 1)$ such that the degree of $u_1, u_2, \ldots, u_{k-1}$ is 2, but u_k has degree > 2. Also, $u_0 \neq u_k$ since otherwise u_0 would be a cut-vertex. Clearly $G = G' + H$ for some $G' = (V', E')$. We must show that $G' = (V', E')$ is nontrivial and biconnected.

It is nontrivial because besides u_0, u_k, there is another vertex since G is not a cycle. Let $a, b \in G'$. We must show a simple cycle in G' containing a, b. Let Z be a shortest cycle in G containing a, b. If H is not contained in Z, then Z is a cycle in G'. Otherwise. Z contains a simple path of the form $P = (u_0\text{–}\cdots\text{–}a\text{–}\cdots\text{–}b\text{–}\cdots\text{–}u_k)$ (wlog). Since $\deg(u_0) \geqslant 3$, there is an edge $u_0\text{–}v_0$. Moreover, if $v_0 \in Z$, then v_0 lies strictly between $a\text{–}\cdots\text{–}b$ in P. Otherwise, Z is not the shortest cycle. Similarly there is an edge $u_k\text{–}v_1$ where, if $v_1 \in Z$, then v_1 lies between $a\text{–}\cdots\text{–}b$. **Q.E.D.**

Corollary 22 (Handle Decomposition)
Every nontrivial biconnected graph G can be expressed in the form

$$G = C + P_1 + P_2 + \cdots + P_k \tag{32}$$

where C is a simple cycle of size at least 3 and for $i = 1, \ldots, k$, each P_i is a handle of

$$G_i := C + P_1 + P_2 \cdots + P_{i-2} + P_{i-1}. \tag{33}$$

The expression (32) is called a **handle decomposition** of G. Moreover, it should be properly be parenthesized as

$$G = ((\cdots((C + P_1) + P_2)\cdots) + P_{k-1}) + P_k$$

and similarly for G_i in (33). In other words, we assume a left-associativity rule for the operator $+$.

¶53. Chain Decomposition: detecting bridges and nexuses. Suppose our connected graph $G = (V, E)$ has its edges classified into tree edges and forward edges, as in ¶49, page 56. The edge classification is represented by a pair of data structures

$$(p[V], F[V]) \tag{34}$$

where $p[V]$ is the parent array of the DFS tree, and $F[V]$ is the forward list array analogous to an adjacency list array.

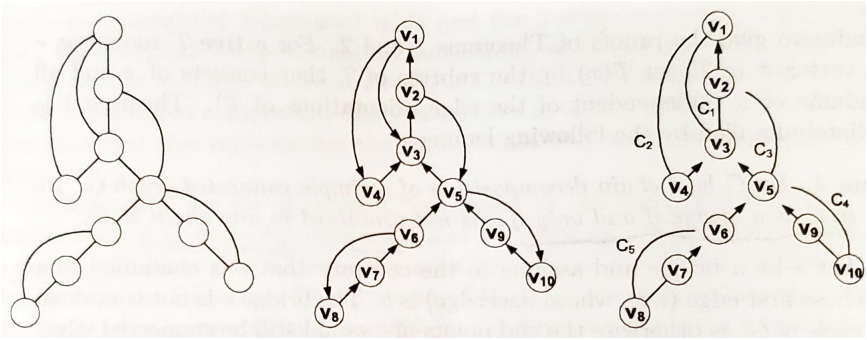


Figure 24: (a) bigraph G_{10}, (b) Its DFS tree, (c) The chain decomposition. Taken from [Jens Schmidt].

EXAMPLE 7 In Figure 24(b), we see such an edge classification of a graph G_{10} of size 10. For any forward edge i–j, there is a unique cycle of the form

$$Cycle(i–j) = [i–j–p[j]–p^2[j]–\cdots–p^k[j]], \qquad (k \geqslant 0) \tag{35}$$

such that $p^{k+1}[j] = i$. Note that $Cycle(i–j)$ is a nontrivial cycle of length at least 3 by the definition of a forward edge.

We see by inspection that G_{10} has 3 blocks: $C_1 = \{v_1, v_2, v_3, v_4, v_5\}$, $C_2 = \{v_6, v_7, v_8\}$, $C_2 = \{v_5, v_9, v_{10}\}$. There is one bridge v_6–v_5 and one nexus v_5

There are 5 forward edges, each giving rise to a chain: $Cycle(v_1 \to v_4) = [v_1–v_4–v_3–v_2]$, $Cycle(v_1 \to v_3) = [v_1–v_3–v_2]$, etc. ■

We decompose the graph into "chains" in the following way: initially mark all vertices as `unseen`. Starting from each forward edge i–j, we mark i and j as `seen`, and then follow parent pointers starting from j:

$$P = (i–j–p[j]–p^2[j]–\cdots–p^k[j]), \qquad (k \geqslant 0) \tag{36}$$

until we first encounter a `seen` vertex $p^k[j]$. If $p^k[j] = i$ then P is $Cycle(i–j)$; otherwise P is a handle. In any case we call P a **chain**, denoted $P = Chain(i–j)$. By iterating through all the forward edges, we will finally output a sequence of chains

$$\boldsymbol{C} = (C_1, C_2, \ldots, C_k). \tag{37}$$

We call $\boldsymbol{C}$ in (37) a **chain decomposition** of bigraph G. In pseudo-code:

Chain Decomposition Algorithm:
INPUT: $p[i \in V]$ and $F[i \in V]$, the parent and forward arrays from a DFS edge classification of $G(V, E)$.
OUTPUT: A chain decomposition $(C_1, C_2, \ldots, C_k)$ of G.
Mark all vertices `unseen`.
Initialize $\boldsymbol{C}$ as an empty list.
For $i = 1, \ldots, n$
  For each forward edge i–j in $F[i]$
    Mark i as `seen`, and initialize the stack $P \leftarrow (i, j)$
    $k \leftarrow j$
    While (k is `unseen`)
      Mark k as `seen`.
      $k \leftarrow p[k]$ and $P.push(k)$
    Append P to $\boldsymbol{C}$
Return $\boldsymbol{C}$

The chain decomposition of a connected bigraph $G = (V, E)$ has

$$k = |E| - |V| + 1 = m - n + 1 \tag{38}$$

chains. To see this, note that since G is connected, its DFS tree has $|V| - 1$ edges. The remaining $|E| - (|V| - 1)$ edges are forward edges. Each of them gives rise to a chain.

Each block C of G has a handle decomposition comprised of chain in the decomposition $\boldsymbol{C}$. Any edge in G that is not in $\boldsymbol{C}$ must clearly be an edge of the DFS tree represented by the array $p[V]$. The next lemma identifies these edges as bridges of G:

Lemma 23 (Bridge Lemma) *Let $\boldsymbol{C}$ be a chain decomposition of G. An edge e of G is a bridge iff e does not appear in any chain of $\boldsymbol{C}$.*

Proof. ($\Rightarrow$) If edge e appears in a chain $Chain(i$–$j) \in \boldsymbol{C}$ then clearly e is not a bridge.

($\Leftarrow$) So suppose e does not appear in any chain in $\boldsymbol{C}$. We will show that e must be a bridge. Clearly e is not a forward edge (or else it appears in the chain of the forward edge). So it must be an edge of the DFS tree. Say $e = (i$–$p[i])$ for some i. Consider the subtree $T(i)$ rooted at i.

Consider a forward edge j–k: if $j \in T(i)$ then $k \in T(i)$ by definition of forward edge. If $k \in T(i)$ we also claim that $j \in T(i)$: otherwise, e would lie in $Cycle(j$–$k)$. This proves that i is either an end vertex (in case $T(i)$ has only one node i) or i is a cut vertex.

Let r be the root of the DFS tree. If $p[i] \neq r$ then $p[i]$ is a cut vertex. If $p[i] = r$ and r has more than one child in the DFS tree, again $p[i]$ is a cut vertex. In either case, we conclude that $e = i$–$p[i]$ is a bridge, and our lemma is proved.

So assume $p[i] \neq r$ and $p[i]$ has only one child. This proves that $p[i]$ is an end vertex. As we have shown that i is either an end vertex or cut vertex, this proves that e is a bridge. **Q.E.D.**

The following lemma identifies the nexusus of G: recall that these are cut vertices not associated with any bridges.

Lemma 24 (Nexus Lemma) *Let $\boldsymbol{C}$ be a chain decomposition of G with DFS tree T.*
(a) The root r of T is a nexus iff there are at least chains in $\boldsymbol{C}$ of the form $Cycle(r \to i)$ and $Cycle(r \to j)$.
(b) A non-root i of T such that $i{-}p[i]$ is not a bridge. Then i is a nexus iff for some j, $\boldsymbol{C}$ contains a cycle of the form $Cycle(i \to j)$.

Proof. (a) If there are two cycles $Cycle(r \to i)$ and $Cycle(r \to j)$, then clearly r is a nexus.

To show the converse: suppose r is a nexus. Then there are two blocks C_1, C_2 such that $C_1 \cap C_2 = \{r\}$. We will find two cycles of C_1 and C_2 in $\boldsymbol{C}$. There is a lexicographically first path in the DFS tree of the form

$$r{-}i_1{-}i_2{-}\cdots{-}i_k)$$

such that $i_k{-}r$ is a back-edge. Thus $r{-}i_k$ is a forward edge and $Cycle(r{-}i_k)$ is in the chain decomposition $\boldsymbol{C}$. [Must complete the argument...] Similarly for we can find a cycle of C_2 in $\boldsymbol{C}$.
(b) Suppose i is not the root and $i{-}p[i]$ is not a bridge. If $Cycle(i{-}j)$ exists, then clearly i is a nexus. Conversely, if i is a nexus, we must find a $Cycle(i{-}j) \in \boldsymbol{C}$ for some j. [Must complete the argument...] **Q.E.D.**

¶54. Applications of Chain Decompositions. There are related notions of "connectivity": biconnectivity is essentially[24] the same concept as 2-connectivity in the literature. We have defined a cut-vertex as the intersection of two biconnected components. Equivalently, we could define a **cut-vertex** (or "articulation point") to be a vertex whose removal from a bigraph G, together with all incident edges, would increase the number of connected components of G. The bridge is the edge analogue of cut-vertex: we could also define a **bridge** is an edge $u{-}v$ whose removal will increase the number of connected components of the resulting bigraph; note that the vertices u, v remain in the graph in this definition. We say G is **edge-connected** if it does not have any bridges. Note that nontrivial biconnected graphs are automatically edge-connected. In the literature, edge-connectivity is also known as 2-edge-connectivity. For example, the graph in Figure 20 has 8 bridges and 9 cut-vertices.

We can now answer basic computational questions about G:

(1) Edge-connectivity: suppose we keep count of the number of edges that we visit in (36), accumulated over all forward edges $i{-}j$. We can easily modify the Handle Decomposition Algorithm to get this count. If this count is less than $m = |E|$ then the graph is not edge-connected.

(2) Biconnectivity: If the graph is not edge-connected or it has two cycles in its handle decomposition, then it is not biconnected. Otherwise, we conclude that it is biconnected.

We leave as an Exercise to prove that the proposed method for detecting edge-connectivity or biconnectivity is correct.

Exercises

[24] We say "essentially" because there are variant definitions for biconnected component of small size.

Exercise 7.1: Let G be a biconnected graph and a, b, c are 3 distinct vertices in G. Prove or disprove: G has a simple cycle that passes through a, b, c (in some order). ♢

Exercise 7.2: Consider the bigraph G_{10} in Figure **??**.
(a) Do the canonical hand-simulation of the bigraph edge classification algorithm on G_{10}. Draw the DFS tree and show the forward edges.
(b) Give the ♢

Exercise 7.3: Recall the Harary-Prins notion of block-cutpoint tree $T(G)$.

- Describe $T(G)$ where G is our Reindeer graph.
- Show that $(T(G))^c = G^c$. Thus, $T(G)$ is "less reduced" than our G^c.

♢

Exercise 7.4: Prove Lemma 20 which characterizes reduced bigraphs. ♢

Exercise 7.5: In the text, we showed how to detect edge-connectivity, and also to detect biconnectivity of a bigraph. Prove that these two methods are correct. ♢

Exercise 7.6: In the text, we computed a handle decomposition of a bigraph G where we output a set of cycles and set of handles. However, we did not explicitly attach each handle to its cycle. Modify the algorithm to determine this information. ♢

Exercise 7.7: In the text, we gave an algorithm to detect if a bigraph is biconnected. Generalize this algorithm to compute the reduced graph G^c of the bigraph G. ♢

Exercise 7.8: The notion of biconnected components is defined for bigraphs. To extend it to a digraph $G = (V, E)$, we say a set $S \subseteq V$ is **biconnected** if every $u, v \in S$ is contained in a simple cycle through vertices of S. The maximal biconnected sets are called **biconnected components**.

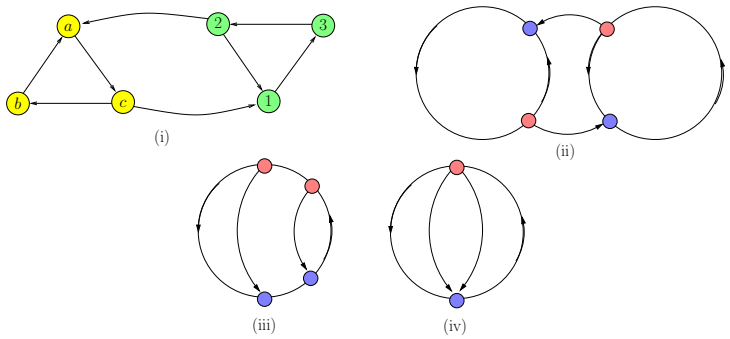


Figure 25: Biconnectedness in digraphs

(a) What are the biconnected components in Figure 25(i)?

(b) How many biconnected components in Figure 25(ii)-(iv)? Note that if C, C' are two biconnected components, $|C \cap C'|$ can be arbitrarily large.
(c) Show that the number of biconnected components is non-polynomial in the number of vertices.
(d) A digraph is **Eulerian** if at every vertex, the outdegree equals the indegree. See Figure 26 for such graphs. Note that Eulerian graphs are a generalization of bigraphs (when viewed as digraphs). Show that the set of edges in an Eulerian graph can be partitioned into a collection of edge-disjoint simple cycles. See Figure 26(iv).
(e) In an Eulerian graph, if two biconnected sets B and B' share at least two common vertices, then $B \cup B'$ is biconnected.
(f) Characterize the set of biconnected components in an Eulerian bigraph. ♢

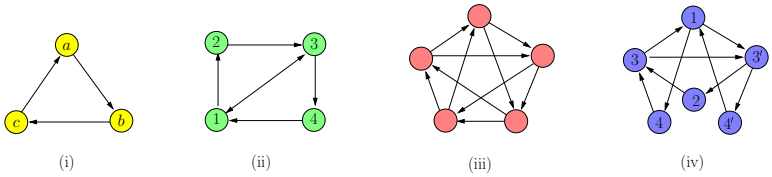


Figure 26: Eulerian digraphs

Exercise 7.9: Suppose we not only want to the number of biconnected components, but also compute the size of each biconnected component. Use Shell programming. ♢

End Exercises

§8. Games on Graphs

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

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

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*

¶55. Game Graphs. To model this, we introduce the concept of a **game graph** $G = (V_A, V_B, E)$ where $V_A \cap V_B = \varnothing$ 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 $v \in V_A$ (resp., $v \in V_B$) represents a state whose next transition is determined by Alice (resp., Bob). A particular path through this graph $(v_1, v_2, \ldots)$ represents a run of the program, with the transition v_i–v_{i+1} determined by Alice if $v_i \in V_A$, and by Bob if $v_i \in V_B$. We might think of the original (single player) reachability/fairness problems as operating in a graph in which V_B is the empty set. The introduction of Bob captures some new realities of an operating system. Reachability/Fairness is now defined to mean "reachable/fair in spite of Bob".

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

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

For $U \subseteq V$, let $\text{Force}(U) = \cup_{u \in U} \text{Force}(u)$ and $\text{Fair}(U) = \cup_{u \in U} \text{Fair}(u)$. The set $\text{Fair}(U)$ is also called the **winning states** for a Büchi game with Büchi objective U. Such games originated in mathematical logic. We will design algorithms to compute the sets $\text{Force}(U)$ and $\text{Fair}(U)$ in times $O(n + m)$ and $O(mn)$. The exercises[25] will show how $\text{Fair}(U)$ can be computed in $O(n^2)$

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

time.

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

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

Let us introduce an operator to capture these properties:

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

such that for all $U \subseteq V$

$$v \in \mu(U) \Leftrightarrow \begin{cases} v \in U, \text{ or} & \text{[BASIS]} \\ v \in V_A \wedge (Out(v) \cap U \neq \varnothing), \text{ or} & \text{[INDUCT(A)]} \\ v \in V_B \wedge (Out(v) \subseteq U) & \text{[INDUCT(B)]} \end{cases} \tag{39}$$

A set $U \subseteq V$ is called a **fixed point** of μ if $\mu(U) = U$. For instance, V is a fixed point of μ. For any $U \subseteq V$, there is a least $i \geqslant 0$ such that $\mu^{(i)}(U) = \mu^{(i+1)}(U)$; define $\mu^*(U)$ to be $\mu^{(i)}(U)$. We easily verify:

Lemma 25 *$\mu^*(U)$ is the **least fixed point** (LFP) of U under the operator μ):*

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

Lemma 26 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 α for Alice such that for all strategies β for Bob, if $\pi = (\alpha, \beta)$ then there exists a $k \geqslant 1$ such that $\pi^k(u) \in U$. This proves that $u \in \mu^k(U)$. **Q.E.D.**

¶57. 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 $v \in V\backslash W$ it is easy to use the conditions in (39) to check whether $v \in \mu(W)$, and if so, add it to W. So the computability of $\mu(W)$ is not in question. But it may be a bit less obvious how to do this efficiently. The critical question is — *in what order should we examine the vertices v or the edges v–w?*

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

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

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

$$C[i] = \begin{cases} 0 & i \in U \\ 1 & i \in V_A \\ degree(i) & i \in V_B \end{cases} . \tag{40}$$

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

It is now clear how we to update this array when processing an edge $(w–u) \in W \times W'$: if $C[u] = 0$, there is nothing to do (u is already in W). Else, we decrement $C[u]$. If $C[u]$ becomes 0 as a result of the decrement, it means u is now a member of W. Note that if $u \in V_A$, then this will happen with the very first decrement of $C[u]$; but if $u \in V_B$, we need to decrement $degree(u)$ times. We need to also take action in case $C[u]$ becomes 0 after decrement: we must now add u to Q. That completes the description of our algorithm, and it is summarized in this pseudo-code:

```
μ*(U):
 Input: G^r = (V_A, V_B, E^r) and U ⊆ V = {1,...,n}
 Output: Array C[1..n] representing μ*(U)
 ▷ Initialization
      Initialize array C[1..n] as in (40)
      Initialize queue Q ← U
 ▷ Main Loop
      While (Q ≠ ∅)
           w ← Q.pop()
           for each u adjacent to w in G^r
                If (C[u] > 0)
                     C[u]--
                     If C[u] == 0, Q.push(u)
      Return(C)
```

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

References

[1] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, Mass., 3rd edition, 2000.

[2] K. Been. *Responsive Thinwire Visualization of Large Geographic Datasets*. Ph.d. thesis, New York University, Department of Computer Science, Courant Institute, Sept. 2002. From `http://cs.nyu.edu/visual/home/pub/`.

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

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

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

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

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

[8] F. Harary and G. Prins. The block-cutpoint-tree of a graph. *Publ.Math.Debrecen*, 13:103–107, 1966.

[9] E. Mäkinen. How to draw a hypergraph. *Intl. J. of Computer Mathematics*, 34:177–185, 1990.

[10] OpenStreetMap.org. Tiger data set, 2020. URL `https://wiki.openstreetmap.org/wiki/TIGER`.

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

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

[13] A. Verroust and M. luce Viaud. Results on hypergraph planarity. Technical Report inria-00389591, INRIA, May 2009.

[14] C. Yap, K. Been, and Z. Du. Responsive thinwire visualization: Application to large geographic datasets. In E. et al., editor, *Proc. SPIE Symp. on Visualization and Data Analysis 2002*, volume 4665, pages 1–12, 2002. 19-25 Jan, 2002, San Jose, California.