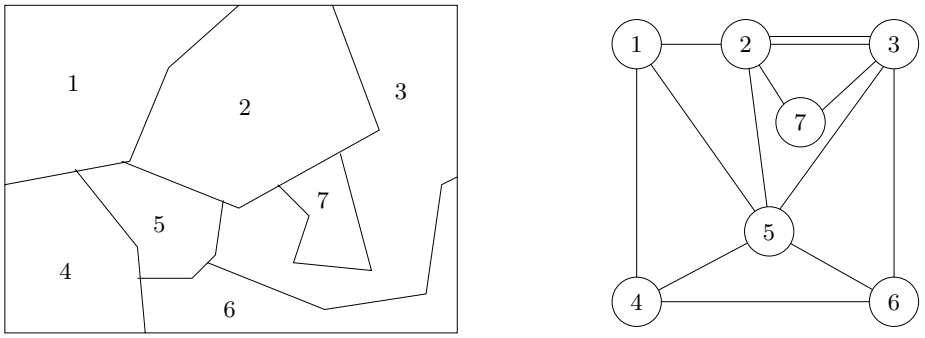
Lecture IV
Pure Graph Problems

A graph is fundamentally a set of mathematical relations (called incidence relations) connecting two sets, a vertex set V and an edge set E. The simplest notion of an edge $e \in E$ is a pair $e = (u, v)$ of vertices. Graphs are useful for modeling abstract mathematical relations in in computer science as well as in many other disciplines. Here are some examples of graphs:



(a) (b)

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

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

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

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

In many applications, our graphs have associated data such as numerical values ("weights") attached to the edges and vertices. These are called **weighted graphs**. The flight connection graph above is an example of this. Graphs without such numerical are called **pure graphs**. In this chapter, we restrict attention to pure graph problems; weighted graphs will be treated in later chapters. The algorithmic issues of pure graphs mostly relate to the concepts of connectivity and paths. These algorithms can be embedded in one of two graph searching strategies called depth-first search (DFS) and breadth-first search (BFS). We also investigate an important problem of pure graphs: testing if a graph is planar. Tarjan [2] was one of the first to systematically study the DFS algorithm and its applications.

§1. Bigraphs and Digraphs

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

We give a general view of graphs. Given two arbitrary sets V, E, an **incidence function** on V, E is $I : E \to 2^V$. Fix an index set J. A **J-graph** G is a set $G = \{I_\alpha : \alpha \in J\}$ of incidence functions, each indexed by an element of J. If $v \in I_\alpha(e)$, we say[1] e is **α-incident** (or simply "incident") on v. Conversely, we say v **α-bounds** e. In case $|J| = 1$, we identify G with the sole incidence function. Elements of V and E are called **vertices** and **edges** of G. Sometimes vertices are called **nodes**, and edges called **arcs**.

Two edges $e, e' \in E$ are **parallel** if for each $\alpha \in J$, $I_\alpha(e) = I_\alpha(e')$. We call G a **simple graph** if it has no parallel edges. Non-simple graphs are also called **multigraphs**. For instance, the adjacency relationship between countries (see figure 1) may require a multigraph representation, since two countries can be adjacent along more than continuous border. In particular, figure 1(b) shows a multigraph with a parallel edge connecting vertices 2 and 3. In the context of pure graphs, a multigraph can be represented by a simple graph together with a positive integer weight associated with each simple edge.

Here are the main types of J-graphs:

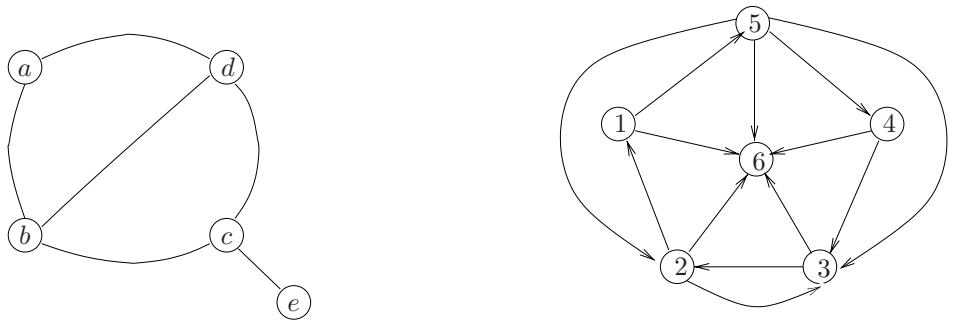
- **Hypergraphs**. Here $|J| = 1$. There is no constraints on the sole incidence relation $I : E \to 2^V$. A simple hypergraph is also called a "set system"; an edge $e \in E$ is then identified with a subset of V and called a "hyperedge".
- **Digraphs**. Here $J = \{0, 1\}$ and $|I_0(e)| = |I_1(e)| = 1$ for all $e \in E$. We call $I_0(e)$ the **start vertex** and $I_1(e)$ the **stop vertex** of e. If $I_0(e) = I_1(e)$, we call e a **self-loop**. Simple digraphs are also known as **directed graphs** because an edge e can be written as an ordered pair $(I_0(e), I_1(e)) = (u, v)$. The edge (u, v) is said to be "directed" from start u to stop v. The edges (u, v) and (v, u) are distinct unless it is a self-loop.
- **Bigraphs**. We can define bigraphs in two equivalent ways: (a) We can regard a bigraph as a hypergraph in which $|I(e)| = 2$ for all $e \in E$. (b) We can regard a bigraph as a digraph with no self-loops and where the edges in E can be partitioned into pairs such that if $e, e' \in E$ are paired then $I_0(e) = I_1(e')$ and $I_1(e) = I_0(e')$. If the digraph is simple, we conclude that (u, v) is an edge iff (v, u) is an edge, and these two are paired. Simple bigraphs are more commonly[2] called **undirected graphs** because its edges are **bi-directional**: (u, v) and (v, u) are considered the same edge.

Non-standard example. Suppose V is the set of people living at a particular instant, and E represents the set of universities. Let $J = \{p, t, s\}$. Let G be a J-graph where $|I_f(e)| = 1$. If $I_f(e) = \{u\}$, it means u is the president of the university e. The sets $I_t(e)$ and $I_s(e)$ are, respectively, the faculty members and students of the university. Clearly, we can extend the index set J to represent other people who are associated with a university in some definite capacity.

Graphical representation of graphs. Bigraphs and digraphs are "linear graphs" in which each edge is incident on one or two vertices. Such graphs have natural graphical representation: elements of V are represented by points (or circles) in the plane and elements of E are represented by finite curve segments connecting these points. Of course, we can distinguish the type of each edge-vertex incidence with a label from the set J.

[1] The incidence terminology is somewhat variable in the literature. In our terminology, "incidence" and "bounding" are inverses: e is incident on v iff v bounds e. The bounding concept comes from a geometric interpretation: the endpoints of a line segments is said to bound the line segment, the edges of a polygon is said to bound the polygon, and so on in higher dimensions.

[2] This terminology is special to this book. Since "undirected graph" is somewhat unwieldy for such a fundamental concept, and since the coinage "digraph" is standard, it seems that the term "bigraph" is justified and helpful.



(a) bigraph (b) digraph

Figure 2: Bigraph and digraph.

In figure 2, we display a bigraph (V, E) where $V = \{a, b, c, d, e\}$ and $E = \{ab, bc, ac, bd, cd, de\}$. Note that for brevity, we simply write "ab" instead of "(a, b)", etc. We also display a digraph (V, E) where $V = \{1, 2, \ldots, 6\}$ and $E = \{15, 54, 43, 32, 21, 16, 26, 36, 46, 56, 52, 53\}$. We display a digraph edge (u, v) by drawing an arrow head incident to, and pointing at, the stop endpoint (v) of its curve segment. E.g., in figure 2(b), all the edges involving vertex 6 has 6 as the stop vertex and so the arrow heads are all pointed at 6. Thus edges are "directed" from the start to the stop vertex. In contrast, the curve segments in bigraphs are undirected (bi-directional).

Set-Theoretic Notations for Simple Graphs. For a simple graph G, each edge $e \in E$ can be completely identified by the set $I(e) = \{I_\alpha(e) : \alpha \in J\}$ of sets. For a hypergraph, $I(e)$ is essentially a subset of V. For a bigraph, $I(e)$ is essentially a 2-element subset of V. For a digraph, $I(e)$ is essentially an ordered pair. To support this alternative characterization of edges, we introduce the following notations. For any set V and integer $k \geq 0$, let

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

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

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

We can now characterize our 3 varieties of graphs as follows:

- A hypergraph is a pair $G = (V, E)$ where $E \subseteq 2^V$.
- A digraph is a pair $G = (V, E)$ where $E \subseteq V^2$.
- A bigraph is a pair $G = (V, E)$ where $E \subseteq \binom{V}{2}$.

Although an edge e in a bigraph is a set $e = \{u, v\}$, as noted above, we also write it as "$e = (u, v)$" as for digraph edges. This convention is useful when we give definitions that cover both digraphs and bigraphs. In the rest of this book, the term "graph" refers to either digraphs or bigraphs. For convenience, some basic graph terminology is collected in §I (Appendix A).

Paths. If (u, v) is an edge, we say that v is **adjacent to** u. A typical usage is this: "for each v adjacent to u, do $\ldots v \ldots$". Note that adjacency is an asymmetric relation in this usage.

Let $p = (v_0, v_1, \ldots, v_k)$, $(k \geq 0)$ be a sequence of edges. We call p a **path** if v_i is adjacent to v_{i-1} for all $i = 1, 2, \ldots, k$.

The **length** of p is k; the path is **trivial** if it has length 0. Call v_0 is the **source** and v_k the **target** of p. Both v_0 and v_k are **endpoints** of p. We also say p is a path **from** v_0 **to** v_k The path p is **closed** if $v_0 = v_k$ and **simple** if all its vertices, with the possible exception of $v_0 = v_k$, are distinct. The **reverse** of $p = (v_0, \ldots, v_k)$ is the path

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

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

Define $\delta_G(u, v)$, or simply $\delta(u, v)$, to be the minimum length of a path from u to v. If there is no path from u to v, then $\delta(u, v) = \infty$ We also call $\delta(u, v)$ the **link distance** from u to v (this terminology will be useful when $\delta(u, v)$ is later generalized to weighted graphs). It is easy to see that

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

These three properties amounts to saying that $\delta(u, v)$ is a metric on V in the case of a bigraph.

Subpaths. Suppose the path p terminates at the vertex where path q begins:

$$p = (v_0, v_1, \ldots, v_k), \quad q = (u_0, u_1, \ldots, v_\ell),$$

where $v_k = u_0$. Then we can **concatenate** them into a new path, written

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

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

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

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

We write $p \equiv q$ in this case. Clearly p and q must both be closed path. It is easily checked that cyclic equivalence is a mathematical equivalence relation. For instance, the following four closed paths are cyclic equivalent:

$$(1, 2, 3, 4, 1) \equiv (2, 3, 4, 1, 2) \equiv (3, 4, 1, 2, 3) \equiv (4, 1, 2, 3, 4).$$

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

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

We define a **cycle** in an equivalence class of closed paths. If the equivalence class of p is the cycle Z, we call p a **representative** of Z; if $p = (v_0, \ldots, v_k)$ then we write Z as

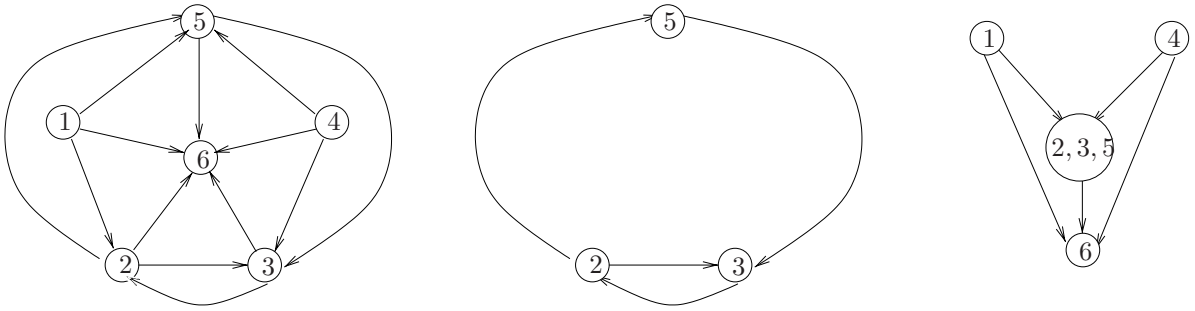
$$Z = [p] = [v_1, v_2, \ldots, v_k] = [v_2, v_3, \ldots, v_k, v_1].$$

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

$$[1, 2, a, b, c, 2, 3, d, e, 3].$$

From the general transfer principle, we conclude that a cycle $Z = [p]$ is **trivial** iff p is a trivial path. We now come to a definition where there is a split between digraphs and bigraphs. A digraph G is **cyclic** if it contains any nontrivial cycle. But for digraphs, this definition will not do. For instance, we can obtain non-trivial cycles of the form $[u, v, u]$ for any edge (u, v). Hence we define a closed path $p = (v_0, v_1, \ldots, v_k)$ to be **irreducible** if (1) $v_{i-1} \neq v_{i+1}$ for all $i = 1, \ldots, k-1$, and (2) $v_1 \neq v_{k-1}$. Otherwise it is **reducible**. So a cycle $Z = [p]$ is reducible iff p is reducible. Finally, a bigraph is **cyclic** if it contains any irreducible non-trivial cycles. In general, a graph is **acyclic** if it is not cyclic.

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



(a) (b) (c)

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

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

Given G, we define the **reduced graph** $G^c = (V^c, E^c)$ whose vertices comprise the components of G, and whose edges are $(C, C') \in E^c$ such that there exists an edge from some vertex in C to some vertex in C'. This is illustrated in figure 3(c).

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

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

DAGs and Trees. A graph without non-trivial cycles is said to be **acyclic**. DAG is a common acronym for "directed acyclic graph". A **tree** is a DAG in which there is a unique node u_0 called the **root** such that there exists a unique path from u_0 to any other node. Trees are ubiquitous in computer science. Thus, we have free trees, rooted trees, ordered trees, search trees, etc. A **free tree** (on a set V of vertices) is a connected bigraph on V with no cycles. This means that it has $|V| - 1$ edges and for every pair of vertices, there is a unique path connecting them. These two properties can also be used as the definition of a free tree. A **rooted tree** is a free tree together with a distinguished node called the **root**. We can convert a rooted tree into a directed graph in two ways: by directing each of its edges away from the root (so the edges are child pointers), or by directing each edge towards the root (so the edges are parent pointers).

Size and Representation Two size parameters are used in measuring the computational complexity of graph problems: $|V|$ and $|E|$. These are typically denoted by n and m. For digraphs or bigraphs, it is clear that $0 \leq m \leq n^2$. If $m = o(n^2)$ for graphs in a family $\mathcal{G}$, we say $\mathcal{G}$ is a **sparse** family of graphs; otherwise the family is **dense**. For example, the family $\mathcal{G}$ of planar graphs is sparse because $m = O(n)$ in planar graphs. Some computational techniques can exploit sparsity of input graphs.

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

- Edge list: a list of the vertices of G and a list edges of G.
- Adjacency list: a list of the vertices of G and for each vertex v, we store the list of vertices that are adjacent to v.
- Adjacency matrix: this is a $n \times n$ Boolean matrix where the (i, j)-th entry is 1 iff there is an edge from the i-th edge to the j-th edge.

The first two methods uses $O(m+n)$ space while the last method uses $O(n^2)$ space. Thus the last method cannot exploit sparsity of the graph.

The above representations can be extended to edge-weighted graphs in a natural way. In case of adjacency matrix, we need to choose some special value that cannot be an edge weight (for instance ∞ or 0). Then each entry is either the edge weight (if an edge is present) or equal to the special value.

In description of many graph algorithms, it is convenient to assume that the vertex set of a graph is $V = \{1, 2, \ldots, n\}$. For instance, this allows us to iterate over all the vertices using an integer variable. To associate an attribute A with each vertex, we can use an array $A[1..n]$ where $A[i]$ is the value of the A-attribute of vertex i.

Coloring Scheme. In many graph algorithms we need to keep track of some "processing status" of the vertices. The status changes in a linear order, from unprocessed to processed. Sometimes, it is important

to denote intermediate status of being partially processed. Viewing the status as colors, we then have a three-color scheme: "white" or "gray" or "black". They correspond to unprocessed, partially processed and completely processed statuses. Alternatively, the white, grey and black colors can be called "unseen", "seen" and "done" (respectively). Initially, all nodes are unseen or white. The color transitions of each node are always in this order:

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

For instance, we may let the status array be an integer $\texttt{color}[1..n]$, with the convention white is 0, gray is 1 and black is 2. Then color transition for vertex i is simply $\texttt{color}[i]++$. Sometimes, a two-color scheme is sufficient: in this case we omit the gray color or the "done" status.

Exercises

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

Exercise 1.2: A trigraph is $G = (V, E)$ where $E \subseteq \binom{V}{3}$. An element $f \in E$ is called a **face** (not "edge"). A pair $\{u, v\} \in \binom{V}{2}$ is called an **edge** provided $\{u, v\} \subseteq f$ for some face f; in this case, we say f is **incident** on e, and e **bound** f). The trigraph is an (abstract) **surface** if each edge bounds exactly two faces. How many nonisomorphic surfaces are there on $n = |V|$ vertices? First consider the case $n = 4, 5, 6$. ♢

End Exercises

§2. Breadth First Search

In many graph problems, we need a **graph traversal** algorithm, that is, an algorithm that systematically "visits" each node and edge of a graph. There is a systematic ways to do this: start from any node s_0 and "visit every edge and node that can be reached from s_0". If there are any other unvisited node s_1, we repeat this process with s_0 replaced by s_1, and so on.

But how do we "visit every edge and node that can be reached from s_0"? This again has a very simple scheme: starting from s, we "process" each edge that we discover from paths starting at s. In general, we will have discovered several edges at once and these edges need to be put into a "container" until it can be processed. There are two standard containers: either a queue or a stack. These two datastructures give rise to the two algorithms for graph traversal: **Breadth First Search** (BFS) and **Depth First Search** (DFS), respectively.

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

The idea of BFS is to systematically visit nodes that are nearer to s_0 before visiting those nodes that are further away. More precisely, if

$$\delta(s_0, u) < \delta(s_0, v) < \infty \tag{2}$$

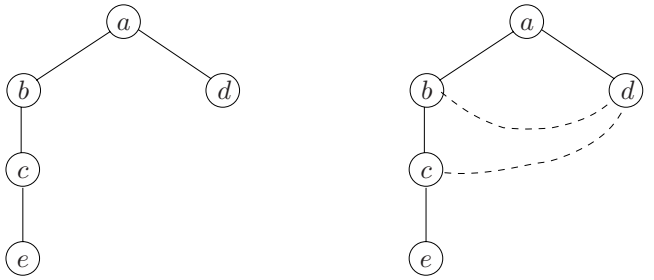
then we visit u before v. If $\delta(s_0, u) = \infty$, then u will not be visited. A **BFS listing at** s_0 is a listing of all the nodes reachable from s_0 in which a node u appears before another node v in the list whenever (2) holds. To illustrate this, suppose G is the bigraph in figure 2 and s_0 is node a. Then two possible BFS listing at a are

$$(a, b, d, c, e) \qquad \text{and} \qquad (a, d, b, c, e). \tag{3}$$

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

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

respectively. To keep track of the status of nodes we will use the color scheme in the previous section (see (1)). We could use two or three colors, but for simplicity, we use only two: white/gray or unseen/seen. There is an underlying tree structure in any particular BFS computation: if v is "seen" from u (in the sense of line 2.3 below), then the edge (u, v) is considered a tree edge. This tree is called the **BFS tree**. The BFS tree corresponding to the first listing in (3) is shown in figure 4(a).



(a) (b)

Figure 4: BFS Tree.

We formulate our BFS algorithm as a shell for accomplishing application specific functions:

```
BFS Algorithm
    Input:    G = (V, E; s_0) a graph (bi- or di-).
    Output:   This is application specific.
    // Initialization:
1.1     Initialize the queue Q to contain just s_0.
1.2     INIT(G) // APPLICATION SPECIFIC
    // Main Loop:
        while Q ≠ ∅ do
2.1         u ← dequeue(Q).
2.2         for each v adjacent to u do
2.3             if v is 'unseen' then
2.4                 color v 'seen'
2.5                 VISIT(v, u) // APPLICATION SPECIFIC
2.6                 enqueue(Q, v).
2.7         POSTVISIT(u) // APPLICATION SPECIFIC.
```

The application-specific subroutines INIT, VISIT and POSTVISIT can be null operations. Note that VISIT(v, u) represents visiting v **from** u. If BFS is a standalone code, then INIT(G) will be expected to

initialize the color of all nodes to `unseen`, and s_0 has color `seen`. In general, they will accomplish application specific tasks. For instance:

- Suppose you wish to print a BFS listing of the nodes reachable from s_0. Then POSTVISIT(u) simply prints the name of u. Other subroutines remain null operations.
- Suppose you wish to compute to compute the BFS tree T. If we view T as a set of edges, then INIT(G) could initial a set T to be empty. In VISIT(v, u), we add the edge (u, v) to T.
- Suppose you wish to determine the depth $d[u]$ of each node u in the BFS Tree. Then INIT(G) could set $d[s_0] = 0$, and in VISIT(v, u), we will set $d[v] = 1 + d[u]$.

Time Analysis. We will not count the time for the application-specific subroutines. The initialization is $\Theta(n)$ and the main loop is $\Theta(m') = \mathcal{O}(m)$ where m' is the number of reachable edges. (An edge (u, v) is 'reachable' if u is reachable). This giving a total complexity of $\mathcal{O}(n + m)$.

The BFS Algorithm defines a BFS Tree and this assigns a depth to each node reachable from s_0. The BFS Algorithm is characterized by what we might call the **BFS Property**: *a node at depth i is POSTVISITED before any node in depth greater than i.*

We claim that the edges of the graph G can be classified into the following types by the BFS Algorithm (see figure 4(b)):

- **Tree edges**: these are the edges of the BFS tree.
- **Level edges**: these are edges between nodes in the same level of the BFS tree. E.g., edge *bd* in figure 4(b).
- **Cross Level edges**: these are non-tree edges that connect nodes in two different levels. But note that the two levels differ by exactly one. E.g., edge *cd* in figure 4(b).
- **Unseen edges**: these are edges that are not used during the computation. The involve nodes not reachable from s_0.

It is easy to see that all the above types of edges exists. The only question is why can't there be other kinds of non-tree edges. In particular, why can't there be an edge (u, v) where the depth of u is 2 or more larger than v's depth? Suppose such an edge exists, and let w be the parent of v in the BFS tree. By the BFS Property, u is POSTVISITED before w. If v was first seen by w, this means that v would be unseen when u was being processed. This contradicts the assumption that (u, v) is not a tree edge.

We will leave it as an exercise to modify our BFS algorithm above so that all edges are correctly classified.

Driver Program. In our BFS algorithm we assume that a source node $s_0 \in V$ is given. This is guaranteed to visit all nodes reachable from s_0. What if we need to process all nodes, not just those reachable from a given node? In this case, we write a "driver program" that repeatedly calls our BFS algorithm. Moreover, we remove the initialization step (Step 1) from BFS and move it into the driver program. Here is the driver program:

```
BFS Driver Algorithm
     Input:    G = (V, E) a graph.
     Output:   E.g., a set of BFS trees that span G.
     // Initialization:
1.1       Color all nodes as 'unseen'.
1.2       INIT(G) // APPLICATION SPECIFIC
     Main Loop:
2.1       For each node v in V do
2.2            if v is 'unseen' then
                    call BFS((V, E; v)).
```

An simple application of this is to compute connected components of a bigraph G. Let us view this task as one of assigning a component number $c[u]$ to each node in V. The component number is arbitrary with the only requirement that $c[u] = c[v]$ iff u, v belongs to the same component. The reader can easily modify the above driver program and BFS to solve this problem.

Exercises

Exercise 2.1: Prove that every node that is reachable from the source will be seen by BFS. ♢

Exercise 2.2:
(a) Prove that the BFS tree is indeed a tree.
(b) Show for any $(u, v) \in E$, if $\delta(s_0, v) < \infty$ then $|\delta(s_0, v) - \delta(s_0, u)| \leq 1$.
(c) For any node v in the BFS tree, the level number of v is equal to $\delta(s_0, v)$. ♢

Exercise 2.3: Modify the BFS algorithm so that it computes the level number $\delta(s_0, v)$ of every node v reachable from s_0. ♢

Exercise 2.4: Let $G = (V, E; \lambda)$ be a connected bigraph in which each vertex $v \in V$ has an associated value $\lambda(v) \in \mathbb{R}$.
(a) Give an algorithm to compute the sum $\sum_{v \in V} \lambda(v)$.
(b) Give an algorithm to label every edge $e \in E$ with the value $|\lambda(u) - \lambda(v)|$ where $e = (u, v)$. ♢

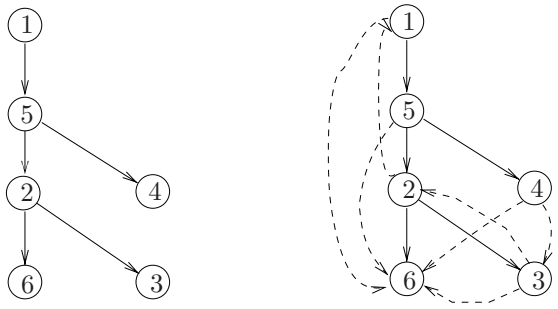
End Exercises

§3. Simple Depth First Search

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

Here is an account of this simplified DFS algorithm. Starting the search from the source s_0, the idea is to go as deep along some path (any path) as much as possible *without visiting any node twice*. When this is

no longer possible, we back up towards the source s_0, but only enough for us to go forward in depth again. In illustration, suppose G is the digraph in figure 2, and s_0 is node 1. Then one possible deepest path from 1 is $(1, 5, 2, 6)$. From node 6, we backup to node 2, from where we can advance to node 3. Again we need to backup, and so on. This can be represented by a DFS tree, as represented in figure 5(a).



(a) (b)

Figure 5: DFS Tree.

To support this backing up along a path, we need the **stack** ADT, which is similar to the queue ADT except that the insertion and deletion of items are based on the Last-In-First-Out (LIFO) discipline. The insert and delete operations are denoted

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

where S is a stack and x an item.

Again, we color every node as 'unseen' or 'seen', exactly as in BFS. We similarly define a **DFS tree** underlying any particular DFS computation: the edges of this tree are precisely those (u, v) such that v is 'seen' from u.

It is instructive to see that the DFS and BFS algorithms are structurally identical, except for their choices of ADTs. Again, INIT(G), VISIT(v, u) and POSTVISIT(u) and application specific code that can the null operation. We now give an equivalent, recursive form of the same algorithm:

```
DFS Algorithm (Recursive form)
     Input:     G = (V, E; s_0) a graph.
     Output:    E.g., DFS tree T rooted at s_0.
     Initialization:
1          INIT(G) // Application Specific.
2          for each v adjacent to s_0 do
3               if v is 'unseen' then
4                    color v 'seen',
5                    VISIT(v, u). // Application Specific
6                    DFS(v).
7          POSTVISIT(u).
```

The behavior of DFS is somewhat more intricate to analyze. We can classify the edges of the graph G as follows (see figure 5(b)):

- **Tree edges**: these are the edges belonging to the DFS tree.
- **Back edges**: these are non-tree edges $(u, v) \in E$ where v is an ancestor of u. Note: (u, u) is considerd a back edge. E.g., edges 21 and 32 in figure 5(b).
- **Forward edges**: these are non-tree edges $(u, v) \in E$ where v is a descendent of u. E.g., edges 16 and 56 in figure 5(b).
- **Cross edges**: these are edges (u, v) that are not classified by the above, but where u, v are visited. E.g., edges 46, 36 and 43 in figure 5(b).
- **Unseen edges**: all other edges are put in this category. These are edges (u, v) in which u is unseen at the end of the algorithm.

Unfortunately, to modify our simple DFS algorithm to classify these edges is a little harder. In particular, the bicolor scheme (seen/unseen) is no longer sufficient (we cannot distinguish between a cross edge from a forward or back edge). In fact, we also cannot distinguish between forward and back edges.

§4. Full Depth First Search

To perform certain computations using the DFS framework, it is useful to compute additional information about the DFS tree. In particular, we may wish to classify the edges as described in the previous algorithm. Instead of the bicolor scheme, we tricolor each node as unseen/seen/done (or white/gray/black). The VISIT(u) subroutine can be used to color the node u as "done". The "seen" nodes are precisely those are currently in the recursion stack.

A more profound embellishment is to **timestamp** the nodes. There are two kinds of time stamp for each node: time when first encountered, and time when last encountered. To implement timestamps, we assume a global counter C that is initially 0. Each time we encounter a node u in a significant way (the first time or the last time), we increment C and associate this value to the array entry `firstTime`$[u]$ or `lastTime`$[u]$. In some applications, we may only need one of these two values. Let `active`(u) denote the time interval [`firstTime`$[u]$, `lastTime`$[u]$], and we say u is **active** within this interval. We also write `active`$(v) <$ `active`(u) if `lastTime`$[v] <$ `firstTime`$[u]$. It is clear from the nature of the recursion that two active are either disjoint or has a containment relationship. We have the following characterization of edges using timestamps:

Lemma 1 *Let $u, v \in V$. Then v is a descendent of u in the DFS tree if and only if at the time that v was first seen, there is a "white path" from v to u, i.e., a path comprising only of white nodes. This is also equivalent to*

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

Proof. If there is a white path, then by induction on the length of this path, every node on this path will be a descendent of u. Conversely, if v is descendent of u then by induction on the distance of v from u, there will be a white path to u.

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

The following is now easy to see:

Lemma 2 *If (u, v) is an edge then*

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

Application to detecting cycles. Suppose that there are no non-tree edges. Then the graph is acyclic iff there are no back edges. One direction is clear – if there a back edge, we have a cycle. Conversely, if there is a cycle Z, then there must be a node in u in Z that is first reached by the DFS algorithm. We will then get a back edge to u. Hence, we can use the DFS algorithm to check if a graph is acyclic.

Exercises

Exercise 4.1: Give a recursive version of the DFS algorithm. Prove that your version achieves the same behavior as the one given in the text. ♢

Exercise 4.2: Construct a small digraph and run the DFS algorithm on it so that all the 6 classifications of edges appear in your example. ♢

Exercise 4.3: Suppose G is a bigraph. Show that a DFS computation on G will not induce any forward or cross edges. ♢

Exercise 4.4: Suppose $G = (V, E; \lambda)$ is a strongly connected digraph in which $\lambda : E \to \mathbb{R}$.
(a) A **potential function** of G is $\phi : V \to \mathbb{R}$ such that for all $(u, v) \in E$,

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

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

Exercise 4.5: Suppose you are given a connected bigraph G on the vertices $V = [1..n]$. Give an efficient algorithm to compute for each $i \in V$ a value $c[i]$ that is equal to the number of components in G when the vertex i is deleted. ♢

End Exercises

§5. Applications of DFS

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

Topological Sort. One motivation is the so called PERT graphs: these are DAG's where nodes represents activities. An edge $(u, v) \in E$ means that activity u must be performed before activity v. By transitivity, if there is a path from u to v, then u must be performed before v. A topological sort of such a graph amounts to a feasible order of execution of all these activities.

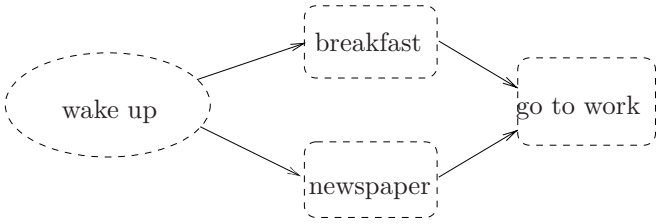


Figure 6: PERT graph

Suppose G is a DAG. Let us call $per[1..n]$ a **topological ranking** of G if the following is true:

$$\text{If } (per[i], per[j]) \in E \text{ then } i < j. \tag{4}$$

Property (4) says that if we perform activities in the order $per[1], per[2], \ldots, per[n]$, then we are assured that there is no "direct" inversion of priority. We say "direct" because the precondition of (4) is information represented by directly by the edges of G. Could there be indirect inversions of priority? The answer is no. In proof, suppose that $i < j$ and $per[i]$ depends on $per[j]$. This means there is a path in the graph G from $per[j]$ to $per[i]$. Let this path be

$$(per[j_0], per[j_1], \ldots, per[j_k])$$

where $j_0 = j$ and $j_k = i$. By (4), we know that $j = j_0 < j_1 < \cdots < j_k = i$. This is a contradiction.

Here then is an algorithm to compute such a permutation:

...

Strong Components. There are three distinct algorithms for computing strong components in digraphs. Here, we will develop a simple yet subtle algorithm based on what we might call "reverse graph search".

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

```
Strong_Component_Subroutine(G, per)
    Input: Digraph G and permutation per[1..n].
    Output: A set of DFS Trees.
        Initialization
1.      For i = 1, ..., n, color[i] = unseen.
        Main Loop
2.      For i = 1, ..., n,
3.          If (color[per[i]] = unseen),
4.              DFS_1(per[i]) // Outputs a DFS Tree
```

This loop is a standard driver program, except that we use $per[i]$ to determine the choice of the next vertex to visit. We assume that $DFS_1(i)$ will (1) change the color of every vertex that it visits from `unseen` to `seen`, and (2) output the DFS tree rooted at i.

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

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

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

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

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

Lemma 3 *There exists a permutation $per[1..n]$ such that the* Strong_Component_Subroutine *is correct, that is, each each DFS Tree that is output in Step 4 corresponds to a strong component of G.*

Proof. Consider the reduced graph G^c of G. Consider a permutation $per[1..n]$ that is a **reverse topological sort** of the vertices of G. More precisely, if $per[i] = u$, we think of i as the ranking of vertex u in our reverse topological sort, and write $rank[u] = i$. So $rank[1..n]$ is just the inverse of $per[1..n]$. Suppose C_1, C_2 are two components of G and (C_1, C_2) is an edge in G^c, then for each vertex $u_1 \in C_1$ and $u_2 \in C_2$, we require the property

$$rank[u_1] > rank[u_2]. \tag{6}$$

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

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

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

We must next show that it is impossible to output vertices that are NOT in C_2. Suppose $DFS_1(u_2)$ reaches some unseen vertex u_0 that belongs to another component C_0. We may assume that u_0 is the first such vertex, and hence (C_2, C_0) is an edge of G^c. By assumption (6), $rank[u_0] < rank[u_2]$. This is a

contradiction because in our main loop, we would have considered the vertex u_0 before u_2. This means that $color[u_0] =$ `seen` by the time we consider u_2. **Q.E.D.**

How do we computer $per[1..n]$ satisfying (6) in the preceding proof? We can compute a topological sort of the *reverse* of graph G. Then $per[i]$ can be the inverse of the topological ranking of the vertices produced by this sort. But rather than compute reverse of G first, we can directly perform a DFS Search of G. For each DFS Tree we find, we rank the vertices according to a pre-order traversal of the DFS Tree. Let us denote this DFS variant by $DFS_0(i)$. Vertices in subsequent DFS trees will receive higher ranks. Moreover, it is simple to modify the code to actually maintain the inverse of the ranking (i.e., directly maintain $per[1..n]$). Here then is the code:

```
Strong_Component_Algorithm(G)
    Input: Digraph G = (V, E), V = {1, 2, ..., n}.
    Output: A permutation per[1..n] of V
        Initialization
1.      For i = 1, ..., n, color[i] = unseen.
2.      Declare array per[1..n].
3.      Rank = 0 (global counter)
        Main Loop
4.      For i = 1, ..., n,
5.          If (color[i] = unseen),
6.              DFS_0(i) // updates per with postorder ranking
        Calls Main Subroutine
6.      Strong_Component_Subroutine(G, per)
```

The code for DFS_0 is as follows:

```
DFS_0(i)
    Input: vertex i in G = (V, E)
    Output: Update of array per[1..n]
        Main Loop
3.      For each vertex v adjacent to i,
4.          If (color[v] = unseen),
5.              DFS_0(v) // recursion
6.      per[++Rank] = i // give vertex i its rank
```

We may verify that the permutation $per[1..6]$ computed by our algorithm on G_6 is precisely that shown in (5).

Remarks. Tarjan [2] was the first to give a linear time algorithm for strong components. R. Kosaraju and M. Sharir independently discovered the reverse graph search method described here. The reverse graph search is conceptually elegant. But since it requires two passes over the graph input, it is slower in practice than the direct method of Tarjan. Yet a third method was discovered by Gabow in 1999. For further discussion of this problem, including history, we refer to Sedgewick [1].

References

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

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