ALGORITHMIC GRAPH THEORY

by

James A. Mc Hugh

New Jersey Institute of Technology

These notes cover graph algorithms, pure graph theory, and applications of graph theory to computer systems. The algorithms are presented in a clear algorithmic style, often with considerable attention to data representation, though no extensive background in either data structures or programming is needed. In addition to the classical graph algorithms, many new random and parallel graph algorithms are included. Algorithm design methods, such as divide and conquer, and search tree techniques, are emphasized. There is an extensive bibliography, and many exercises. The book is appropriate as a text for both undergraduate and graduate students in engineering, mathematics, or computer science, and should be of general interest to professionals in these fields as well.

Chapter 1 introduces the elements of graph theory and algorithmic graph theory. It covers the representations of graphs, basic topics like planarity, matching, hamiltonicity, regular and eulerian graphs, from both theoretical, algorithmic, and practical perspectives. Chapter 2 overviews different algorithmic design techniques, such as backtracking, recursion, randomization, greedy and geometric methods, and approximation, and illustrates their application to various graph problems.

Chapter 3 covers the classical shortest path algorithms, an algorithm for shortest paths on euclidean graphs, and the Fibonacci heap implementation of Dijkstra's algorithm. Chapter 4 presents the basic results on trees and acyclic digraphs, a minimum spanning tree algorithm based on Fibonacci heaps, and includes many applications, such as to register allocation, deadlock avoidance, and merge and search trees.

Chapter 5 gives an especially thorough introduction to depth first search and the classical graph structure algorithms based on depth first search, such as block and strong component detection. Chapter 6 introduces both the theory of connectivity and network flows, and shows how connectivity and diverse routing problems may be solved using flow techniques. Applications to reliable routing in unreliable networks, and to multiprocessor scheduling are given.

Chapters 7 and 8 introduce coloring, matching, vertex and edge covers, and allied concepts. Applications to secure (zero-knowledge) communication, the

design of deadlock free systems, and optimal parallel algorithms are given. The Edmonds matching algorithm, introduced for bipartite graphs in Chapter 1, is presented here in its general form.
Chapter 9 presents a variety of parallel algorithms on different architectures, such as systolic arrays, tree processors, and hypercubes, and for the shared memory model of computation as well. Chapter 10 presents the elements of complexity theory, including an introduction to the complexity of random and parallel algorithms.

Table of Contents

1. INTRODUCTION TO GRAPH THEORY

1-1 BASIC CONCEPTS

Graphs are mathematical objects that can be used to model networks, data structures, process scheduling, computations, and a variety of other systems where the relations between the objects in the system play a dominant role. We will consider graphs from several perspectives: as mathematical entities with a rich and extensive theory; as models for many phenomena, particularly those arising in computer systems; and as structures which can be processed by a variety of sophisticated and interesting algorithms. Our objective in this section is to introduce the terminology of graph theory, define some familiar classes of graphs, illustrate their role in modelling, and define when a pair of graphs are the same.

TERMINOLOGY

A *graph* G(V,E) consists of a set V of elements called *vertices* and a set E of unordered pairs of members of V called *edges*. We refer to Figure 1-1 for a geometric presentation of a graph G. The vertices of the graph are shown as points, while the edges are shown as lines connecting pairs of points. The cardinality of V, denoted |V|, is called the *order* of G, while the cardinality of E, denoted |E|, is called the *size* of G. When we wish to emphasize the order and size of the graph, we refer to a graph containing p vertices and q edges as a (p,q) graph. Where we wish to emphasize the dependence of the set V of vertices on the graph G, we will write V(G) instead of V, and we use E(G) similarly. The graph consisting of a single vertex is called the *trivial graph*.

Figure 1-1 here

We say a vertex u in V(G) is *adjacent to* a vertex v in V(G) if {u,v} is an edge in E(G). Following a convention commonly used in graph theory, we will denote the edge between the pair of vertices u and v by (u,v). We call the vertices u and v the *endpoints* of the edge (u,v), and we say the edge (u,v) is *incident with* the vertices u and v. Given a set of vertices S in G, we define the *adjacency set* of S, denoted *ADJ(S)*, as the set of vertices adjacent to some vertex in S. A vertex with no incident edges is said to be *isolated*, while a pair of edges incident with a common vertex are said to be *adjacent*.

The *degree* of a vertex v, denoted by deg(v), is the number of edges incident with v. If we arrange the vertices of G, $v_1,...,v_n$, so their degrees are in nondecreasing order of magnitude, the sequence $(\deg(v_1),...,\deg(v_n))$ is called the *degree sequence* of G. We denote the *minimum degree* of a vertex in G by min(G) and the *maximum degree* of a vertex in G by max(G). There is a simple relationship between the degrees of a graph and the number of edges.

THEOREM (DEGREE SUM) The sum of the degrees of a graph G(V,E) satisfies

$$\sum_{i=1}^{|V|} \deg(v_i) = 2|E|.$$

The proof follows immediately from the observation that every edge is incident with exactly two vertices. Though simple, this result is frequently useful in extending local estimates of the cost of an algorithm in terms of vertex degrees to global estimates of algorithm performance in terms of the number of edges in a graph.

A *subgraph* S of a graph G(V,E) is a graph S(V',E') such that V' is contained in V, E' is contained in E, and the endpoints of any edge in E' are also in V'. A subgraph is said to *span* its set of vertices. We call S a *spanning subgraph* of G if V' equals V. We call S an *induced subgraph* of G if whenever u and v are in V' and (u,v) is in E, (u,v) is also in E'. We use the notation G - v, where v is in V(G), to denote the induced subgraph of G on V-{v}. Similarly, if V' is a subset of V(G), then G - V' denotes the induced subgraph on V - V'. We use the notation G - (u,v), where (u,v) is in E(G), to denote the subgraph G(V,E - {(u,v)}). If we add a new edge (u,v), where u and v are both in V(G) to G(V,E), we obtain the graph G(V, E U {(u,v)}), which we will denote by G(V,E) U (u,v). In general, given a pair of graphs $G(V_1,E_1)$ and $G(V_2,E_2)$, their *union* $G(V_1,E_1)$ U $G(V_2,E_2)$ is the graph $G(V_1 \cup V_2, E_1 \cup E_2)$. If $V_1 = V_2$ and E_1 and E_2 are disjoint, the union is called the *edge sum* of $G(V_1,E_1)$ and $G(V_2,E_2)$. The *complement* of a graph G(V,E), denoted by G^c, has the same set of vertices as G, but a pair of vertices are adjacent in the complement if and only if the vertices are not adjacent in G.

We define a *path* from a vertex u in G to a vertex v in G as an alternating sequence of vertices and edges,

$$v_1, e_1, v_2, e_2, \ldots e_{k-1}, v_k,$$

where $v_1 = u$, $v_k = v$, all the vertices and edges in the sequence are distinct, and successive vertices v_i and v_{i+1} are endpoints of the intermediate edge e_i. If we relax the definition to allow repeating vertices in the sequence, we call the resulting structure a *trail*. If we relax the definition further to allow both repeating edges and vertices, we call the resulting structure a *walk*. If we relax the definition of a path to allow the first and last vertices (only) to coincide, we call the resulting closed path a *cycle*. A graph consisting of a cycle on n vertices is denoted by C(n). If the first and last vertices of a trail coincide, we call the resulting closed trail a *circuit*. The *length* of a path, trail, walk, cycle, or circuit is its number of edges.

We say a pair of vertices u and v in a graph are *connected* if and only there is a path from u to v. We say a graph G is *connected* if and only if every pair of vertices in G are connected. We call a connected induced subgraph of G of maximal order a *component* of G. Thus, a connected graph consists of a single component. The graph in Figure 1-1 has two components. A graph that is not

connected is said to be *disconnected*. If all the vertices of a graph are isolated, we say the graph is *totally disconnected*.

A vertex whose removal increases the number of components in a graph is called a cut-vertex. An edge whose removal does the same thing is called a *bridge*. A graph with no cut-vertex is said to be *nonseparable* (or *biconnected*). A maximal nonseparable subgraph of a graph is called a *block* (*biconnected component* or *bicomponent*). In general, the *vertex connectivity* (*edge connectivity*) of a graph is the minimum number of vertices (edges) whose removal results in a disconnected or trivial graph. We call a graph G k-connected or k-vertex connected (k-edge connected) if the vertex (edge) connectivity of G is at least k.

If G is connected, the path of least length from a vertex u to a vertex v in G is called the *shortest path* from u to v, and its length is called the *distance* from u to v. The *eccentricity* of a vertex v is defined as the distance from v to the most distant vertex from v. A vertex of minimum eccentricity is called a *center*. The eccentricity of a center of G is called the *radius* of G, and the maximum eccentricity among all the vertices of G is called the *diameter* of G. We can define the nth *power* of a connected graph G(V,E), denoted by G^n as follows: $V(G^n) = V(G)$, and an edge (u,v) is in $E(G^n)$ if and only if the distance from u to v in G is at most n. G^2 and G^3 are called the *square* and *cube* of G, respectively.

If we impose directions on the edges of a graph, interpreting the edges as ordered rather than unordered pairs of vertices, we call the corresponding structure a *directed graph* or *digraph*. In contrast and for emphasis, we will sometimes refer to a graph as an *undirected graph*. We will follow the usual convention in graph theory of denoting an edge from a vertex u to a vertex v by (u,v), leaving it to the context to determine whether the pair is to be considered ordered (directed) or not (undirected). The first vertex u is called the *initial vertex* or *initial endpoint* of the edge, and the second vertex is called the *terminal vertex* or *terminal endpoint* of the edge. If G is a digraph and (u,v) an edge of G, we say the initial vertex u is *adjacent to* v, and the terminal vertex v is *adjacent from* u. We call the number of vertices adjacent to v the in-degree of v, denoted indeg(v), and the number of vertices adjacent from v the out-degree of v, denoted outdeg(v). The Degree Sum Theorem for graphs has the obvious digraph analog.

THEOREM (DIGRAPH DEGREE SUM) Let G(V,E) be a digraph; then

$$\sum_{i=1}^{|V|} \text{indeg}(v_i) = \sum_{i=1}^{|V|} \text{outdeg}(v_i) = |E|.$$

Generally, the terms we have defined for undirected graphs have straightforward analogs for directed graphs. For example, the paths, trails, walks, cycles, and circuits of undirected graphs are defined similarly for directed graphs, with the obvious refinement that pairs of successive vertices of the defining sequences must determine edges of the digraph. That is, if the defining sequence of the directed walk (path, etc.) includes a subsequence

v_i, e_i, vi+1 , then e_i must equal the directed edge (v_i, v_{i+1}). If we relax this restriction and allow e_i to equal either (v_i, v_{i+1}) or (v_{i+1}, v_i), we obtain a *semiwalk* (*semipath*, *semicycle*, and so on).

A digraph G(V,E) is called *strongly connected*, if there is a (directed) path between every pair of vertices in G. A vertex u is said to be *reachable from* a vertex v in G, if there is a directed path from v to u in G. The digraph obtained from G by adding the edge (v,u) between any pair of vertices v and u in G whenever u is reachable from v (and (v,u) is not already in E(G)) is called the *transitive closure* of G.

There are several other common generalizations of graphs. For example, in a *multigraph*, we allow more than one edge between a pair of vertices. In contrast, an ordinary graph that does not allow parallel edges is sometimes called a *simple graph*. In a *loopgraph*, both endpoints of an edge may be the same, in which case such an edge is called a *loop* (or self-loop). If we allow both undirected and directed edges, we obtain a so-called *mixed graph*.

SPECIAL GRAPHS

Various special graphs occur repeatedly in practice. We will introduce the definitions of some of these here, and examine them in more detail in later sections.

A graph containing no cycles is called an *acyclic graph*. A directed graph containing no directed cycles is called an *acyclic digraph*, or sometimes a Directed Acyclic Graph (DAG). Perhaps the most commonly encountered special undirected graph is the *tree*, which we define as a connected, acyclic graph. An arbitrary acyclic graph is called a *forest*. Thus, the components of a forest are trees. We will consider trees and acyclic digraphs in Chapter 4.

A graph of order N in which every vertex is adjacent to every other vertex is called a *complete graph*, and is denoted by K(N). Every vertex in a complete graph has the same full degree. More generally, a graph in which every vertex has the same, not necessarily full, degree is called a *regular graph*. If the common degree is r, the graph is called *regular of degree r*.

A graph that contains a cycle which spans its vertices is called a *hamiltonian graph*. These graphs are the subject of an extensive literature revolving around their theoretical properties and the algorithmic difficulties involved in efficiently recognizing them. A graph that contains a circuit which spans its edges is called an *eulerian graph*. Unlike hamiltonian graphs, eulerian graphs are easy to recognize. They are merely the connected graphs all of whose degrees are even. We will consider them later in this chapter.

A graph G(V,E) (or G(V_1,V_2,E)) is called a *bipartite graph* or *bigraph* if its vertex set V is the disjoint union of sets V_1 and V_2, and every edge in E has the form (v_1,v_2), where v_1 î V_1 and v_2 î V_2. A *complete bipartite graph* is a bigraph in which every vertex in V_1 is adjacent to every vertex in V_2. A complete bigraph depends only on the cardinalities M and N of V_1 and V_2 respectively, and so is denoted by K(M,N). Generally, we say a graph G(V,E) or

$G(V_1,...,V_k,E)$ is k-partite if the vertex set V is the union of k disjoint sets $V_1,...,V_k$, and every edge in E is of the form (v_i,v_j), for vertices v_i î V_i and v_j î V_j, V_i and V_j distinct. A complete k-partite graph is defined similarly. We refer to Figure 1-2 for an example.

Figure 1-2 here

GRAPHS AS MODELS

We will describe many applications of graphs in later chapters. The following are illustrative.

Assignment Problem

Bipartite graphs can be used to model problems where there are two kinds of entities, and each entity of one kind is related to a subset of entities of the other. For example, one set may be a set V_1 of employees and the other a set V_2 of tasks the employees can perform. If we assume each employee can perform some subset of the tasks and each task can be performed by some subset of the employees, we can model this situation by a bipartite graph $G(V_1,V_2,E)$, where there is an edge between v_1 in V_1 and v_2 in V_2 if and only if employee v_1 can perform task v_2.

We could then consider such problems as determining the smallest number of employees who can perform all the tasks, which is equivalent in graph-theoretic terms to asking for the smallest number of vertices in V_1 that together are incident to all the vertices in V_2, a so-called covering problem (see Chapter 8). Or, we might want to know how to assign the largest number of tasks, one task per employee and one employee per task, a problem which corresponds graph-theoretically to finding a maximum size regular subgraph of degree one in the model graph, the so-called *matching problem*. Section 1-3 gives a condition for the existence of matchings spanning V_1, and Section 1-5 and Chapters 8 and 9 give algorithms for finding maximum matchings on graphs.

Data Flow Diagrams

We can use directed bipartite graphs to model the flow of data between the operators in a program, employing a *data flow diagram*. For this diagram, let the data (program variables) correspond to the vertices of one part V_1 of the bigraph $G(V_1,V_2)$, while the operators correspond to the other part V_2. We include an edge from a datum to an operator vertex if the corresponding datum is an input to the corresponding operator. Conversely, we include an edge from an operator vertex to a datum, if the datum is an output of the operator. A bipartite representation of the data flow of a program is shown in Figure 1-3.

Figure 1-3 here

A data flow diagram helps in analyzing the intrinsic parallelism of a program. For example, the length of the longest path in a data flow diagram

determines the shortest completion time of the program, provided the maximum amount of parallelism is used. In the example, the path X(+:1)P(DIV:2)Q(-:4)S(DIV:6)U is a longest path; so the program cannot possibly be completed in less than four steps (sequential operations). A minimum length parallel execution schedule is {1}, {2, 3} concurrently, {4, 5} concurrently, and {6}, where the numbers indicate the operators.

GRAPH ISOMORPHISM

For any mathematical objects, the question of their equality or identity is fundamental. For example, a pair of fractions (which may look different) are the same if their difference is zero. Just like fractions, a pair of graphs may also look different but actually have the same structure. Thus, the graphs in Figure 1-4 are nominally different because the vertices v_1 and v_2 are adjacent in one of the graphs, but not in the other. However, if we ignore the names or labels on the vertices, the graphs are clearly structurally identical. Structurally identical graphs are called Isomorphic, from the Greek words iso for "same" and morph for "form." Formally, we define a pair of graphs $G_1(V,E)$ and $G_2(V,E)$ to be *isomorphic* if there is a one-to-one correspondence (or mapping) M between $V(G_1)$ and $V(G_2)$ such that u and v are adjacent in G_1 if and only if the corresponding vertices M(u) and M(v) are adjacent in G_2. See Figure 1-5 for another example.

Figures 1-4 and 1-5 here

To prove a pair of graphs are isomorphic, we need to find an isomorphism between the graphs. The brute force approach is to exhaustively test every possible one-to-one mapping between the vertices of the graphs, until we find a mapping that qualifies as an isomorphism, or determine there is none. See Figure 1-6 for a high-level view of such a search algorithm. Though methodical, this method is computationally infeasible for large graphs. For example, for graphs of order N, there are a priori N! distinct possible 1-1 mappings between the vertices of a pair of graphs; so examining each would be prohibitively costly. Though the most naive search technique can be improved, such as in the backtracking algorithm in Chapter 2, all current methods for the problem are inherently tedious.

Figure 1-6 here

A *graphical invariant* (or *graphical property*) is a property associated with a graph G(V,E) that has the same value for any graph isomorphic to G. One may fortuitously succeed in finding an invariant a pair of graphs do not share, thus establishing their nonisomorphism. Of course, two graphs may agree on many invariants and still be nonisomorphic, although the likelihood of this occurring decreases with the number of their shared properties. The graphs in Figure 1-7 agree on several graphical properties: size, order, and degree sequence. However, the subgraph induced in $G_2(V,E)$ by the vertices of degree 2 is regular of degree 0, while the corresponding subgraph in $G_1(V,E)$ is regular of degree 1. This corresponds to a graphical property the graphs do not share; so the graphs are not isomorphic. The graphs in Figure 1-8 also agree on size, order, and degree sequence. However, $G_1(V,E)$ appears to have

cycles only of lengths 4, 6, and 8, while $G_2(V,E)$ has cycles of length 5. One can readily show that G_1 is bipartite while G_2 is not, so these graphs are also nonisomorphic.

Figures 1-7 and 1-8 here

1-2 REPRESENTATIONS

There are a variety of standard data structure representations for graphs. Each representation facilitates certain kinds of access to the graph but makes complementary kinds of access more difficult. We will describe the simple static representations first. The linked representations are more complicated, but more economical in terms of storage requirements, and they facilitate dynamic changes to the graphs.

1-2-1 STATIC REPRESENTATIONS

The standard static representations are the |V|x|V| adjacency matrix, the edge list, and the edge incidence matrix.

ADJACENCY MATRIX

We define the *adjacency matrix* A of a graph G(V,E) as the |V|x|V| matrix:

$$A(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is in } E(G) \\ 0 & \text{if } (i,j) \text{ is not in } E(G). \end{cases}$$

The adjacency matrix defines the graph completely in $O(|V|^2)$ bits. We can define a corresponding data type Graph_Adjacency or Graph as follows. We assume that the vertices in V are identified by their indices 1..|V|.

```
type   Graph = record
                 |V|: Integer Constant
                 A(|V|,|V|): 0..1
               end
```

Following good information-hiding practice, we package all the defining information, the order |V| (or |V(G)|) included, in one data object.

Let us consider how well this representation of a graph facilitates different graph access operations. The basic graph access operations are

(1) Determine whether a given pair of vertices are adjacent, and

(2) Determine the set of vertices adjacent to a given vertex.

Since the adjacency matrix is a direct access structure, the first operation takes O(1) time. On the other hand, the second operation takes O(|V|) time, since it entails sequentially accessing a whole row (column) of the matrix.

We define the *adjacency matrix representation of a digraph* G in the same way as for an undirected graph:

$$A(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is in } E(G) \\ 0 & \text{if } (i,j) \text{ is not in } E(G). \end{cases}$$

While the adjacency matrix for a graph is symmetric, the adjacency matrix for a digraph is asymmetric.

Matrix operations on the matrix representations of graphs often have graphical interpretations. For example, the powers of the adjacency matrix have a simple graphical meaning for both graphs and digraphs.

THEOREM (ADJACENCY POWERS) If A is the adjacency matrix of a graph (or digraph) G(V,E), then the (i,j) element of A^k equals the number of distinct walks from vertex i to vertex j with k edges.

The smallest power k for which $A^k(i,j)$ is nonzero equals the distance from vertex i to vertex j. Since the expansion of $(A + I)^k$ contains terms for every positive power of A less than or equal to k, its (i,j) entry is nonzero if and only if there is a walk (path) of length k or less from i to j.

The *reachability matrix* R for a digraph G is defined as

$$R(i,j) = \begin{cases} 1 & \text{if there is a path from vertex i to vertex j,} \\ 0 & \text{otherwise.} \end{cases}$$

We can compute R in $O(|V|^4)$ operations using the Adjacency Powers Theorem, but can do it much more quickly using methods we shall see later.

EDGE LIST

The *edge list* representation for a graph G is merely the list of the edges in G, each edge being a vertex pair. The pairs are unordered for graphs and ordered for digraphs. A corresponding data type Graph_Edge_List or simply Graph is

```
type  Graph = record
                |V|: Integer Constant
                |E|: Integer Constant
                Edges(|E|,2): 1..|V|
              end
```

For completeness, we include both the order |V| and the size |E|. The array Edges contains the edge list for G. (Edges(i,1), Edges(i,2)) is the i^{th} edge of G, i = 1..|E|.

The storage requirements for the list are O(|E|) bits, or strictly speaking, O(|E| log |V|) bits since it takes O(log |V|) bits to uniquely identify one of |V| vertices. If the edges are listed in lexicographic order, vertices i and j can be tested for adjacency in O(log |E|) steps (assuming comparisons take O(1) time). The set of vertices adjacent to a given vertex v can be found in time O(log(|E|) + deg(v)), in contrast with the O(|V|) time required for an adjacency matrix.

INCIDENCE MATRIX

The incidence matrix is a less frequently used representation. Let G be a (p,q) undirected graph. If we denote the vertices by $v_1,...,v_p$ and the edges by $e_1,...,e_q$, we can define the *incidence matrix* B of G as the pxq matrix:

$$B(i,j) = \begin{cases} 1 & \text{if vertex } v_i \text{ is incident with edge } e_j, \\ 0 & \text{otherwise.} \end{cases}$$

There is a well-known relationship between the incidence matrix and the adjacency matrix. If we define the *degree matrix* D of a graph G as the |V| by |V| matrix whose diagonal entries are the degrees of the vertices of G and whose off-diagonal entries are all zero, it is easy to prove the following:

THEOREM (INCIDENCE TRANSPOSE PRODUCT) Let G be a nonempty graph, with incidence matrix B, adjacency matrix A, and degree matrix D. Then $BB^t = A + D$.

We define the *incidence matrix* B *of the digraph* G as the p x q matrix:

$$B(i,j) = \begin{cases} +1 & \text{if vertex } v_i \text{ is incident with edge } e_j \\ & \text{and } e_j \text{ is directed out of } v_i, \\ -1 & \text{if vertex } v_i \text{ is incident with edge } e_j, \\ & \text{and } e_j \text{ is directed into } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

The following theorem is easy to establish using the Binet-Cauchy Theorem on the determinant of a product of matrices.

THEOREM (SPANNING TREE ENUMERATION) Let G(V,E) be an undirected connected graph, and let B be the incidence matrix of a digraph obtained by assigning arbitrary directions to the edges of G. Then, the number of spanning trees of G equals the determinant of BB^t.

1-2-2 DYNAMIC REPRESENTATIONS

The *adjacency list* representation for a graph (or digraph) G(V,E) gives for each vertex v in V(G), a list of the vertices adjacent to v. We denote the list of vertices adjacenct to v by ADJ(v). Figure 1-9 gives an example. The details of the representation vary depending on how the vertices and edges are represented.

Figure 1-9 here

Vertex Representation

The vertices can be organized in

(1) A linear array,
(2) A linear linked list, or
(3) A pure linked structure.

In each case, the set of adjacent vertices ADJ(v) at each vertex v is maintained in a linear linked list; the list header for which is stored in the record representing v. Each of the three types of vertex organization are illustrated for the digraph in Figure 1-9 in Figure 1-10.

Figure 1-10 here

In the *linear array* organization, each vertex is represented by a component of a linear array, which acts as the header for the list of edges incident at the vertex. In the *linear list* organization, the vertices are stored in a linked list, each component of which, for a vertex v, also acts as the header for the list of edges in ADJ(v). Refer to Figure 1-10a and b for illustrations.

In the *pure linked* organization, a distinguished vertex serves as an entry vertex to the graph or digraph, and an Entry Pointer points to that entry vertex. The remaining vertices are then accessed solely through the links provided by the edges of the graph. The linked representation for a binary search tree is an example. If not every vertex is reachable from a single entry vertex, a list of enough entry pointers to reach the whole graph is used. Refer to Figure 1-10c for an illustration.

Edge Representation

The representation of edges is affected by whether

(1) The adjacency list represents a graph or digraph, and whether
(2) The representative of an edge is shared by its endpoint vertices or not.

In the case of an undirected graph, the endpoints of an edge (v_i,v_j) play a symmetric role, and so should be equally easily accessible

from both ADJ(v_i) and ADJ(v_j). The most natural approach is to represent the edge redundantly under both its endpoints, as one does for an adjacency matrix. Alternatively, we can let both lists ADJ(v_i) and ADJ(v_j) share the representative of the edge. The adjacency list entry for each edge then merely points to a separate record which represents the edge. This shared record contains pointers to both its endpoints and any data or status fields appropriate to the edge. Figure 1-11 illustrates this approach.

Figure 1-11 here

The adjacency list for a digraph, on the other hand, typically maintains an edge representative (v_i,v_j) on ADJ(v_i) but not on ADJ(v_j). But, if we wish to facilitate quick access to both adjacency-to vertices and adjacency-from vertices, we can use a shared edge representation that lets us access an edge from both ADJ(v_i) and ADJ(v_j). Refer also to Figure 1-11.

Positional Pointers

Many of the graph processing algorithms we will consider process the adjacency lists of the graph in a nested manner. That is, we first process a list ADJ(v_i) partially, up to some edge e_1. Then, we start processing another edge list ADJ(v_j). We process that list up to an edge e_2. But, eventually, we return to processing the list ADJ(v_i), continuing with the next edge after the last edge processed there, e_1. In order to handle this kind of processing, the algorithm must remember for each list the last edge on the list that it processes.

We will define a *positional pointer* at each vertex, stored in the header record for the vertex, to support this kind of nested processing. The positional pointer will initially point to the first edge on its list and be advanced as the algorithm processes the list. Using the positional pointers, we can define a Boolean function Next(G,u,v) which returns in v the next edge or neighbor of u on ADJ(u), and fails if there is none. Successive calls to Next(G,u,v) return successive ADJ(u) elements, starting with the element indicated by the initial position of the positional pointer. Eventually, Next returns the last element. The subsequent call of Next(G,u,v) fails, returning a nil pointer in v and setting Next to False. If we call Next(G,u,v) again, it responds in a cyclic manner and returns a pointer to the head of ADJ(u).

Data Types

We can define data types corresponding to each of the adjacency list representations we have described. We will illustrate this for the linear array representation of a graph by defining a data type Graph. First, we will define the constituent types, Vertex and Edge.

We define the type Vertex as follows. We assume the vertices are indexed from 1..|V|. Each vertex will be represented by a component of an array Head(|V|), the array components of which will be records of type Vertex. The representative of the i^{th} vertex will be stored in the i^{th} position of Head. The type definition is

```
type  Vertex = record
                 Vertex data fields: Unspecified
                 Successor: Edge pointer
                 Positional Pointer: Edge pointer
               end
```

Successor(v), for each vertex v, is the header for ADJ(v) and points to the first element on ADJ(v). Positional Pointer(v) is used for nested processing as we have indicated.

The Edge type is defined as

```
type  Edge =  record
                Edge data fields: Unspecified
                Neighboring Vertex: 1..|V|
                Successor: Edge pointer
              end
```

Neighboring Vertex gives the index of the other endpoint of the edge. Successor just points to the next edge entry on the list.

The overall type Graph is then defined as

```
type   Graph = record
                 |V|: Integer Constant
                 Head(|V|): Vertex
               end
```

We can define similar data types for the other graph representations.

1.3 BIPARTITE GRAPHS

We described a model for an employee-task assignment problem in Section 1-1 and considered the problem of assigning every employee to a task, with one task per employee and one employee per task. We can analyze this problem using the concept of matching. A *matching* M on a graph G(V,E) is a set of edges of E(G) no two of which are adjacent. A matching determines a regular subgraph of degree one. We say a matching *spans* a set of vertices X in G if every vertex in X is incident with an edge of the matching. We call a matching that spans V(G) a *complete* (*spanning* or *perfect*) *matching*. A 1-1 mapping between employees and tasks in the assignment problem corresponds to a spanning matching.

A classical algorithm for constructing maximum matchings uses *alternating paths*, which are defined as paths whose edges alternate between matching and nonmatching edges. Refer to Figure 1-12 for an example. We can use alternating paths in a proof of a simple condition guaranteeing the existence of a spanning matching in a bigraph, and hence a solution to the assignment problem.

Figure 1-12 here

THEOREM (NONCONTRACTING CONDITION FOR EXISTENCE OF SPANNING MATCHINGS IN BIPARTITE GRAPHS) If $G(V_1,V_2,E)$ is bipartite, there exists a matching spanning V_1 if and only if $|ADJ(S)| \geq |S|$ for every subset S of V_1.

The proof of the theorem is as follows. First, we observe that if there exists a matching spanning V_1, $|ADJ(S)|$ must be greater than or equal to $|S|$ for every subset S of V_1. We will establish the converse by contradiction.

Suppose the condition is satisfied, but there is no matching that spans V_1. Let M be a maximum matching on G. By supposition, there must be some vertex v in V_1 not incident with M. Define S as the set of vertices in G reachable from v by alternating paths with respect to M; v is the only unmatched vertex in S. Otherwise, we could obtain a matching larger than M merely by reversing the roles of the edges on the alternating path P from v to another unmatched vertex in S. That is, we could change the matching edges on P into nonmatching edges, and vice versa, increasing the size of the matching.

Define W_i (i = 1,2) as the intersection of S with V_i. Since W_1 and W_2 both lie in S and all of S except v is matched, W_1 - {v} must be matched by M to W_2. Therefore, $|W_2| = |W_1| - 1$. W_2 is trivially contained in $ADJ(W_1)$. Furthermore, $ADJ(W_1)$ is contained in W_2, by the definition of S. Therefore, $ADJ(W_1)$ must equal W_2, so that $|ADJ(W_1)| = |W_2| = |W_1| - 1 < |W_1|$, contrary to supposition. This completes the proof.

We next consider the question of how to recognize whether or not a graph is bipartite. The following theorem gives a simple mathematical characterization of bipartite graphs.

THEOREM (BIPARTITE CHARACTERIZATION) A (nontrivial) graph G(V,E) is bipartite if and and only if it contains no cycles of odd length.

The proof of this theorem is straightforward.

Despite its mathematical appeal, the theorem does not provide an efficient way to test whether a graph is bipartite. Indeed, a search algorithm based on the theorem that tested whether every cycle was even would be remarkably inefficient, not because of the difficulty of generating cycles and testing the number of their edges, but because of the sheer volume of cycles that have to be tested. Figure 1-13 overviews an exhaustive search algorithm that systematically generates and tests every combinatorial object in the graph (here, cycles) which must satisfy a certain property (here, not having odd length) in order for a given property (here, bipartiteness) to be true for the graph as a whole. The performance of the algorithm is determined by how often its search loop is executed, which in turn is determined by how many test objects the graph contains. In this case, the number of test objects (cycles) is typically exponential in the number of vertices in the graph.

Figure 1-13 here

Another search algorithm to test for bipartiteness follows immediately from the definition: merely examine every possible partition of V(G) into disjoint parts V_1 and V_2, and test whether or not the partition is a bipartite one. Once again, the difficulty that arises is not in generating and testing the partitions but in examining the exponential number of partitions that occur.

We now describe an efficient bipartite recognition algorithm. The idea of the algorithm is quite simple. Starting with an initial vertex u lying in part V_1, we fan out from u, assigning subsequent vertices to parts (V_1 or V_2) which are determined by the initial assignment of u to V_1. Obviously, the graph is nonbipartite only if some vertex is forced into both parts.

A suitable representation for G is a standard linear array, where the Header array is H(|V|) and the array components are records of type

```
type  Vertex = record
                Part: 0..2
                Positional Pointer: Edge pointer
                Successor: Edge pointer
               end
```

The entries on the adjacency lists have the form

```
type  Edge = record
               Neighbor: 1..|V|
               Successor: Edge pointer
             end
```

We call the type of the overall structure *Graph*.

The function Bipartite uses a utility Get(u) which returns a vertex u for which Part(u) is zero or fails. The outermost loop of the procedure processes the successive components of the graph. The middle loop processes the vertices within a component. The innermost loop processes the immediate neighbors of a given vertex. The algorithm takes time O(|V| + |E|). This is the best possible performance since any possible algorithm must inspect every edge because the inclusion of even a single additional edge could alter the status of the graph from bipartite to nonbipartite.

```
Function Bipartite (G)

(* Returns the bipartite status of G in Bipartite *)

var G: Graph
    Component: Set of 1..|V|
    Bipartite: Boolean function
    v,u: 1..|V|

Set Bipartite to True
Set Part(v)   to 0, for every v in V(G)
```

```
while Get(u) do

  Set Component to {u}
  Set Part(u)   to  1

  while  Component <> Empty  do

     Remove a vertex v from Component

     for every neighbor w of v do

        if    Part(w) = 0

        then  Add w to Component
              Set Part(w) to 3 - Part(v)

        else  if    Part(v) = Part (w)
              then  Set Bipartite to False
                    Return

End_Function_Bipartite
```

1.4 REGULAR GRAPHS

A graph G(V,E) is *regular of degree r* if every vertex in G has degree r. Regular graphs are encountered frequently in modeling. For example, many parallel computers have interconnection networks which are regular. The five regular polyhedra (the tetrahedron, cube, octahedron, dodecahedron, and icosahedron) also determine regular graphs. The following theorems summarize their basic properties.

THEOREM (REGULAR GRAPHS) If G(V,E) is a (p,q) graph which is regular of degree r, then

$$pr = 2q.$$

If G is also bipartite, then

$$|V_1| = |V_2|.$$

Interestingly, every graph G can be represented as an induced subgraph of a larger regular graph.

THEOREM (EXISTENCE OF REGULAR SUPERGRAPH) Let G(V,E) be a graph of maximum degree M. Then, there exists a graph H which is regular of degree M and that contains G as an induced subgraph.

The following result of Koenig is classical. It is an example of a so-called *factorization* result (see also Chapter 8).

THEOREM (PARTITIONING REGULAR BIGRAPHS INTO MATCHINGS) A bipartite graph $G(V_1,V_2,E)$ regular of degree r can be represented as an edge disjoint union of r complete matchings.

We can prove this result using the condition for the existence of spanning matchings in bipartite graphs given in Section 1-3. The proof is by induction on the degree of regularity r. The theorem is trivial for r equal to 1. We will assume the theorem is true for degree r - 1 or less, and then establish it for degree r.

First, we shall prove that V_1 satisfies the condition of the Bipartite Matching Theorem. Let S be a nonempty subset of V_1 and let ADJ(S) denote the adjacency set of S. We will show that $|ADJ(S)| \geq |S|$. Observe that there are r|S| edges emanating from S. By the regularity of G, the number of edges linking ADJ(S) and S cannot be greater than r|ADJ(S)|. Therefore, $r|ADJ(S)| \geq r|S|$, or $|ADJ(S)| \geq |S|$, as required. Therefore, we can match V_1 with a subset of V_2.

Since G is regular, $|V_1| = |V_2|$. Therefore, the matching is actually a perfect matching. We can remove this matching to obtain a reduced regular graph of degree r - 1. It follows by induction that $G(V_1,V_2,E)$ can be covered by a union of r edge disjoint matchings. This completes the proof.

Regular graphs are prominent in *extremal graph theory*, the study of graphs that attain an extreme value of some numerical graphical property under a constraint on some other graphical property. An extensively studied class of extremal regular graphs are the *cages*. Let us define the *girth* of a graph G as the length of a shortest cycle in G. Then, an (r,n)-cage is an r-regular graph of girth n of least order. If r equals 3, then we call the (r,n)-cage an n-cage. Alternatively, an n-cage is a 3-regular (or *cubic*) graph of girth n. Cages are highly symmetric, as illustrated by the celebrated examples of n-cages shown in Figure 1-14, the *Petersen graph*, the unique five-cage, and in Figure 1-15, the *Heawood graph*, the unique six-cage.

Figure 1-14 and 1-15 here

1-5 MAXIMUM MATCHING ALGORITHMS

There are a number of interesting algorithms for finding maximum matchings on graphs. This section describes four of them, each illustrating a different methodology: maximum network flows, alternating paths, integer programming, or randomization.

1-5-1 MAXIMUM FLOW METHOD (BIGRAPHS)

A maximum matching on a bipartite graph can be found by modeling the problem as a network flow problem and finding a maximum flow on the model network. The theory of network flows is elaborated in Chapter 6. Basically, a flow network is a digraph whose edges are assigned capacities, and each edge is

interpreted as a transmission link capable of bearing a limited amount of "traffic flow" in the direction of the edge. One can then ask for the maximum amount of traffic flow that can be routed between a pair of vertices in such a network. This maximum traffic flow value and a set of routes that realize it can be found using maximum flow algorithms. Efficient maximum flow algorithms for arbitrary graphs are given in Chapter 6.

To model the bipartite matching problem as a maximum network flow problem, we can use the kind of flow network shown in Figures 1-16 and 1-17. We transform the bigraph to a flow network by introducing vertices s and t, which act as the source and sink of the traffic to be routed through the network; we then direct the edges of the bigraph downwards and assign a capacity limit of one unit of traffic flow per edge. A flow maximization algorithm is then used to find the maximum amount of traffic that can be routed between s and t. The internal (bipartite) edges of the flow paths determine a maximum matching on the original bigraph. In the example, the traffic is routed on the paths $s\text{-}v_1\text{-}v_4\text{-}t$, $s\text{-}v_2\text{-}v_6\text{-}t$, and $s\text{-}v_3\text{-}v_5\text{-}t$, one unit of traffic flow per path. The corresponding maximum matching is (v_1,v_4), (v_2,v_6), and (v_3,v_5).

Figure 1-16 and 1-17 here

1-5-2 ALTERNATING PATH METHOD (BIGRAPHS)

The method of alternating paths is a celebrated search tree technique for finding maximum matchings. A general method for arbitrary graphs is given in Chapter 8, but it simplifies greatly when the graph is bipartite, which is the case considered here.

Let G(V,E) be a graph and let M be a matching on G. A path whose successive edges are alternately in M and outside of M is called an *alternating path with respect to* M. We call a vertex not incident with an edge of M a *free* (or *exposed*) *vertex relative to* M. An alternating path with respect to M whose first and last vertices are free is called an *augmenting path with respect to* M. We can use these paths to transform an existing matching into a larger matching by simply interchanging the matching and nonmatching edges on an augmenting path. The resulting matching has one more edge than the original matching. Refer to Figure 1-18 for an example.

Fig. 1-18 here

Augmenting paths be used to enlarge a matching; if they do not exist, a matching must already be maximum.

THEOREM (AUGMENTING PATH CHARACTERIZATION OF MAXIMUM MATCHINGS) A matching M on a graph G(V,E) is a maximum matching if and only if there is no augmenting path in G with respect to M.

The proof follows. We show that if M is not a maximum matching there must exist an augmenting path in G between a pair of free vertices relative to M. The proof is by contradiction. Suppose that M' is a larger matching then M. Let G' denote the symmetric difference of the subgraphs induced by M and M'. That is, the edges of G' are the edges of G which are in either M or M' but not both. By supposition, there are more edges in G' from M' than from M.

Observe that every vertex in G' has degree at most two (in G') since, at most, two edges, one from M and one from M', are adjacent to any vertex of G'. Therefore, the components of G' are either isolated vertices, paths, or cycles. Furthermore, since none of the edges of a given matching are mutually adjacent, each path or cycle component of G' must be an alternating path. In particular, any path of even length and every cycle (all of which are necessarily of even length) must contain an equal number of edges from M and M'. Therefore, if we remove every cycle and every component consisting of an even length path from G', there still remain more edges of M' than of M. Therefore, at least one of the remaining path components must have more edges from M' than M, indeed exactly one more edge. Since the edges of this path alternate between M and M', its first and last edges must lie in M'. Consequently, neither the first nor last vertex of the path can lie in M. Otherwise, the path could be extended, contrary to the maximality of a component. Thus, the component must be an alternating path with respect to M with free endpoints, that is, an augmenting path with respect to M. This completes the proof.

The Characterization Theorem suggests a procedure for finding a maximum matching. We start with an empty matching, find an augmenting path with respect to the matching, invert the role of the edges on the path (matching to nonmatching, and vice versa), and repeat the process until there are no more augmenting paths, at which point the matching is maximum. The difficulty lies in finding the augmenting paths. We shall show how to find them using a search tree technique. The technique is extremely complicated when applied to arbitrary graphs, but simplifies in the special case of bipartite graphs.

ALTERNATING SEARCH TREE

The augmenting path search tree consists of alternating paths emanating from a free vertex (Root), like the tree shown in Figure 1-19. The idea is simply to repeatedly extend the tree until it reaches a free vertex v (other than Root). If the search is successful, the alternating path through the tree from Root to v is an augmenting path. Thus, in the example the tree path from Root to v_9 is augmenting. We can show, conversely, that if the search tree does not reach a free vertex, there is no augmenting path (with Root as a free endpoint) in the graph. The blocked search tree is called a *Hungarian tree*, and its vertices can be ignored in all subsequent searches. When the search trees at every free vertex become blocked, there are no augmenting paths at all, and so the matching is maximum.

Figure 1-19 here

We construct the search tree by extending the tree two alternating edges at a time, maintaining the alternating character of the paths through the search tree (as required in order to eventually find an augmenting alternating path) by always making the extensions either from the root of the tree or from the outermost vertex of some matching edge. That is, suppose that (u,x) is a matching edge already lying in the search tree and that x is the endpoint of (u,x) lying farthest from the root of the search tree, as illustrated in Figure 1-20. Denote an unexplored edge at x by (x,y). If y is free, the path

through the tree from Root to y is an augmenting path, and the search may terminate. Otherwise, if y is a matched vertex, we let its matching edge be (y,z). Then, we can extend the search tree by adding the pair of alternating edges (x,y) and (y,z) to the tree.

Figure 1-20 here

We will refer to search tree vertices as either *outer* or *inner vertices* according to whether their distance, through the search tree, from the root is even or odd, respectively. The tree advances from outer vertices only. If we construct a search tree following these guidelines, the tree will find an augmenting path starting at the exposed vertex Root if there is one, though not necessarily a given augmenting path. We state this as a theorem.

THEOREM (SEARCH TREE CORRECTNESS) Let G(V,E) be an undirected graph and let Root be an exposed vertex in G. Then, the search tree rooted at Root finds an augmenting path at Root if one exists.

To prove this, we argue as follows. Suppose there exists an augmenting path P from Root to an unmatched vertex v, but that the search tree becomes blocked before finding any augmenting path from Root, including P. We will show this leads to a contradiction.

Let z be the last vertex on P, starting with v as the first, which is not in the search tree; z exists since by supposition v is not in the search tree. Let w be the next vertex after z on P, proceeding in the direction from v to Root; w must lie in the search tree, and so must all the vertices on P between w and Root, though not necessarily all the edges of P between w and Root. Furthermore, (w,z) cannot be a matching edge, since every tree vertex is already matched except for Root, which is free. Therefore, (w,z) is an unmatched edge. We will distinguish two cases according to whether w is an outer or an inner vertex.

(1) In the case that w is an outer vertex, the search procedure would eventually extend the search tree from w along the unmatched edge (w,z) to z, unless another augmenting path had been detected previously, contradicting the assumption that z is not in the search tree.

(2) We will show the case where w is an Inner vertex cannot occur. For, since P is alternating, the tree matching edge at w, that is (w,w_1), would be the next edge of P. Therefore, P would necessarily continue through the tree in the direction of (w,w_1). Subsequently, whenever P entered an inner vertex, necessarily via a nonmatching edge, P would have to exit that vertex by the matching tree edge at the vertex, since P is alternating. Furthermore, if P exits the tree at any subsequent point, it must do so for at most a single edge, only from an outer vertex and only via an unmatched edge. If we denote such an edge by (x,y) and x is the vertex where P exits and y is the vertex where P reenters the tree, y cannot also be an outer vertex. For otherwise, the path through the search tree from z, the common ancestor of x and y, to x, plus the edge (x,y), plus the path through the search tree from y to z would together constitute an odd cycle. But, since the graph is a bigraph, it contains no odd cycles by our Characterization Theorem for bigraphs. Consequently, the reentry vertex y must be an inner vertex.

Therefore, just as before, P must leave y along the matching edge in the tree which is incident at y. This pattern continues indefinitely, preventing P from ever reaching Root, which is contrary to the definition of P as an augmenting path. This completes the proof of the theorem.

The Hungarian Trees produced during the search process have the following important property.

THEOREM (HUNGARIAN TREES) Let G(V,E) be an undirected graph. Let H be a Hungarian tree with respect to a matching M on G rooted at a free vertex Root. Then, H may be ignored in all subsequent searches. That is, a maximum matching on G is the union of a maximum matching on H and a maximum matching on G - H.

The proof is as follows. Let M_1 be a maximum matching for G - H, and suppose that M' is an arbitrary matching for G, equal to $M_1' \cup M_H' \cup M_I$, where M_1' is in G - H, M_H' is in H, and M_I satisfies that the intersection of M_I and (G - H) U H is empty. By supposition, $|M_1| \geq |M_1'|$. Furthermore, since every edge in M_I is incident with at least one inner vertex of H; then if I' is the set of inner vertices in H which are incident with M_I, then $|M_I|$ is at most |I'|. Finally, observe that H - I' consists of |I'| + 1 disjoint alternating trees whose inner vertices together comprise I - I', where I denotes the set of inner vertices of H. Since the cardinality of a maximum matching in an alternating tree equals the number of its inner vertices, it follows that $|M_H'|$ is at most |I - I'|. Combining these results, we obtain that $|M'| \leq |M_1'| + |M_H'| + |M_I| \leq |M_1| + |I - I'| + |I'| = |M_1| + |I| = |M_1 \cup M_H|$, which completes the proof.

MAXIMUM MATCHING ALGORITHM

A procedure Maximum_Matching constructs the matching. Its hierarchical structure is

```
Maximum_Matching (G)
   Next_Free (G,Root)
   Find_Augmenting_Tree (G,Root,v)
      Create (Q)
      Enqueue (w,Q)
      Next (G,Head(Q),v)
   Apply_Augmenting_Path (G,Root,v)
   Remove_Tree (G)
   Clear(G)
```

Next_Free returns a free vertex Root, or fails. The procedure Find_Augmenting_Tree then tries to construct an augmenting search tree rooted at Root. If Find_Augmenting_Tree is successful, the free vertex at the terminus of the augmenting path is returned in v, and Apply_Augmenting_Path uses the augmenting path to change the matching by following the search tree (predecessor) pointers from the breakthrough vertex v back to Root, modifying the existing matching accordingly. If Find_Augmenting_Tree fails, the resulting blocked search tree is by definition

a Hungarian tree, and we ignore its vertices during subsequent searches, since they cannot be part of any augmenting path.

We have used a queue based search procedure (called *Breadth First Search*, see Chapter 5) to control the search order, whence the utilities Create (Q) and Enqueue (w,Q). However, other search techniques such as depth first search could have been used. Find_Augmenting_Tree uses Next(G,u,v) to return the next neighbor v of u, and fails when there is none. The utility Clear (G) is used to reset Pred and Positional Pointer back to their initial values for every vertex in G. Remove_Tree (G) removes from G the subgraph induced by a current Hungarian tree in G. The other utilities are familiar. Refer to Figure 1-21 for an example, where an initial nontrivial matching is given in Figure 1-21a.

Figure 1-21 here

The outer-inner terminology used to describe the search process is not explicit in the algorithm. In Find_Augmenting_Tree, w is an outer vertex. These are the only vertices queued for subsequent consideration for extending the search tree. The vertices named v in Find_Augmenting_Tree correspond to what we have called inner vertices. The tree is not extended from them so they are not queued.

We will now describe the data structures for the algorithm. We represent the graph G(V,E) as a linear list with vertices indexed 1..|V|. The components of the list are of type Vertex. Both the search tree and the matching are embedded in the representation of G. Each vertex component heads the adjacency list for the vertex. The type definition is as follows:

```
type  Vertex = record
                 Matching vertex: 0..|V|
                 Pred: Vertex pointer
                 Positional Pointer,
                     Successor: Edge_Entry pointer
               end
```

Matching vertex gives the index of the matching vertex for the given vertex; it is 0 if there is none. Pred points to the predecessor of the given vertex in the search tree or is nil. Pred(v) is nil until v is scanned: so it can be used to detect whether a vertex was scanned previously. Positional Pointer and Successor serve their customary roles as explained in Section 1-2.

We represent the adjacency lists as doubly linked lists with the representative for each edge shared by its endpoints. This organization facilitates the deletion of vertices and edges required when Hungarian trees are deleted from G. The records on the adjacency lists are of type Edge_Entry. Each of these points to its list predecessor, successor, and its associated edge representative. The edge representative itself is of type Edge and is defined by

```
type  Edge =  record
                Endpoints(2): 1..|V|
                Endpoint-pointer(2): Edge_Entry pointer
              end
```

Endpoints(2) gives the indices of the endpoints of the represented edge; while Endpoint-pointer(2) points to the edge entries for these endpoints. We call the overall representation for the graph type Graph. Q packages a queue whose entries identify outer vertices to be processed.

The formal statements of Maximum_Matching and Find_Augmenting_Tree are as follows.

```
Procedure Maximum_Matching (G)

(* Finds a maximum matching in G *)

var  G: Graph
     Root, v: Vertex pointer
     Next_Free, Find_Augmenting_Tree: Boolean function

while  Next_Free (G, Root)  do

   if    Find_Augmenting_Tree (G, Root,v)

   then  Apply_Augmenting_Path (G, Root,v)
         Clear (G)

   else  Remove_Tree (G)

End_Procedure_Maximum_Matching
```

```
Function_Find_Augmenting_Tree (G,Root,v)

(* Tries to find an augmenting tree at Root, returning the
   augmenting vertex in v and fails if there is none. *)

var  G: Graph
     Root, v, w: Vertex pointer
     Q: Queue
     Empty, Next, Find_Augmenting_Tree: Boolean function
     Dequeue: Vertex pointer function

Set Find_Augmenting_Tree to False
Create (Q);  Enqueue (Root, Q)

repeat

  while Next(G,Head(Q),v) do

    if  v Free then  (* Augmenting path found *)
                     Set Pred(v) to Head(Q)
                     Set Find_Augmenting_Tree to  True
                     return
```

```
            else  if v unscanned

                  then (* Extend search tree *)
                       Let w be vertex matched with v
                       Set Pred(w) to v
                       Set Pred(v) to Head(Q)
                       Enqueue(w,Q)

until  Empty(Dequeue(Q))

End_Function_Find_Augmenting_Tree
```

The performance of the matching algorithm is summarized in the following theorem.

THEOREM (MATCHING ALGORITHM PERFORMANCE) Let G(V,E) be an undirected graph. Then, the search tree matching algorithm finds a maximum matching on G in O(|V||E|) steps.

The proof of the theorem is as follows. The performance statement depends strongly on the Hungarian Trees Theorem. Observe that there are at most |V|/2 search phases, where a search phase consists of generating successive search trees, each rooted at a different free vertex, until either one search tree succeeds in finding an augmenting path, or every search tree is blocked. Each such search phase may generate many search trees. Nonetheless, since the Hungarian trees can be ignored in all subsequent searches (even during the same phase), at most |E| edges will be explored during a phase up to the point where one of the searches succeeds or every search fails. Thus, each search phase takes time at most time O(|V| + |E|); so the algorithm takes O(|V||E|) steps overall. This completes the proof.

1-5-3 INTEGER PROGRAMMING MODEL

Integer programming is a much less efficent technique than the previous methods for finding maximum matchings, but it has the advantage of being easy to apply and readily adaptable to more general problems, such as weighted matching, where the objective is to find a maximum weight matching on a weighted graph. The method can also be trivially adapted to maximum degree-constrained matching, where the desired subgraph need not be regular of degree one but only of bounded degree. An application of degree-constrained matching is given in Chapter 8. Despite the simplicity of the method, it suffers from being NP-Complete (see Chapter 10 for this terminology).

The idea is to use a system of linear inequalities for 0-1 variables to model the matching problem, for an arbitrary graph, and then apply the techniques of 0-1 integer programming to solve the model. Let G(V,E) be a graph for which a maximum matching is sought and let A_{ij} be the adjacency matrix of G. Let x_{ij}, i, j (j > i) = 1,...,|V| denote a set of variables which we constrain to take on only 0 or 1 as values. We also constrain the variables; so they behave like indicators of whether an edge is in the matching or not. That is, we restrict each x_{ij} so that it is 0 if (i,j) is not a matching edge and is 1 if (i,j) is a matching edge. Finally, we

define an objective function whose value equals the number of edges in the matching and which is maximized subject to the constraints. The formulation is as follows.

Edge Constraints

$$x_{ij} \leq A_{ij}, \quad \text{for } i, j\ (j > i) = 1,\ldots,|V|$$

Matching Constraints

$$\sum_{j>i}^{|V|} x_{ij} + \sum_{j<i}^{|V|} x_{ji} \leq 1, \quad \text{for } i = 1,\ldots,|V|$$

Objective Function (Matching size)

$$\sum_{i=1}^{|V|} \left(\sum_{j>i} x_{ij} \right)$$

The edge constraints ensure only edges in the graph are allowed as matching edges. The matching constraints ensure there is at most one matching edge at each vertex, as required by the graphical meaning of a matching. The objective function just counts the number of edges in the matching.

Figure 1-22 gives a simple example. The system of inequalities for the example is as follows.

Edge Constraints

$$x_{12} \leq 1, \quad x_{13} \leq 1, \quad x_{23} \leq 1.$$

Matching Constraints

$$x_{12} + x_{13} \leq 1,\ x_{12} + x_{23} \leq 1, \quad x_{23} + x_{13} \leq 1.$$

Objective Function

$$x_{12} + x_{13} + x_{23}$$

Three solutions are possible: $x_{12} = 1$, $x_{13} = 0$, $x_{23} = 0$, and the two symmetric variations of this. Each corresponds to a maximum matching.

Figure 1-22 here

1-5-4 PROBABILISTIC METHOD

Surprisingly, combinatorial problems can sometimes be solved faster and more simply by random or probabilistic methods than by deterministic methods. The

algorithm we describe here converts the matching problem into an equivalent matrix problem, which is then "solved" by randomizing. The method is due to Lovasz and is based on the following theorem of Tutte.

THEOREM (TUTTE MATRIX CONDITION FOR PERFECT MATCHING) Let G(V,E) be an undirected graph. Define a matrix T of indeterminates as follows: for every pair i,j (j > i) = 1, ..., |V|, for which (i,j) is an edge of G, set T(i,j) to an indeterminate $+x_{ij}$ and set T(j,i) to an indeterminate $-x_{ij}$; otherwise, set T(i,j) to 0. Then, G has a complete matching if and only if the determinant of T is not identically zero.

The matrix T is called the *Tutte matrix* of G and its determinant is the *Tutte determinant*. The theorem provides a simple test for the existence of a complete matching in a graph. However, there are computational difficulties involved in applying the theorem.

It is usually straightforward to calculate a determinant. For example, Gaussian Elimination can be used to reduce the matrix to upper triangular form and the determinant is then just the product of the diagonal entries of the upper triangular matrix. This procedure has complexity $O(|V|^3)$. However, Gaussian Elimination cannot be used when the matrix entries are symbolic, as is the case here. In order to evaluate a symbolic, as opposed to a numerical determinant, we apparently must fall back on the original definition of a determinant. Recall that by definition a determinant of a matrix T is a sum of |V|! terms.

$$\sum_{\text{all permutations}} \text{sign}(i_1, \ldots, i_{|V|})T(1,i_1)* \ldots *T(|V|,i_{|V|}),$$

where $(i_1, \ldots, i_{|V|})$ is a permutation of the indices 1,...,|V| and sign(x) returns the sign of the permutation x: +1 if x is an even permutation and -1 if x is an odd permutation. To apply the Tutte condition entails testing if the symbolic polynomial that results from the evaluation of the determinant is identically zero or not. This apparently simple task is really quite daunting. The polynomial has |V|! terms, which is exponential in |V|. Therefore, even though the required evaluation is trivial in a conceptual sense, it is very nontrivial in the computational sense.

Observe that we do not actually need to know the value of the determinant, only whether it is identically zero or not. This suggests the following alternative testing procedure: perform the determinant condition test by randomly assigning values to the symbolic matrix entries and then evaluating the resulting determinant by a numerical technique like Gaussian Elimination.

If the determinant randomly evaluates to nonzero, then the original symbolic determinant must be nonzero; so the graph must have a complete matching. If the determinant evaluates to zero, there are two possibilities. Either it is really identically zero or it is not identically zero, but we have randomly (accidentally) stumbled on a root of the Tutte determinant. Of course, by repeating the random assignment and numerical evaluation process a few times, we can make the risk of accidentally picking a root negligible. In particular, if the determinant randomly evaluates to zero several times, it is almost certainly identically zero. Thus, we can with great confidence

interpret a zero outcome as implying that the graph does not have a complete matching. Thus, positive conclusions (that is, a complete matching exists) are never wrong, while negative conclusions (that is, a complete matching does not exist) are rarely wrong.

The random approach allows us to apply the determinant condition efficiently and with a negligible chance of error. The following procedure Prob_Matching_Test (G) formalizes this approach. Because a subsequent procedure that invokes Prob_Matching_Test will make vertex and edge deletions from the graph, we will use a linear list representation for G instead of the simpler adjacency matrix representation. The vertex type is

```
type  Vertex = record
                 Index: 1..|V|
                 Adjacency_Header: Edge pointer
                 Successor: Vertex pointer
               end
```

G itself is just a pointer to the head of the vertex list. Each vertex has an index field for identification, an Adjacency_Header field, which points to the first element on its edge list, and a Successor field, which points to the succeeding vertex on the vertex list. We can assume the vertex records are ordered according to index. We will use adjacency lists structured in the same manner as for the alternating paths algorithm to facilitate deletions that will be required in a later algorithm.

```
Function Prob_Matching_Test (G)

(* Succeeds if G has a perfect matching, and fails otherwise *)

var  G: Vertex pointer
     X(|V|,|V|), T(|V|,|V|): Integer
     R: Integer Constant
     Prob_Matching_Test: Boolean function

Set T to a Tutte matrix for G
Set Prob_Matching_Test to False

repeat  R  Times

   Set X to a random instance of T

   if    Determinant (X) <> 0
   then  Set Prob_Matching_Test to True
         return

End_Function_Prob_Matching_Test
```

Prob_Matching_Test(G) determines whether or not G has a complete matching, but does not show how to find a complete matching if one exists. But, we can easily design a constructive matching algorithm (for graphs with complete

matchings) using this existence test. The following procedure Prob_Complete_Matching returns a complete matching if one exists. We assume that M is initially empty. The representations are the same as before. Figures 1-23 and 1-24 illustrate the technique.

```
Procedure Prob_Complete_Matching(G, M)

(* Returns a complete matching in M, or the empty set if there
   is none *)

var  G: Vertex pointer
     x: Edge
     M: Set of Edge
     Prob_Matching_Test: Boolean function

if    Prob_Matching_Test(G)

then  Select an edge x in G
      Set G to G - x

      if      Prob_Matching_Test (G)

      then    Prob_Complete_Matching (G, M)

      else    Set M to  M U {x}
              Set G to  G - {Endpoints of x}
              Prob_Complete_Matching (G, M)

End_Procedure_Prob_Complete_Matching
```

Figures 1-23 and 1-24 here

1-6 PLANAR GRAPHS

A graph G(V,E) is called a *planar graph* if it can be drawn or embedded in the plane in such a way that the edges of the embedding intersect only at the vertices of G. Figures 1-25 and 1-26 show planar and nonplanar graphs. Both planar and nonplanar embeddings of K(4) are shown in Figure 1-25, while a partial embedding of K(3,3) is shown in Figure 1-26. One can prove that K(3,3) is nonplanar; so this embedding cannot be completed. For example, the incompletely drawn edge x in the figure is blocked from completion by existing edges regardless of whether we try to draw it through region I or region II. Of course, this only shows that the attempted embedding fails, not that no embedding of K(3,3) is possible, though in fact none is. This section describes the basic theoretical and algorithmic results on planarity. We begin with a simple application.

Figures 1-25 and 1-26 here

MODEL: FACILITY LAYOUT

Consider the following architectural design problem. Suppose that n simple planar areas $A_1,...,A_n$ (of flexible size and shape) are to be laid out next

to each other but subject to the constraint that each of the areas be adjacent to a specified subset of the other areas. The adjacency constraints can be represented using a graph G(V,E) whose vertices represent the given areas A_i and such that a pair of vertices are adjacent in G if and only if the areas they represent are supposed to be adjacent in the layout. The planar layout can be derived as follows:

(1) Construct a planar representation R of G(V,E).

(2) Using R, construct the *planar dual* G' of G as follows:

    (i) Corresponding to each planar region r of R, define a vertex v_r of G', and
    (ii) Let a pair of vertices v_r and v_r' in G' be adjacent if the pair of corresponding regions r and r' share a boundary edge in R.

(3) Construct a planar representation R' of G'.

If step (1) fails, G is nonplanar and the constraints are infeasible. Otherwise, steps (1) and (2) succeed. (Strictly speaking, in step (2ii), if r and r' have multiple boundary edges in common, we connect v_r and v'_r multiply; and if an edge lies solely in one region r, we include a self-loop at v_r in G'. Neither case occurs here.) It can be shown that if G is planar, its dual G' is also; so the planar representation required by step (3) exists. The representation R' is the desired planar layout of the areas $A_1,...,A_n$. The regions of R' correspond to the given areas, and the required area adjacency contraint are satisfied. The process is illustrated in Figure 1-27.

Figure 1-27 here

PLANARITY TESTING

There are two classical tests for planarity. The Theorem of Kuratowski characterizes planar graphs in terms of subgraphs they are forbidden to have, while an algorithm of Hopcroft and Tarjan determines planarity in linear time and shows how to draw the graph if it is planar.

Kuratowski's Theorem uses the notion of homeomorphism, a generalization of isomorphism. We first define *series insertion* in a graph G(V,E) as the replacement of an edge (u,v) of G by a pair of edges (u,z) and (z,v), where z is a new vertex of degree two. That is, we insert a new vertex on an existing edge. We define *series deletion* as the replacement of a pair of existing edges (u,z) and (z,v), where z is a current vertex of degree two, by a single edge (u,v), and delete z. That is, we "smooth away" vertices of degree two. Then, a pair of graphs G_1 and G_2 are said to be *homeomorphic* if they are isomorphic or can be made isomorphic through an appropriate sequence of series insertions and/or deletions.

THEOREM (KURATOWSKI'S FORBIDDEN SUBGRAPH CHARACTERIZATION OF PLANARITY) A graph G(V,E) is planar if and only if G contains no subgraph homeomorphic to either K(5) or K(3,3).

We omit the difficult proof of the theorem. Figures 1-28 and 1-29 illustrate its application. The graph in Figure 1-28 is Petersen's graph. Though this graph does not contains a subgraph isomorphic to either K(5) or K(3,3), it does contain the homeomorph of K(3,3) shown in Figure 1-29; so it is nonplanar.

Figures 1-28 and 1-29 here

Kuratowski's Theorem suggests an exhaustive search algorithm for planarity based on searching for subgraphs homeomorphic to one of the forbidden subgraphs. But, it is not obvious how to do this in polynomial time since the forbidden homeomorphs can have arbitrary orders (see the exercises). Williamson (1984) describes a fairly complicated algorithm based on depth-first search that finds a Kuratowski subgraph in a nonplanar graph in O(|V|) time.

It is sometimes simpler to apply Kuratowski's planarity criterion in a different form. We define *contraction* as the operation of removing an edge (u,v) from a graph G and identifying the endpoints u and v (with a single new vertex uv) so that every edge (other than (u,v)) originally incident with either u or v becomes incident with uv. We say a graph $G_1(V,E)$ is *contractible to* a graph $G_2(V,E)$ if G_2 can be obtained from G_1 by a sequence of contractions.

THEOREM (CONTRACTION FORM OF KURATOWSKI'S THEOREM) A graph G(V,E) is planar if and only if G contains no subgraph contractible to either K(5) or K(3,3).

We can use the Contraction form of Kuratowski's Theorem to directly show Petersen's graph is nonplanar, since it reduces to K(5) if we contract the edges $(v_i,vi+5)$, i = 1,...,5.

DEMOUCRON'S PLANARITY ALGORITHM

We have mentioned that the Hopcroft-Tarjan planarity algorithm tests for planarity in time O(|V|), and also shows how to draw a planar graph. But, instead of describing this complex algorithm, we will present the less efficient, but simpler and still polynomial, algorithm of Demoucron, et al. The algorithm is based on a criterion for when a path in a graph can be drawn through a face of a partial planar representation of the graph.

Let us define a *face* of a planar representation as a planar region bounded by edges and vertices of the representation and containing no graphical elements (vertices or edges) in its interior. The unbounded region outside the outer boundary of a planar graph is also considered a face. Let R be a planar representation of a subgraph S of a graph G(V,E) and suppose we try to extend R to a planar representation of G. Then, certain constraints will immediately impose themselves.

First of all, each component of G - R, being a connected piece of the graph, must, in any planar extension of R, be placed within a face of R. Otherwise, if the component straddled more than one face, any path in the component connecting a pair of its vertices lying in different faces would have to cross

an edge of R, contrary to the planarity of the extension. A further constraint is observed by considering those edges connecting a component c of G - R to a set of vertices W in R. All the vertices in W must lie on the boundary of a single face in R. Otherwise, c would have to straddle more than one face of R, contrary to the planarity of the extension.

These constraints on how a planar representation can be extended are only necessary conditions that must be satisfied by any planar extension. Nonetheless, we will show that we can draw a planar graph merely by never violating these guidelines.

Before describing Demoucron's algorithm, we will require some additional terminology. If G is a graph and R is a planar representation of a subgraph S of G, we define a part p of G relative to R as either

(1) An edge (u,v) in E(G) - E(R) such that u and v in V(R), or

(2) A component of G - R together with the edges (called *pending edges*) that connect the component to the vertices of R.

A *contact vertex* of a part p of G relative to R is defined as a vertex of G - R that is incident with a pending edge of p. We will say a planar representation R of a subgraph S of a planar graph G is *planar extendible* to G if R can be extended to a planar representation of G. The extended representation is called a *planar extension* of R to G. We will say a part p of G relative to R is *drawable* in a face f of R if there is a planar extension of R to G where p lies in f. We have already observed the following necessary condition for drawability in a face.

> Let p be a part of G relative to R.
> Then, p is drawable in a face f of R only if
> every contact vertex of p lies in V(f).

The condition is certainly not sufficient for drawability, since it merely says that if the contact vertices of a part p are not all contained in the set of vertices of a face f, p cannot be drawn in f in any planar extension of R to G. For convenience, we shall say that a part p which satisfies this condition is *potentially drawable* in f, prescinding from the question of whether or not either p or G are planar.

Demoucron's algorithm works as follows. We repeatedly extend a planar representation R of a subgraph of a graph G until either R equals G or the procedure becomes blocked, in which case G is nonplanar. We start by taking R to be an arbitrary cycle from G, since a cycle is trivially planar extendible to G if G is planar. We then partition the remainder of G not yet included in R into parts of G relative to R and apply the drawability condition. For each part p of G relative to R, we identify those faces of R in which p is potentially drawable. We then extend R by selecting a part p, a face f where p is drawable, and a path q through p, and add q to R by drawing it through f in a planar manner. The process can become blocked (before G is completely represented) only if we find some part that does not satisfy the drawability condition for any face, in which case R is not planar extendible to G and G is nonplanar.

A high level procedural statement of Demoucron's algorithm follows. We rely on the previous discussion to make the meaning of the data types clear.

```
Function Draw (G, R)

(* Returns a planar representation of G in R, or fails *)

var  G: Graph
     R: Planar Representation
     p: Part
     P: Set of Part
     f: Face
     F: Set of Face
     Draw: Boolean function

Set Draw to True
Set R to some cycle in G

repeat

   Set P to the set of parts of R relative to G

   for each p in P  do  Set F(p) to the set of faces of R
                        in which p is drawable

   if         F(p) is empty for any p

   then       Set Draw to False

   else if    For some part p, F(p) contains only a single face

        then  Let f be the (unique) face in which p is drawable

        else  Let p be any part and let f be a face in F(p)

        Let q be a path between a pair of contact vertices
        of p with R that contains only edges of p

        Set R to R U q

until   R = G   or   not Draw
End_Function_Draw
```

The construction of a path q between a pair of contact vertices c_1 and c_2 on the boundary of a face f of a planar representation R is illustrated in Figure 1-30. An application of Demoucron's algorithm to Petersen's graph is shown in Figures 1-31 and 1-32. Initially, R consists of the cycle v_1-v_2-v_3-v_4-v_5-v_1. There is only one part of G relative to R, consisting of all the remaining vertices and edges of G, that is, the induced subgraph on vertices $v_6..v_{10}$ and the edges connecting this subgraph to the cycle R. The contact vertices of this part are $v_1..v_5$. Any path through this unique part which

connects one of its contact vertices to another can serve as the path q described in the procedure. We will select v_1-v_6-v_8-v_3 as q and update R to R U q. The new subgraph R has two interior faces and a single exterior face. There is only one part of G relative to R, as shown in Figure 1-32, and its contact vertices are $\{v_2,v_4,v_5,v_6,v_8\}$. Since none of the faces of R contains this set of contact vertices, the part is not drawable in any face; hence we conclude Petersen's graph is nonplanar.

Figures 1-30, 1-31, and 1-32 here

THEOREM (CORRECTNESS OF DEMOUCRON'S ALGORITHM) Let G(V,E) be a graph. Then, Demoucron's algorithm finds a planar embedding of G if G is planar or correctly recognizes G as nonplanar.

The proof of the theorem is as follows. We use an induction over the (implicit) loop index of the algorithm. We show that if G is planar, the representation R constructed by the algorithm is always planar extendible to a planar representation of G. The initial representation R is a cycle, and so is trivially planar extendible, provided G is planar in the first place. In general, we show that if the current representation R is planar extendible to G, its updated representation remains planar extendible. If R is planar extendible, every part of G relative to R must have some face of R in which it is drawable. We will distinguish two cases depending on whether or not some part is drawable in only a single face.

If a part p has a unique face f in which it is drawable, since R is planar extendible and there is only one possible choice of face in which to embed p with respect to R, then p must lie in f in every possible planar extension of R to G. Consequently, we may draw the path q through f and the resulting extension of R remains planar extendible.

Suppose, on the other hand, that for every part p of G relative to R there are at least two faces in which p is drawable. We will show that in this case the choice of face in which p is drawn is not critical. Let f be a face in which p is drawable. Then, we show there exists a planar extension of R to G for which p lies in f. Let RX denote an extension of R to G where p lies in a face f' not equal to f. Then, the contact vertices of p must lie on both the border of f and the border of f', and so must lie on the shared boundary B between f and f'. Consider all the parts of G relative to R whose contact vertices lie on B. Some of these parts lie in f in RX and some lie in f' in RX. Define a new planar representation RX' of G by flipping these parts about B. That is, flip those parts lying in f in RX so they lie in f' in RX', and flip those parts lying in f' in RX so they lie in f in RX'. Then, RX' is a planar representation of G for which p lies in f. Therefore, we can draw p in f and R remains planar extendible, as required.

Thus far, we have shown that if G is planar, the algorithm can extend a planar representation of a subgraph until all of G is drawn. On the other hand, of course, if G is nonplanar, the algorithm will recognize this at some stage by finding a part with no face in which it can be drawn, which completes the proof of the theorem.

PLANAR GRAPH THEORY

We will now prove some elementary but basic results of planar graph theory. The first theorem is simple and was one of the first contributions to the subject. Its name derives from the relation it identifies between the number of vertices, edges, and faces of a polyhedron (cube, tetrahedron, etc.). The edges of these solids define planar graphs when they are appropriately projected on the plane.

THEOREM (EULER POLYHEDRON FORMULA) Let G(V,E) be a connected planar graph and let |F| denote the number of faces in a planar embedding of G. Then,

$$|F| + |V| = |E| + 2.$$

See Figure 1-33 for an example.

Figure 1-33 here

The proof of the theorem is as follows. It is by induction on the number of edges of the graph. If the graph has only one edge, |V| = 2, |E| = 1, and |F| = 1; so the theorem is trivial. By induction, we assume the theorem is true for any graph with n edges, and then consider what happens if we add a further edge. (Refer to Figure 1-34.) If the additional edge leads to a new vertex, |V| and |E| will both increase by 1, while |F| will remain unchanged; so the balance of the equation will be maintained. On the other hand, if the new edge is between existing vertices, the edge must divide an existing face into two parts, thus increasing both |F| and |E| by 1, which again preserves the equation. This completes the proof.

Figure 1-34 here

THEOREM (LINEAR BOUND ON |E|) If G(V,E) is a planar graph, then

$$|E| \leq 3|V| - 6.$$

The proof is as follows. We let $|F_i|$ denote the number of edges bounding the i^{th} face of some planar representation of G. Each face must contain at least 3 bounding edges. Therefore,

$$\sum_{i=1}^{|F|} |F_i| \geq 3|F|.$$

Since each edge borders exactly two faces, the summation counts each edge twice. Therefore,

$$\sum_{i=1}^{|F|} |F_i| = 2|E|.$$

Combining these results and using the Euler Polyhedron Formula, we obtain

$$|E| \leq 3|V| - 6,$$

as was to be shown.

THEOREM (NONPLANARITY OF KURATOWSKI GRAPHS) The graphs K(3,3) and K(5) are nonplanar.

The proof of this theorem is as follows. The nonplanarity of K(5) follows by contradiction from the Linear Bound Theorem, since if K(5) were planar we would have $|E|$ $(= 10) \leq 3|V| - 6 = 9$, which is a contradiction. For K(3,3), we argue as follows. By Euler's Theorem, $|F| = |E| - |V| + 2 = 5$. However, since every cycle in K(3,3) has at least four edges, we can refine the proof of the Linear Bound Theorem in this case to prove that $|F| \leq 4$. That is, if K(3,3) were planar we would have

$$\sum_{i=1}^{|F|} |F_i| \geq 4|F|.$$

whence, emulating the Linear Bound proof, we could conclude that $|E|$ $(= 9) \geq 2|F|$, so that $|F| \leq 4$, contrary to the previous lower bound of 5 for $|F|$. Therefore, K(3,3) must be nonplanar. This completes the proof of the theorem.

The following upper bound on the minimum degree will prove useful when we consider an algorithm for five-coloring a planar graph in Chapter 7.

THEOREM (MINIMUM DEGREE BOUND) If G(V,E) is a planar graph, $\min(G) \leq 5$.

The proof is by contradiction. Suppose that the minimum degree of G is at least 6. Then,

$$\sum_{i=1}^{|V|} \deg(v_i) \geq 6|V|.$$

Since the sum of the degrees is always $2|E|$,

it follows that $|E|$ is at least $3|V|$, contrary to the Linear Bound Theorem. It follows that the minimum degree of G must be at most 5, which completes the proof.

EMBEDDING GRAPHS ON MANIFOLDS

The attempted embedding of K(3,3) in Figure 1-26 succeeds except for the single edge x. But, K(3,3) can be embedded on the surface of a *torus*, the closed two-dimensional manifold that corresponds to the surface of a doughnut.

We can represent a torus in a planar fashion by cutting it across its tubular cross-section, and then cutting the resulting cylinder to form a rectangle. We consider the opposite borders of the resulting rectangle as identified. Figure 1-35 gives a toroidal embedding of K(3,3). We think of the edge x in the figure, that strikes out vertically towards the upper border of the rectangle, as passing around the back of the torus, reappearing on the lower border after having circumnavigated the tubular dimension of the torus. The edge y that starts out to the right and reappears at the opposite point on the left border of the rectangle corresponds to a radial circumnavigation of the torus.

Figure 1-35 here

This example suggests two common generalizations of the concept of planar embedding: the genus of a graph and the crossing number of a graph.

The *genus of a* (closed two-dimensional) *surface* is the number of handles on the surface. Thus, we can consider a sphere as having genus zero, while a torus, which can be visualized as a sphere with one handle has genus one. The *genus of a graph* G is then defined as the genus of the surface of least genus on which G can be embedded, that is, drawn without spurious edge intersections. For example, since we can draw K(3,3) on the torus but not on the sphere (which is equivalent for present purposes to the plane), K(3,3) has genus one. K(5), K(6), and K(7) can also be embedded on the torus; so their genus' are also one. On the other hand, K(8) requires a two handled sphere for embedding and so has genus two.

The *crossing number* of a graph G(V,E) is defined as the minimum number of edge crossings among all possible embeddings of G in the plane. Obviously, the crossing number of a planar graph equals zero while, as we have just seen, the crossing number of K(3,3) equals one.

1-7 EULERIAN GRAPHS

An *euler trail* in a graph G(V,E) is a closed trail (or circuit) that spans all the edges of G. An *open euler trail* is an open trail that spans all the edges of G. An *eulerian graph* is a graph that contains an eulerian trail. A *semieulerian graph* is a graph that contains an open euler trail. Eulerian, semieulerian, and noneulerian graphs are shown in Figure 1-36.

Figure 1-36 here

For convenience, we will use the following notation (for this section only). Let T be a trail (open or closed) in a connected graph G(V,E). Then, we denote by G - T the graph obtained by removing E(T) from E(G) along with any vertices in V(G) isolated as a result of the removal of E(T) from G. Eulerian graphs have a simple characterization in terms of their degree sequences.

THEOREM (CHARACTERIZATION OF EULERIAN GRAPHS) Let G(V,E) be a connected graph. Then, G is eulerian if and only if the degree of every vertex in G is even.

The proof of the theorem is as follows. The necessity of the condition is obvious. If G is eulerian, the euler trail clearly induces a graph whose

vertices have even degree. We can prove sufficiency as follows. Since every vertex in G has even degree, G must contain some closed trail T. Since G is connected, every component of G - T must be incident with some vertex of T. The subgraphs induced by each of these components are connected and have degree sequences with even degrees; so we may assume by induction that each of these components is eulerian. Let T' be an euler trail in a component that intersects T at a vertex v. We can combine T and T' into one euler trail by starting T at any of its vertices, proceeding until we reach v, and then following the trail T' until we have traversed it completely; whence we continue with the traversal of T. If we repeat this process, we eventually obtain a closed spanning trail of G. This completes the proof of the theorem.

If a connected graph has only two vertices of odd degree, it is semieulerian. This is a special case of the following more general theorem.

THEOREM (EULER TRAIL DECOMPOSITION) Let G(V,E) be a connected graph with 2k vertices of odd degree. Then, the edges of G can be partitioned into k edge disjoint open trails $E_1,...,E_k$.

The proof of the theorem is as follows. Denote the 2k vertices of odd degree by $v_1,...,v_k$ and $w_1,...,w_k$. Add k new vertices $u_1,...,u_k$ to G together with 2k new edges (v_i,u_i) and (u_i,w_i), where $i = 1,...,k$. Denote the resulting graph by G'. By the previous theorem, G' is eulerian, and so has an euler trail. Since every new vertex u_i is of degree two in G', removing these vertices from the trail, breaks the euler trail on G' into k edge disjoint open tails, which together cover the edges of G. This completes the proof.

An *eulerian directed graph* is a digraph containing a directed euler trail. We say a digraph G is connected if the underlying undirected graph of G is connected. The following theorem is easily proved.

THEOREM (CHARACTERIZATION OF DIRECTED EULERIAN GRAPHS) Let G(V,E) be a connected digraph. Then, G is eulerian if and only if for every vertex v in V(G), indeg(v) equals outdeg(v).

EULER TRAIL ALGORITHMS

We will describe two algorithms for finding euler trails in graphs. The first algorithm, which is due to Fleury, has an intuitive appeal; while the second algorithm, which is motivated by the proof of the Eulerian Characterization Theorem, is more efficient.

Fleury's algorithm is a visually convenient way of constructing an euler trail in an eulerian graph but suffers from $O(|E|^2)$ performance. The idea of Fleury's algorithm is to successively trace out the edges of a trail, erasing traversed edges and any resulting isolated vertices as we proceed, never traversing an edge if in doing so we would disconnect the remaining graph into nontrivial components or isolate the starting vertex before all the edges are traversed. Thus, Fleury's algorithm repeatedly extends an incomplete

eulerian trail T from its terminal vertex v by appending to T any edge x incident with v which is not a bridge of G - T and whose removal would not isolate the starting vertex before all the edges are traversed. Since it takes O(|E|) time to test whether an edge is a bridge, each edge only has to be tested once, and there are |E| edges in G, Fleury's algorithm has performance $O(|E|^2)$.

Refer to Figure 1-37 for an illustration of the pitfalls avoided by Fleury's algorithm. If we start a trail at the vertex c traversing it until we obtain T = c-b-a, at that point the edge (a,d) which is incident with the endpoint a of T is a bridge of G - T. If we follow the edge selection principle in Fleury's algorithm and continue the trail via either of the edges (a,e) or (a,g), we will be able to successfully complete an euler trail. On the other hand, if we extend T via the edge (a,d), which is contrary to the bridge avoidance criterion in Fleury, the resulting trail becomes trapped when it reaches c, before it has traversed all of E(G). The following theorem is easily proved.

Figure 1-37 here

THEOREM (FLEURY'S ALGORITHM)Let G(V,E) be an eulerian graph.Then, Fleury's algorithm finds an euler trail in G in $O(|E|^2)$ time.

We will now describe an O(|E|) algorithm for euler trails based on the proof of the eulerian characterization theorem. We start with a trail T consisting of a single vertex and extend it until it becomes blocked. We then select a previously reached but incompletely traversed vertex from which we initiate a new closed trail T'. We then subsequently patch T' onto T, forming a new and more extensive trail. The process continues until all of E(G) is traversed.

If we denote the list of vertices that have been reached by the partial euler trail T but that still have untraversed incident edges by PL (meaning the Pending list), the vertices on PL correspond to those vertices in the characterization proof at which additional closed trails were patched onto an existing trail to produce a more extensive closed trail. The pending list becomes exhausted precisely when we have completed an euler trail on the component of G in which the trail was initiated, which in the case that G is connected is an euler trail on G.

We will now describe the procedure for constructing the euler trail more formally. We first consider the design of the requisite data structures. We will represent the graph G(V,E) as a linear array with doubly linked adjacency lists and with shared representatives for its edges to facilitate the edge deletions that will be required by the algorithm. We include a field, denoted PL-Pointer(v), at each vertex v, which points to the entry for v, if any, on the pending list PL. Another direct access pointer, denoted by Occurrence(v), points to an entry for v on T, if there is one. Both T and PL are represented by doubly linked lists. This organization, in conjunction with Occurrence, allows us to efficiently patch additional closed trails onto T. Similarly, in conjunction with the PL-Pointer field, it allows us to easily delete elements from PL. The corresponding type definitions are as follows.

```
type  Graph   =  record
                   H(|V(N)|): Vertex
                 end

      Vertex  =  record
                   Positional-Pointer, Successor: Edge pointer
                   PL-Pointer, Occurrence: Entry pointer
                 end

      Edge    =  record
                   Shared-rep: Shared-edge pointer
                   Edge-successor,
                        Edge-predecessor: Edge pointer
                 end



      Shared-edge  =
                 record
                   E(1,2): 1..|V|
                   Endpoints(2): Edge pointer
                 end

type  List  =    record
                   Head, Tail: Entry pointer
                 end

      Entry  =   record
                   Index: 1..|V|
                   Predecessor, Successor: Entry pointer
                 end
```

The procedure for constructing the euler trail is named Euler. Euler uses several subprocedures, which we will now describe. Create(T) initializes a doubly linked list T. Put(u,T) adds a vertex u to the end of T; while Put(u,PL,G) adds u to the pending list PL (it has a null effect if u is already on the list) and sets the corresponding PL-Pointer in G to point to that entry. Get(PL,w) returns an element w from the pending list PL, without deleting it, and fails if there is none. The function Next(G,w,v) returns in v the next neighbor of w or fails if there is none. Delete_edge(G,w,v) deletes the edge (w,v) from G. Empty(G,u) succeeds if the adjacency list at u is empty; Single(G,u) succeeds if the adjacency list at u has at most one more element; while Empty(G) succeeds if G is empty. Delete(w,G,PL) deletes the entry for w on PL using PL-Pointer(w) in G (and has a null effect if w is not on the list).

The procedure Patch(T,T',w) patches the closed trail T' that begins and ends at w onto the trail T by replacing one occurrence of w on T by the trail T'. Patch has a simple string substitution interpretation.

(1) If the trail T is x_1-x_2-...-a-(w)-b-...-x_n,

(2) and the substitute trail T'is (w-w_1-...-w_k-w),

(3) the new trail T U T' after substitution is

$$x_1\text{-}x_2\text{-}\ldots\text{-}a\text{-}(w\text{-}w_1\text{-}\ldots\text{-}w_k\text{-}w)\text{-}b\text{-}\ldots\text{-}x_n, \text{ or}$$
$$x_1\text{-}x_2\text{-}\ldots\text{-}a\text{-}w\text{-}w_1\text{-}\ldots\text{-}w_k\text{-}w\text{-}b\text{-}\ldots\text{-}x_n.$$

The doubly linked representations for T and T' together with the Occurrence field in G which points to an occurrence of w in T allow an O(1) realization of Patch. The procedure for Euler follows.

```
Function Euler (G, u, T)

(* Returns euler trail for G starting at u in T, or fails *)

var G: Graph
    u,w,v: 1..|V|
    T, T', PL: List
    Next, Get, Empty, Single, Euler: Boolean function

Create(T); Put(u,T); Create(PL); Put(u,PL,G)

while  Get (PL,w)  do

   Create(T'); Put(w,T')

   while  Next(G,w,v)  do  Put (v,T')
                           Delete_edge (G,w,v)
                           if Empty (G,w) then Delete (w,G,PL)
                           if Single(G,v) then Delete (v,G,PL)
                                          else Put (v,PL,G)
                           Set w to v

   Patch (T,T',w)

Set Euler to Empty (G)
End_Function_Euler
```

1-8 HAMILTONIAN GRAPHS

We have already defined hamiltonian graphs as graphs that contain spanning cycles. There is an extensive and profound theory of hamiltonian graphs, which is currently largely of theoretical interest. We will consider this topic here partly because, like Mount Everest, "It is There," but also because the problem of recognizing such graphs plays a paradigmatic role in algorithmic complexity theory. Indeed, the recognition of these graphs is a classic instance of a problem for which straightforward search algorithms are available but for which no efficient algorithm seems possible.

We call a spanning cycle in a hamiltonian graph a *hamiltonian cycle*, while a spanning path in a graph is called a *hamiltonian path*. Analogous terms are defined for digraphs. The problem of determining whether a graph has a

hamiltonian cycle is called the *hamiltonian cycle problem*. The related *traveling salesman problem* seeks to find the shortest spanning circuit in a weighted graph, which, when the edge weights satisfy the triangle inequality, can be shown to be a hamiltonian cycle. The hamiltonian cycle and traveling salesman problems are both NP-Complete, that is, as we shall see in Chapter 10, they are probably unsolvable in polynomial time. We will describe a backtracking algorithm for finding hamiltonian cycles, a search technique often appropriate for nonpolynomial problems. We will also describe an approximation algorithm for the euclidean version of the traveling salesman problem.

MODEL: OPTIMAL SCHEDULES

The following problem can be solved by finding a shortest hamiltonian path in a weighted digraph. Let $\{P_i, i = 1,...,n\}$ be a set of n processes, all of which need a resource which they access sequentially. An ordered list of the processes constitutes a process schedule. If a process P_i is scheduled just prior to process P_j, a reset cost c_{ij} is incurred in preparing the resource to run P_j. The cost of a schedule is defined as the total of the reset costs summed over all the whole schedule. We can model the scheduling of the set of processes and the reset costs as a weighted digraph G, where the processes correspond to the vertices of the digraph and the weight of an edge (i,j) equals the reset cost c_{ij}. The cost of a process schedule then corresponds to the total cost of the edges between successive processes of the schedule. If we define an optimal schedule as a schedule of minimum cost, an optimal schedule corresponds to a shortest hamiltonian path in G.

BASIC CONCEPTS

In general, it is difficult to determine if a graph has a hamiltonian cycle or hamiltonian path, but there are some simple necessary and/or sufficient conditions for a graph to be hamiltonian. The following theorem of Ore is well-known.

THEOREM (ORE'S PATH AND CYCLE LENGTH CONDITION) Let G(V,E) be a graph of order $|V| \geq 3$, and suppose that for every pair of nonadjacent vertices u and v in G

$$\deg(u) + \deg(v) \geq M,$$

for an integer M. If M equals |V|, G is hamiltonian, while if G is connected, it contains a path of length M.

We shall prove the theorem in the special case that M equals |V| and leave the generalization to the complete theorem to the reader. The proof is by contradiction. Suppose the theorem does not hold for some order |V|. Let G be a nonhamiltonian graph of order |V| = p which satisfies the conditions of the theorem and has maximum size among all such graphs. Let u and v be nonadjacent vertices in G. Then, G U (u,v) must be hamiltonian, and so there must be a hamiltonian path P in G from u to v. Let the successive vertices of P be $u = u_1, u_2 ,..., u_p = v$. If u is adjacent to u_i, then v cannot be adjacent to ui-1. Otherwise, the sequence $u_1, u_2 ,..., ui-1, v, up-1 ,...,$

u_i, u_1 would determine a hamiltonian cycle in G, contrary to assumption. Therefore, for every vertex in G that u is adjacent to, there is a vertex in G that v is not adjacent to. That is, deg(v) $\leq$ |V| - 1 - deg(u), contrary to the condition of the theorem. It follows that G must be hamiltonian, which completes the proof.

We refer to Figure 1-38 for an example illustrating the sharpness of the theorem. The conclusions of the theorem can be strengthened, for when M = |V| the graph can be shown to have cycles of every order, unless |V| is even and G is isomorphic to K(|V|/2,|V|/2).

Figure 1-38 here

The Theorem of Ore provides a sufficient condition for the existence of a hamiltonian cycle. The following theorem gives a necessary and sufficient condition. First, we define the *closure* of a graph G(V,E) of order |V| as the graph obtained from G by recursively inserting an edge between every pair of nonadjacent vertices of degree sum at least |V| (or which becomes so recursively as the result of adding edges).

THEOREM (CLOSURE CONDITION FOR HAMILTONICITY) A graph G(V,E) is hamiltonian if and only if the closure of G is hamiltonian.

We refer to Figure 1-39 for an example of the closure operation. The final graph is K(6), which is hamiltonian; so the original graph is hamiltonian.

Figure 1-39 here

There is a simple condition for a digraph to have a hamiltonian path.

THEOREM (REDEI'S CONDITION) If G(V,E) is a digraph whose underlying graph is complete, G has a directed hamiltonian path.

The proof is by an induction on the order of G. Assume the theorem is true for digraphs with order at most p, and consider a digraph G of order p + 1. Let u be any vertex in G. Then, by induction, the digraph G - u has a hamiltonian path P: u_1 ,..., u_p. By assumption, either (u, u_1) or (u_1, u) is in G. If (u, u_1) is in G, then u, u_1 ,..., u_p determines a hamiltonian path in G, as required. If (u_1, u) is in G, let u_i be the first vertex on P for which (u,u_i) is in G, if any. If u_i exists, then u_1, u_2 ,..., ui-1, u, u_i, ui+1 ,..., u_p determines a hamiltonian path in G. Otherwise, u_1, u_2 ,..., u_p, u determines a hamiltonian path in G. This completes the proof.

There are interesting relations between hamiltonicity, connectivity, planarity, and the powers of a graph. We call a graph G(V,E) *hamiltonian connected* if there is a hamiltonian path between every pair of vertices in G. A hamiltonian graph is necessarily hamiltonian connected, though not vice versa. We have the following theorem.

THEOREM (HAMILTONIAN POWERS) If G(V,E) is one connected, the cube of G is hamiltonian connected. If G(V,E) is two connected, the square of G is

hamiltonian connected.

The square of the one connected graph in Figure 1-40 is nonhamiltonian, proving the sharpness of the first condition in the theorem. Contrary to the progression suggested by the theorem, it is not the case that if G is three connected, G must be hamiltonian. Indeed, Figure 1-41 shows Tutte's famous counterexample to the longstanding conjecture that even three connected cubic planar graphs were necessarily hamiltonian.

Figure 1-40 and 1-41 here

THEOREM (TUTTE'S CONDITION) If G(V,E) is four connected and planar, G is hamiltonian.

BACKTRACKING ALGORITHM

Backtracking is a general technique for generating solutions to a combinatorial problem by systematically extending partial solutions to the problem. The technique is described in greater detail in Chapters 2 and 8. We will use it here to find hamiltonian cycles in a graph G. We will assume G is represented by the type adjacency matrix (with matrix ADJ), described in Section 1-2.

The algorithm is stated in Find_Hamiltonian_Path(G). This finds all the hamiltonian paths in a graph G(V,E) by repeatedly extending partial hamiltonian paths. The successive vertices of the path are stored in an array Path. Find_Hamiltonian_Path provides the driver logic for the backtracking algorithm, while the procedure Next(Path,k) does the work of getting the next candidate vertex for extending the path. Next returns, in Path(k), the next vertex which is adjacent to the current endpoint of the path, Path(k - 1), and which has an index higher than the current value of Path(k). If there is no such vertex, Next returns the dummy index |V(G)| + 1. The new path is Path(1) ,..., Path(k - 1) 3 Path(k), where 3 denotes path concatenation. We use an array Status to indicate if a vertex is on the current path. Status(i) is 1 if vertex i is on the path, and 0 otherwise. To simplify the implementation of Next, we use the conditional operator or*, which leaves a later operand unevaluated if an earlier one is true.

```
Procedure Find_Hamiltonian_Paths (G)

(* Finds all hamiltonian Paths in G *)

var G: Graph
    k: 1..|V|
    Path(1..|V|): 0..|V| + 1
    Status(1..|V| + 1): 0..1
    No_More_Paths: Boolean

Set Path (1..|V|) to 0
Set Status (1..|V| + 1) to 0
Set k to 1
```

```
Set No_More_Paths to False

repeat

   Next(Path, k)

   case

   1: k < |V| and Path(k) ≤ |V| :  Set k to k + 1

   2: k = |V| and Path(k) ≤ |V| :  Display Path(1..|V|)
                                   Set Status(Path(|V|)) to 0

   3: k > 1   and Path(k) > |V| :  Set Path(k) to 0
                                   Set Status(Path(k - 1)) to 0
                                   Set k to k - 1

   4: k = 1   and Path(k) > |V| :  Set No-More_Paths to True

until  No-More_Paths

End_Procedure_Find_Hamiltonian_Paths



Procedure Next(Path,k)

(* Returns in Path(k) the next vertex adjacent to Path(k - 1)
   not currently on the path *)

var  G: Graph
     k: 1..|V|
     Path(1..|V|): 0..|V| + 1
     Status(1..|V| + 1): 0..1

repeat

  repeat  Increment Path(k)  until  Status(Path(k)) = 0

until  k = 1  or*  Path(k) > |V|  or*  Adj(Path(k),Path(k-1)) = 1

if Path(k) ≤ |V|   then   Set Status(Path(k)) to 1

End_Procedure_Next
```

We refer to Figures 1-42 and 1-43 for an example.

Figures 1-42 and 1-43 here

We can adapt the same procedure with minor variations to find a variety of other combinatorial objects such as

(1) Hamiltonian cycles,
(2) A longest path starting at a given vertex,
(3) A longest path between a given pair of vertices,
(4) A longest path between a given pair of vertices and which avoids a given set of vertices, and
(5) Paths of length at least k starting at a given vertex.

APPROXIMATE ALGORITHM

Let G(V,E) be a complete graph with positive weights assigned to each edge. We denote the weight assigned to edge (i,j) by c(i,j). We will say G is a *euclidean graph* if the edge weights satisfy the *triangle inequality*

$$c(i,j) + c(j,k) \geq c(i,k),$$

for every three vertices i, j, and k. For euclidean graphs, optimal circuits are optimal cycles.

THEOREM (EUCLIDEAN HAMILTONIAN CYCLES) If G(V,E) is a euclidean graph, a hamiltonian cycle on G of least cost is also a solution to the traveling salesman problem on G.

The proof of the theorem is as follows. We will show that a least cost circuit can always be transformed into a hamiltonian cycle of equal cost. Thus, suppose C is a least cost circuit in G which is not a cycle. Then, C must contain at least one multiply covered vertex u and so can be represented as a sequence

$$v_1\text{- ... -}u\text{- ... -}v_i\text{-}u\text{-}v_j\text{- ... -}v_k\text{-}v_1.$$

If we replace the edges (v_i,u) and (u,v_j) by the single edge (v_i,v_j), then u still lies on the modified cycle since u had already been spanned before this occurrence of u. It follows from the triangle inequality that

$$c(v_i,v_j) \leq c(v_i,u) + c(u,v_j).$$

Therefore, replacement does not increase the cost of the circuit and the new circuit remains closed and spanning. If we repeat this procedure for every such vertex u that appears at multiple points on the circuit, we must eventually obtain a simple, non-self-intersecting circuit, that is, a cycle. By construction, the cycle is spanning and of minimum cost. This completes the proof of the theorem.

Although it is computationally difficult to obtain an exact solution to the traveling salesman problem, we can easily obtain a good approximation to an optimal solution. Indeed, if the graph is euclidean, by the preceding theorem we can even convert an approximately optimal circuit to an approximately optimal hamiltonian cycle. We will assume that G is euclidean.

The procedure for an approximate solution to the traveling salesman problem on G is as follows:

(1) Construct a minimum weight spanning tree *mst* on G,
(2) Convert *mst* to an eulerian graph *eul* by doubling its edges,
(3) Construct an euler trail *et* on *eul*, and
(4) Apply the euclidean transformation to convert *eul* to a spanning cycle *cyc*.

The minimum weight spanning tree referred to in (1) can be efficiently constructed using techniques described in Chapter 4. The doubling of the edges in (2) transforms the graph into what is strictly speaking a multi-graph (where parallel edges are allowed). Nonetheless, the standard results for eulerian graphs developed in the preceding section still apply, whence *eul* is eulerian and so contains an euler trail *et*. Then, in (4), we apply the circuit-to-cycle transformation described previously. We refer to Figure 1-44 for an example of the procedure. In the example, the approximately optimal solution C_3 is actually optimal. It is easy to establish the following bound on the error of the approximation.

Figure 1-44 here

THEOREM (TSP ERROR BOUND) Let G(V,E) be a euclidean graph. Let S denote an optimal solution to the traveling salesman problem on G and let cost(S) denote its weight. Let *cyc* denote the spanning cycle constructed by the approximation procedure we have described and let cost(*cyc*) denote its weight. Then,

$$\text{cost}(cyc) \leq 2\ \text{cost}(S).$$

The proof is simple. The total weight of the spanning tree *mst* is certainly less than or equal to the weight of an optimal circuit S since any spanning circuit contains a spanning tree. Therefore, the weight of *et* is at most twice the weight of *mst*. It follows that the cost of the spanning cycle *cyc* is at most twice the cost of an optimal solution.

CHAPTER 1: REFERENCES AND FURTHER READING

Behzad, et al. (1979) is an excellent, detailed introduction to pure graph theory. Wilson (1985) is a delightful briefer introduction to the subject. Harary (1971) is classical and comprehensive. See Capobianco and Molluzzo (1978) for a comprehensive collection of examples and counterexamples to conjectures in graph theory. Roberts (1984) has many interesting examples of the applications of combinatorics in general, and graphs in particular, as does the older Busacker and Saaty (1965).

Gibbons (1985) is a very readable introduction to graph algorithms. Reingold, Nievergelt and Deo (1977), Christofides (1975), and Papadimitriou and Steiglitz (1982) cover combinatorial and graph algorithms, as do Gondran and Minoux (1984) and Swamy and Thulasiraman (1981). Tarjan (1983) has extensive discussions of efficient data structures for graph algorithms. For data structures, see Standish (1980) and all the volumes of Knuth. Mc Hugh (1986) gives an overview of data structures. Welsh (1983) reviews random combinatorial algorithms and refers to Lovasz (1979). See also, Lovasz and Plummer (1987) and the references given for matching in Chapter 8. See Bondy and Murty (1976) for the algorithm of Demoucron, et al., as well as Demoucron,

Malgrange, and Pertuiset (1964); also see Gibbons (1985) for a discussion. The procedure for drawing a planar graph with straight line segments described in the exercises is from Tutte (1963). The linear time planarity algorithm was given in Hopcroft and Tarjan (1974).

CHAPTER 1: EXERCISES

(1) Let G(V,E) be a graph. Prove either G or G^c is connected.

(2) What is the largest number of edges in a disconnected graph G of order |V|?

(3) Show a graph of order |V| with more than $|V|^2 / 4$ edges cannot be bipartite.

(4) Construct two nonisomorphic graphs with the same reachability matrix.

(5) What are the nonisomorphic graphs of order 5?

(6) Show G is bipartite if and only if its adjacency matrix can be rearranged and partitioned into submatrices A_{ij} (i, j = 1,2), where A_{11} and A_{22} are 0, and A_{12} and A_{21} are transposes. Show G is disconnected if and only if its adjacency matrix can be rearranged and partitioned into submatrices so that A_{12} and A_{21} are 0. What partitions are feasible if G has a cut-vertex? a bridge?

(7) Define a *perfect graph* as a graph all of whose degrees are distinct. Prove there is no nontrivial perfect graph.

(8) Design algorithms for converting from one graphical representation to another, for adjacency matrix, edge list, and adjacency list.

(9) Design an algorithm for computing G - v, G - (u,v), G U {v}, and G U (u,v) with respect to each representation.

(10) Write an algorithm that generates random graphs of given edge density, that is, with a given value of 2|E|/(|V|(|V| - 1)).

(11) Write an algorithm that generates random connected r-regular graphs.

(12) Derive the $O(|V|^4)$ estimate for the time required to calculate the reachability matrix.

(13) Show there are only five regular planar graphs in which each face has the same number of bounding edges.

(14) What can you say about a cubic graph that satisfies |E| = 2|V| - 3?

(15) Can a vertex be a cut-vertex in both G and G^c?

(16) Let G(V,E) be a graph with weights on its edges. Prove that if M_1 has maximum edge weight among all matchings with k edges, and if p is an

augmenting path with respect to M_1 of maximum edge weight, the matching M_2 obtained by augmenting M_1 using p has maximum edge weight among all matchings with k + 1 edges.

(17) Prove the given alternating path algorithm does not work if the graph has odd cycles by giving a counterexample.

(18) Prove that if M_1 and M_2 are edge disjoint matchings in a graph G(V,E), there are matchings M_1' and M_2' such that $3M_1'3 = 3M_13 - 1$, $3M_2'3 = 3M_23 + 1$, and $M_1 \cup M_2 = M_1' \cup M_2'$.

(19) Determine the performance of Demoucron's algorithm, assuming some suitable data structures and using the known bounds on the size of planar graphs.

(20) Use Demoucron's algorithm to establish the nonplanarity of K(3,3) and K(5).

(21) Prove that for $|V| \geq 9$, G or G^c is non-planar; while for $|V| < 8$, G or G^c is planar. The problem is difficult when $|V| = 9$ or 10.

(22) Design an $O(|V|^6)$ algorithm to determine planarity using Kuratowski's theorem.

(23) Embed K(5) on the Mobius strip.

(24) Construct a planar graph G with $\min(G) \geq 5$.

(25) Show that every planar graph of order at least 4 has at least four vertices of degree at most 5.

(26) Prove there is no planar map with five regions in which every pair of regions is adjacent.

(27) If G(V,E) is planar and girth(G) equals $k \geq 3$,then $|E| \leq k(|V| - 2)/(k - 2)$. Use this result to show that Petersen's graph is nonplanar.

(28) Let G(V,E) be a nonseparable planar graph. Prove that G is bipartite if and only if the dual of G is eulerian.

(29) Represent K(10) as the union of three planar graphs.

(30) Carefully check that the eulerian trail algorithm is correct. Design an alternative recursive implementation of the algorithm based directly on the inductive proof of the Eulerian Characterization Theorem.

(31) Prove the Hamiltonian Closure Theorem.

(32) Prove the remaining part of Ore's Theorem.

(33) Modify the backtracking algorithm for hamiltonian paths so that it

finds a hamiltonian path starting at a given vertex i and fails if there is none. The algorithm should return the next hamiltonian path, with respect to some ordering of the paths, each time it is called.

(34) Adapt the hamiltonian path algorithm so it allows one vertex to be covered twice.

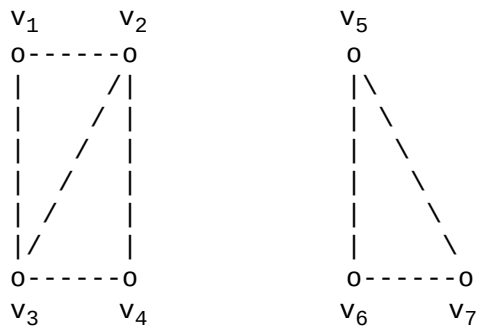
(35) Prove K(5,3) has neither a hamiltonian cycle nor path.

(36) Show Petersen's graph P(V,E) is not hamiltonian, but P - v is hamiltonian for every vertex v in V(P).

(37) Suppose V is a set of people with $|V| \geq 4$. Prove that if for every subset X of 4 people in V, there is someone in X who knows everyone in X, someone in V knows everyone in V.

(38) Implement the following procedure of Tutte (1963) for drawing any planar three-connected graph (with straight lines!). Let C be a cycle which bounds a region in some planar drawing of G. Place the vertices of C in order at the vertices of a regular polygon. Then, place each of the other vertices so that it is at the centroid of its adjacent vertices. Finally, connect adjacent vertices by straight lines.

SECTION 1-1



V(G) = $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

E(G) = { (v_1,v_2), (v_2,v_4), (v_2,v_3), (v_4,v_3),

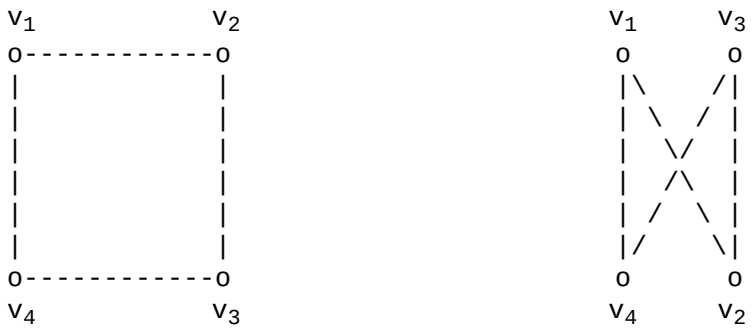
(v_3,v_1), (v_5,v_6), (v_6,v_7), (v_7,v_5) }

Order (|V(G)|) = 7

Size (|E(G)|) = 8

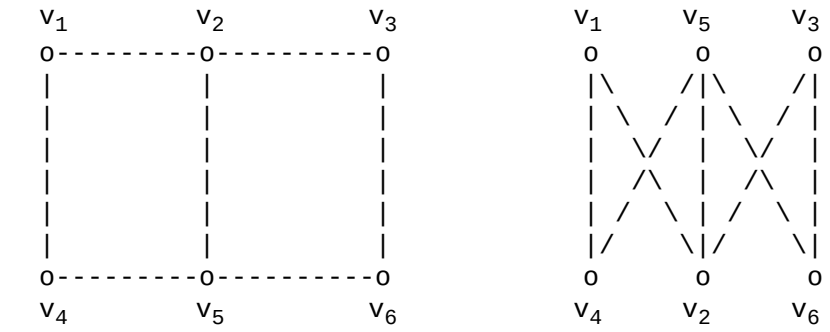
Number of components = 2

Figure 1-1. Example Graph G(V,E).



$V_1 = \{v_1, v_3\}$ $V_2 = \{v_2, v_4\}$

(a) Cyclic and Bipartite Presentations of C(4).



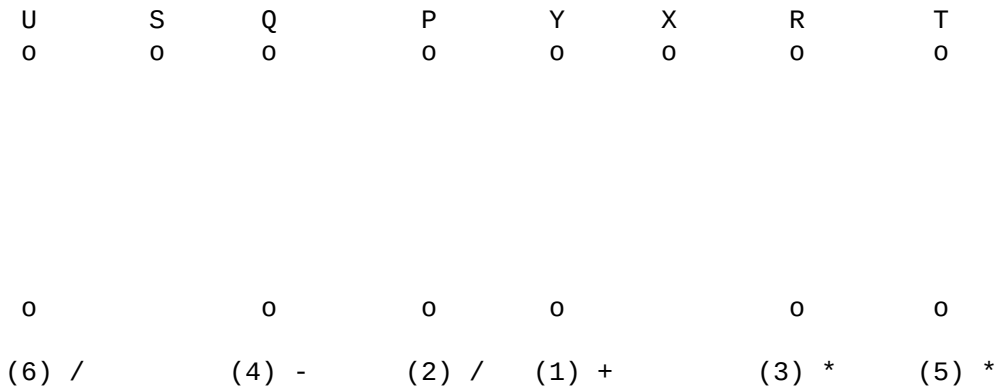
$V_1 = \{v_1, v_3, v_5\}$ $V_2 = \{v_2, v_4, v_6\}$

(b) Different Presentations of a Bipartite Graph $G(V_1, V_2, E)$.

Figure 1-2. Bipartite and Nonbipartite Graphs.

```
(1)  P = X + Y

(2)  Q = Y div P

(3)  R = X * P

(4)  S = R - Q

(5)  T = R * P

(6)  U = T div S
```

(a) Code Sequence.



(b) Data Flow Diagram for (a).

Figure 1-3. Bipartite Data Flow Model.

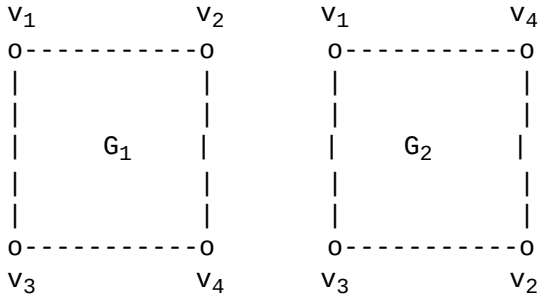
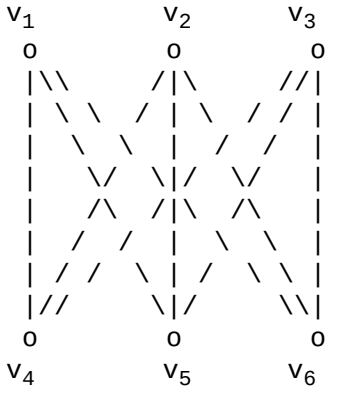
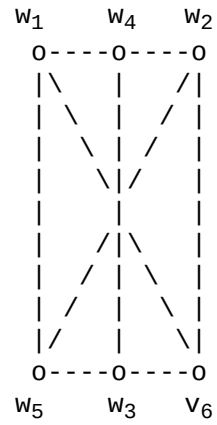


Figure 1-4. Isomorphic Graphs.



(a) K(3,3).



(b) G.

Isomorphic Mapping from K(3,3) to G:

v_1 ---> w_1

v_2 ---> w_2

v_3 ---> w_3

v_4 ---> w_4

v_5 ---> w_5

v_6 ---> w_6

Figure 1-5. A Pair of Isomorphic Graphs

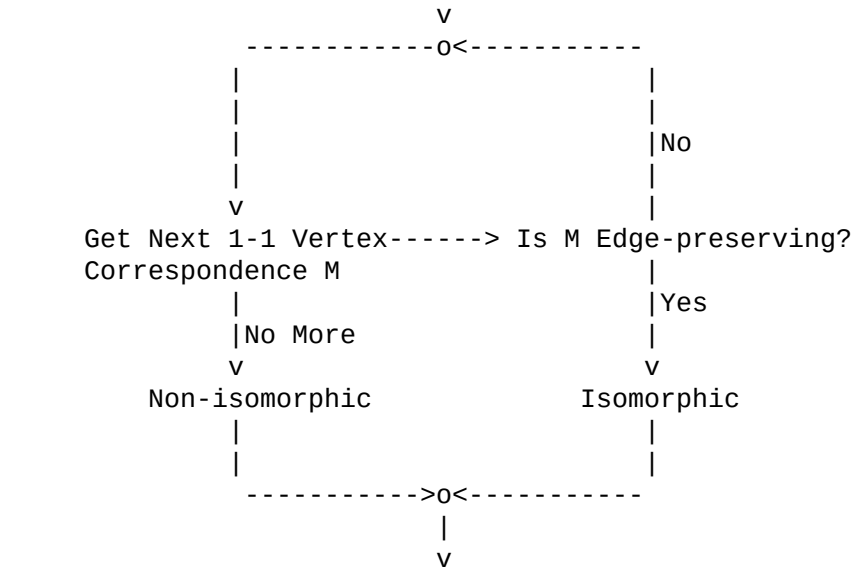
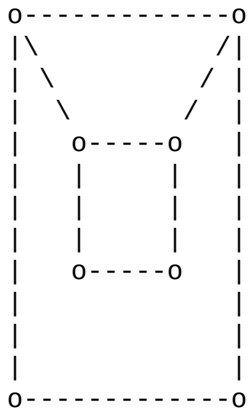
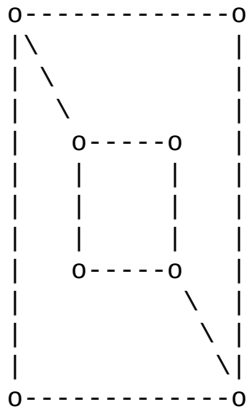


Figure 1-6. Exhaustive Search Algorithm for Isomorphism.

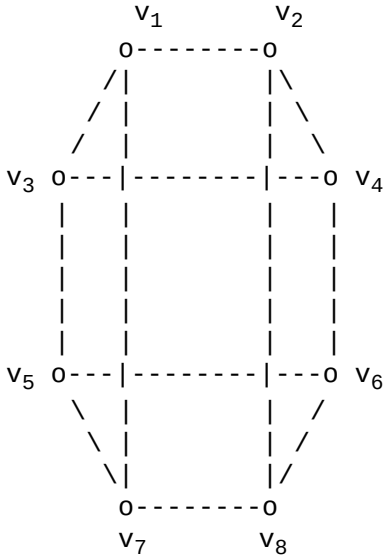


(a) G_1.

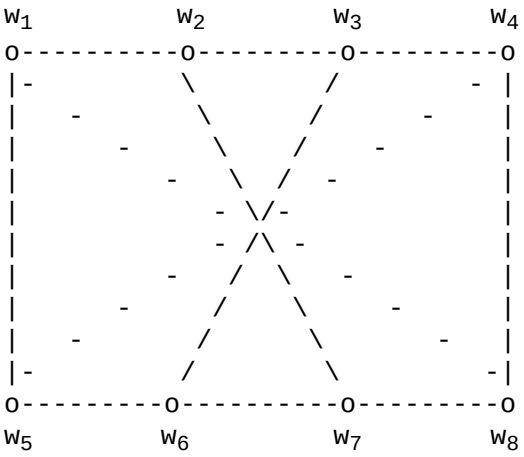


(b) G_2.

Figure 1-7. G_1 isomorphic to G_2?



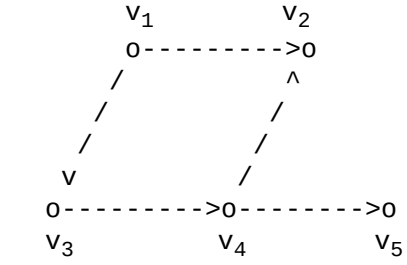
(a) G_1.



(b) G_2.

Figure 1-8. G_1 Isomorphic to G_2?

SECTION 1-2



(a) Digraph G.

ADJ(v_1): v_2-->v_3

ADJ(v_2): Nil

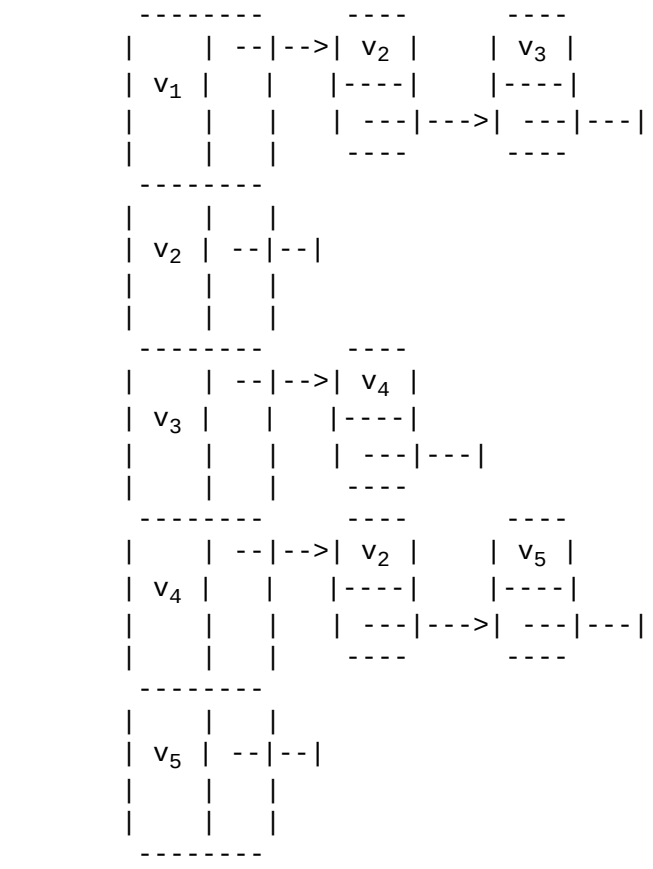
ADJ(v_3): v_4

ADJ(v_4): v_2-->v_5

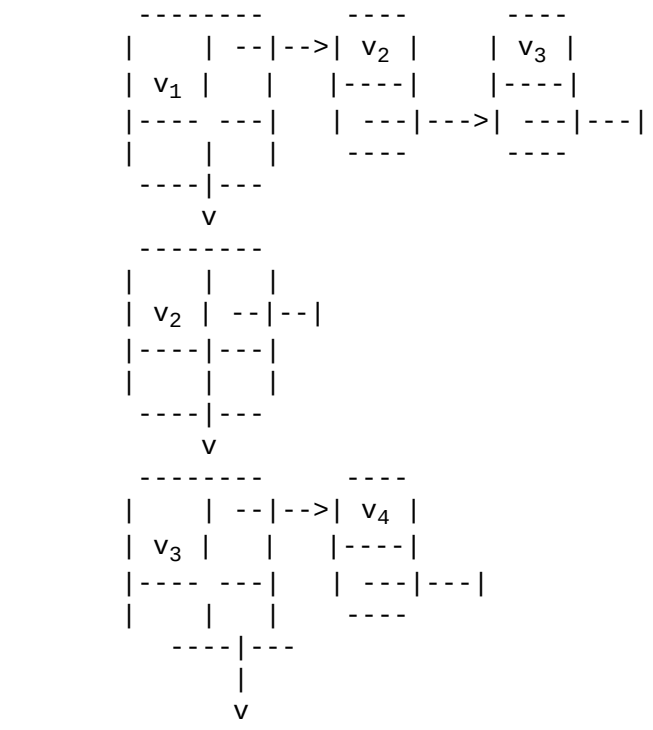
ADJ(v_5): Nil

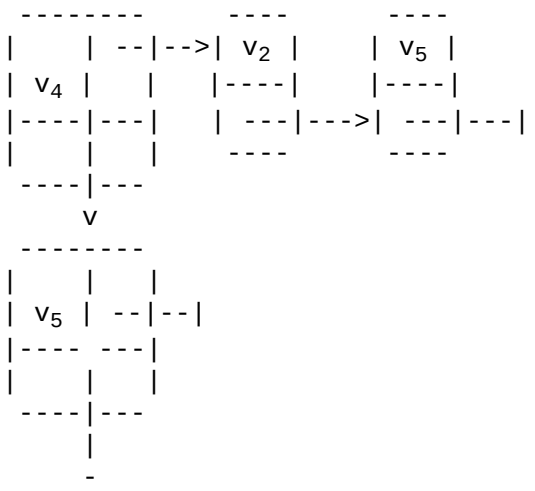
(b) Adjacency Lists for G.

Figure 1-9. Adjacency List Representation for a Digraph.

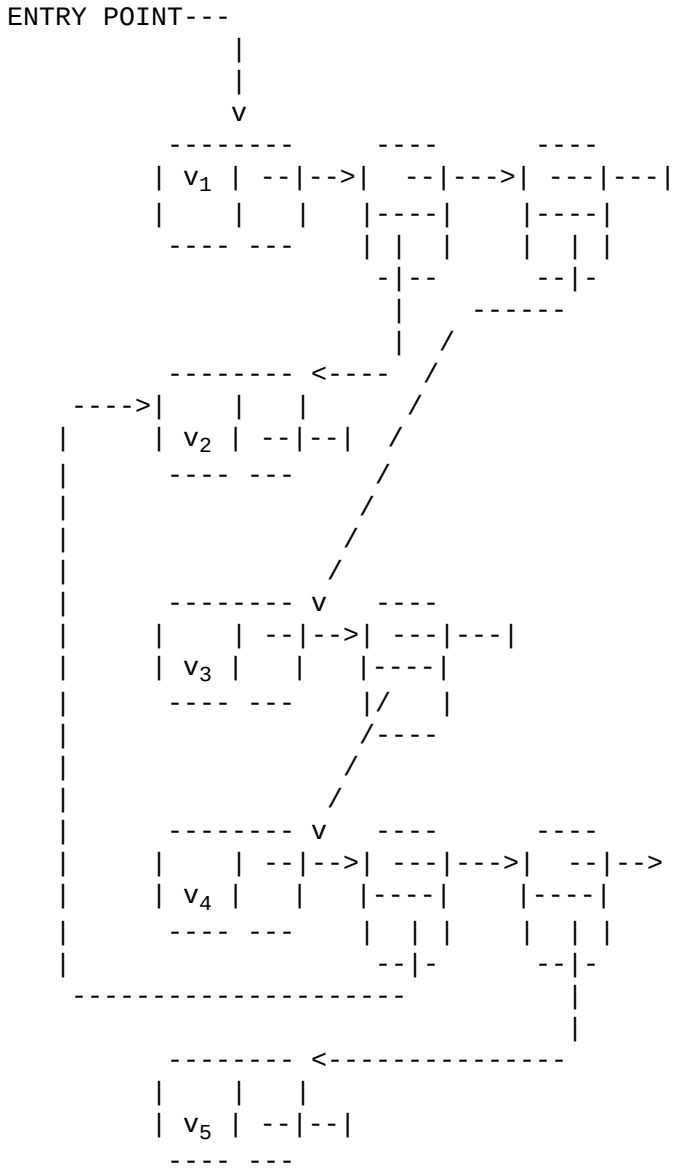


(a) Linear Array Representation for G.





(b) Linked Linear List Representation for G.



(c) Pure Linked Representation for G.

Figure 1-10. Different Vertex Organizations for Adjacency Lists.

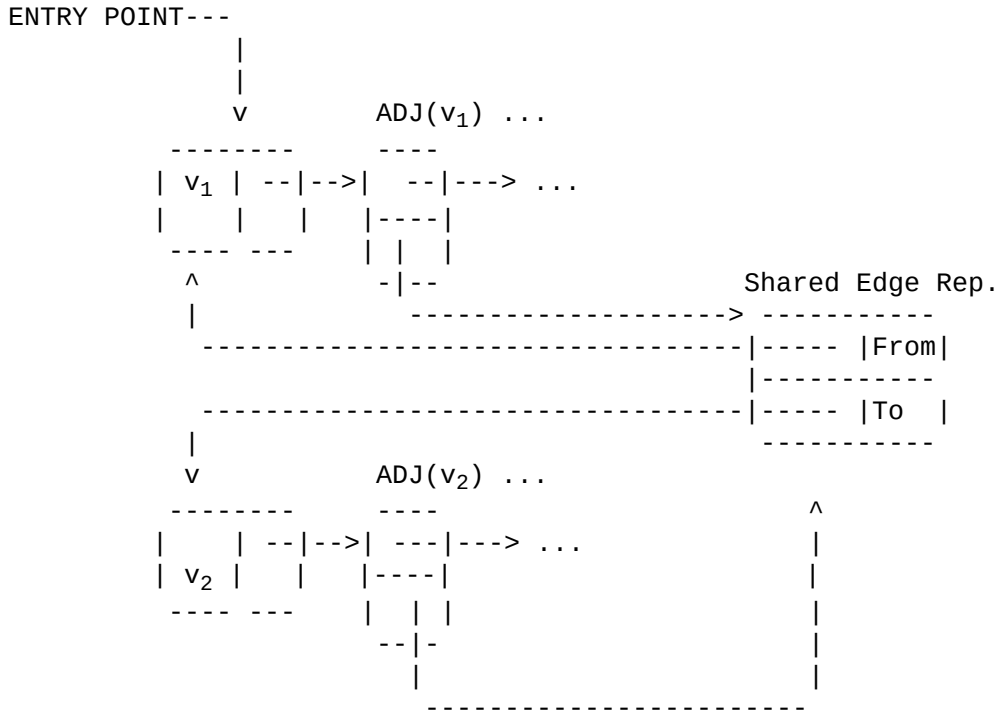
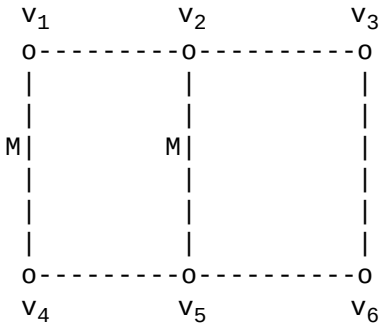


Figure 1-11. Shared Representation for Pure Linked Organization.

SECTION 1-3



Alternating Path: v_4-v_1-v_2-v_5-v_6

Alternating Path: v_4-v_1-v_2-v_5

Non-Alternating Path: v_4-v_1-v_2-v_3

Figure 1-12. Alternating Paths.

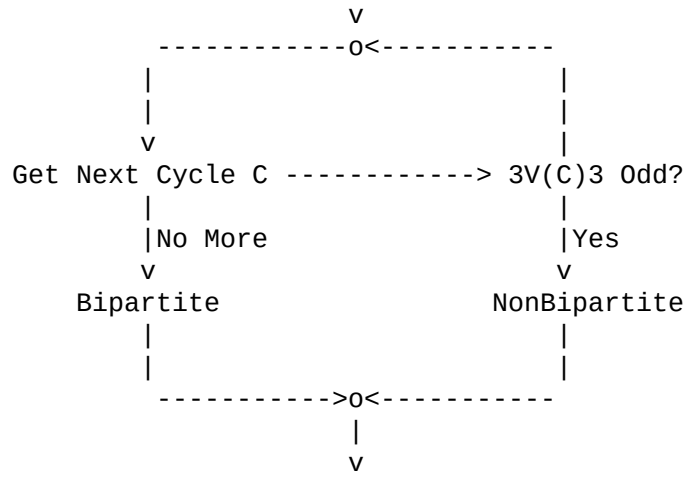


Figure 1-13. Exhaustive Search Algorithm for Testing Bipartiteness.

SECTION 1-4

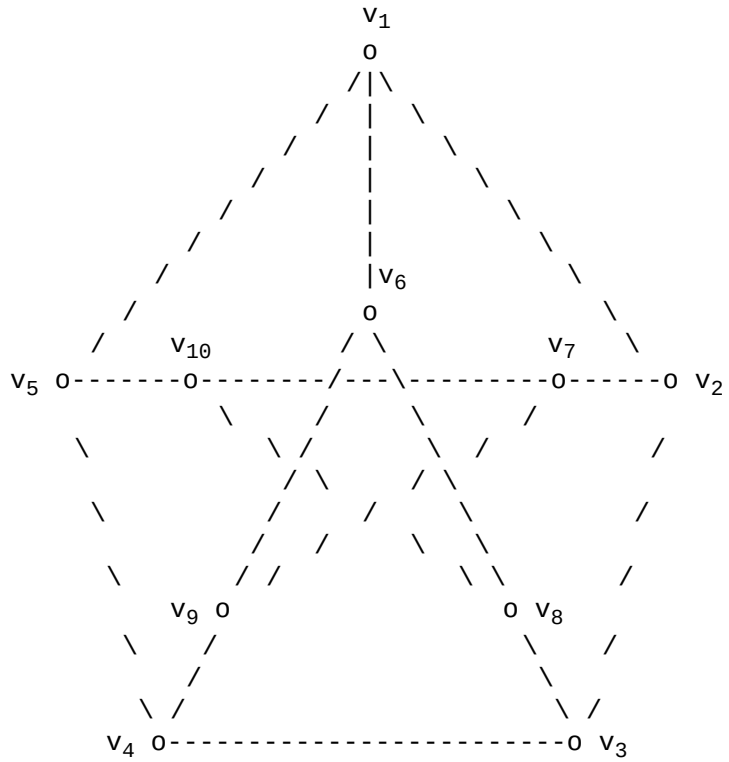


Figure 1-14. Petersen's Graph: The Unique Five-Cage.

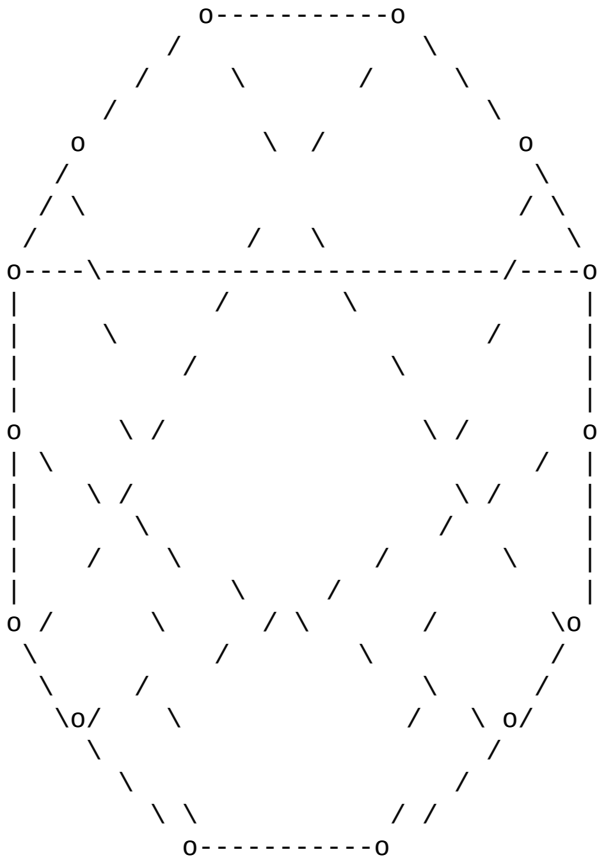


Figure 1-15. Heawood's Graph: The Unique Six-Cage.

SECTION 1-5-1

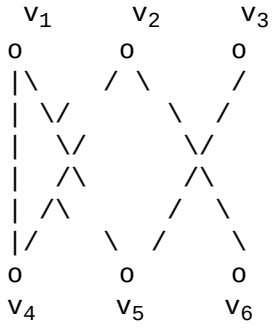


Figure 1-16. Bigraph.

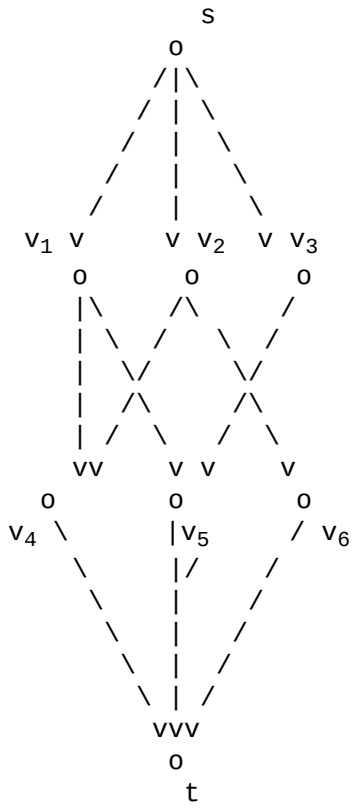
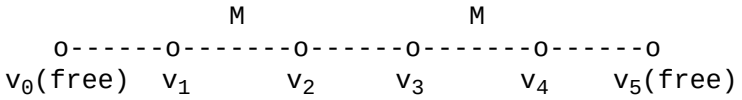
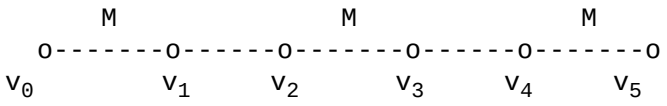


Figure 1-17. Flow Network with Unit Capacity Edges.

SECTION 1-5-2

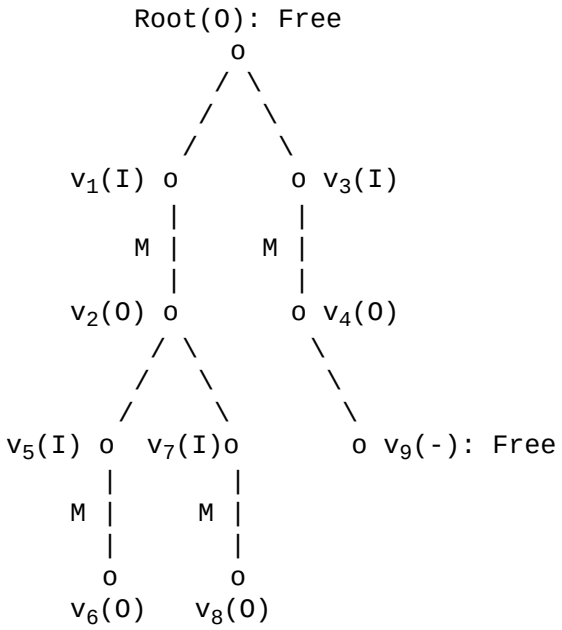


(a) Augmenting Path.



(b) After Augmentation.

Figure 1-18. Application of an Augmenting Path.



Notation: Inner - I, Outer - O.

Figure 1-19. Search Tree.

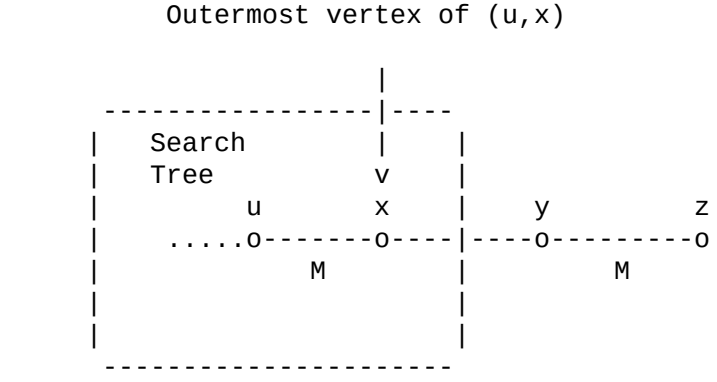
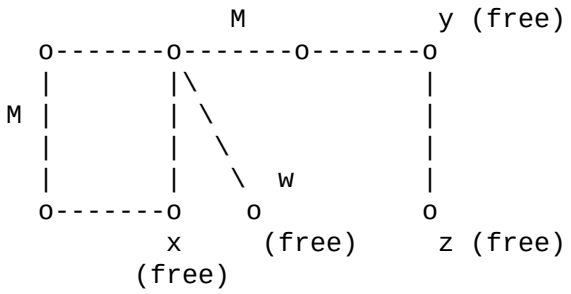
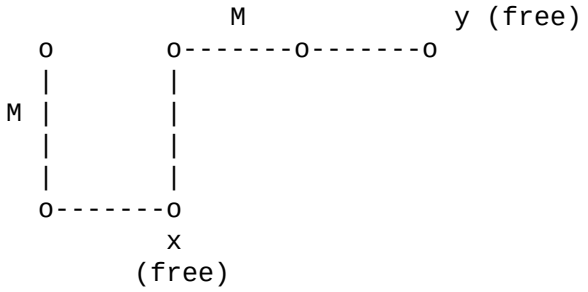


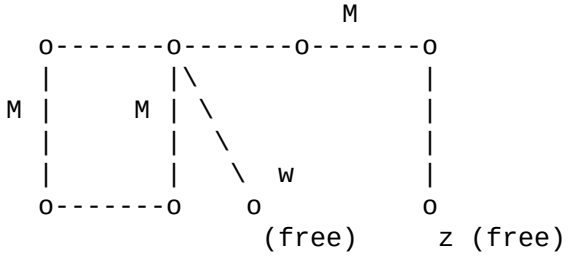
Figure 1-20. Extension of Search Tree Using the Pair of Edges (x,y) U (y,z).



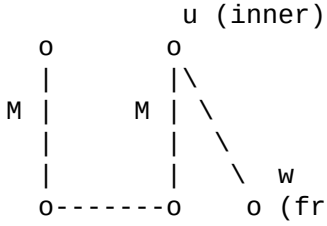
(a) Matching in G.



(b) Augmenting Tree at x.



(c) Revised Matching in G.



(d) Hungarian Tree H at w.

(e) G - H (also Hungarian).

Figure 1-21. Maximum Matching on Bipartite Graph.

SECTION 1-5-3

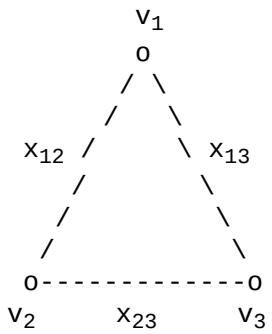
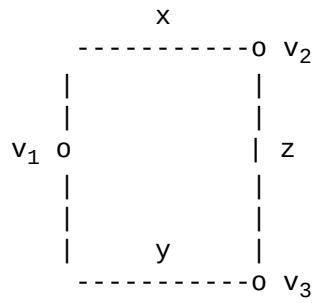


Figure 1-22. Maximum Matching Problem.

SECTION 1-5-4



(a) Example Graph K(3).

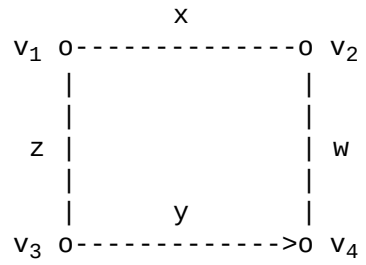
$$\begin{matrix} 0 & x & y \\ -x & 0 & z \\ -y & -z & 0 \end{matrix}$$

(b) Tutte matrix.

$$+xyz - xyz = 0$$

(c) Expanded Determinant.

Figure 1-23. Symbolic Determinant Example 1.



(a) Example Graph C(4).

$$\begin{matrix} 0 & x & z & 0 \\ -x & 0 & 0 & w \\ -z & 0 & 0 & y \\ 0 & -w & -y & 0 \end{matrix}$$

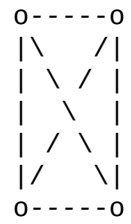
(b) Tutte matrix.

$$-2xyzw + x^2\, y^2 + w^2\, z^2$$

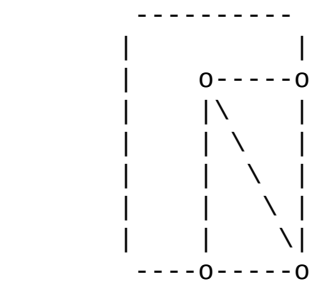
(c) Expanded determinant.

Figure 1-24. Symbolic Determinant Example 2.

SECTION 1-6



(a) Non-Planar Embedding of K(4).



(b) Planar Embedding of K(4).

Figure 1-25. Different Embeddings of K(4).

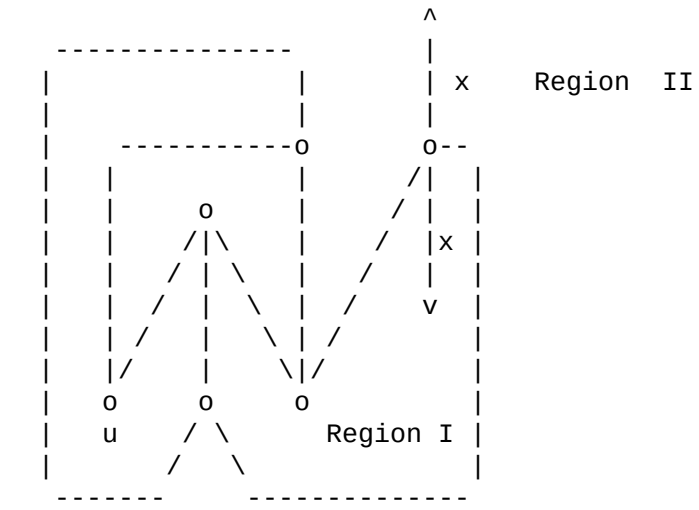


Figure 1-26. Attempted Planar Embedding of K(3,3).

Area 1: 2, 4, 6

Area 2: 1, 3, 4, 5, 6

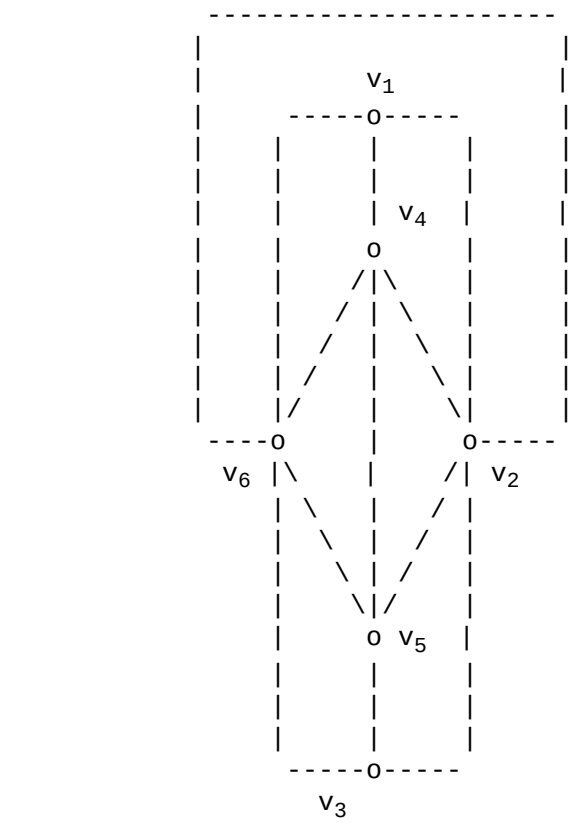
Area 3: 2, 5, 6

Area 4: 1, 2, 5, 6

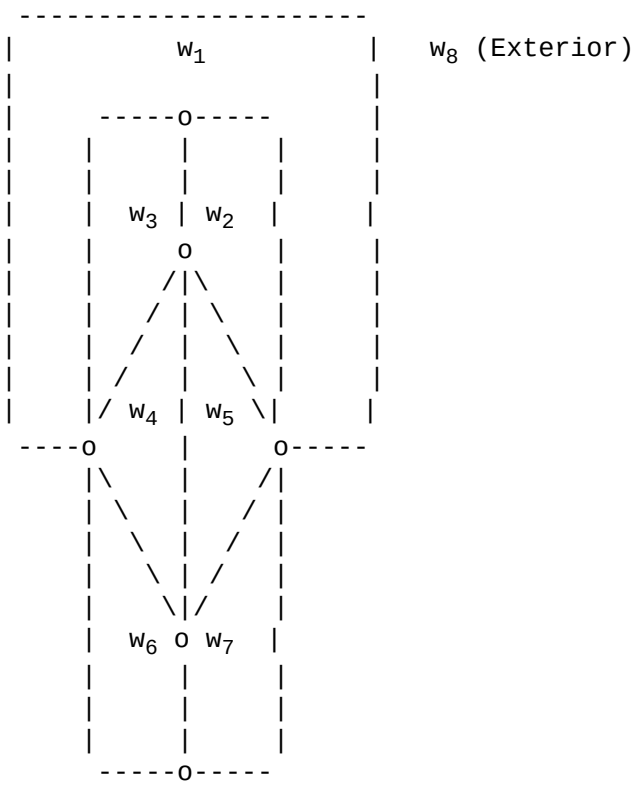
Area 5: 2, 3, 4, 6

Area 6: 1, 2, 3, 4, 5

(a) Area Adjacency Requirements.



(b) Graphical Representation of Adjacency Requirements.



ADJ(w_1): w_2, w_3, w_8

ADJ(w_2): w_1, w_3, w_5

ADJ(w_3): w_1, w_2, w_4

ADJ(w_4): w_3, w_5, w_6

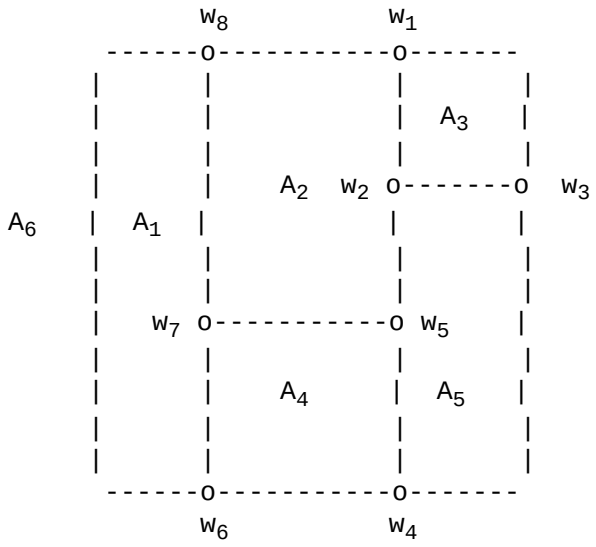
ADJ(w_5): w_2, w_4, w_7

ADJ(w_6): w_4, w_7, w_8

ADJ(w_7): w_5, w_6, w_8

ADJ(w_8): w_1, w_6, w_7

(c) Planar Dual Data.



(d) Planar Representation of Dual Graph.

Figure 1-27. Planar Dual and Adjacency Constraints.

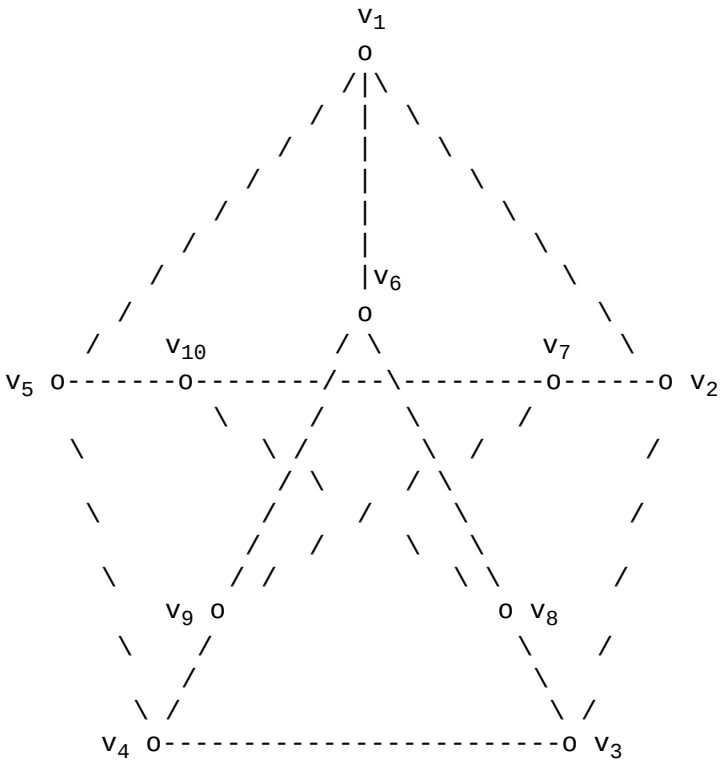


Figure 1-28. Petersen's Graph: Nonplanar.

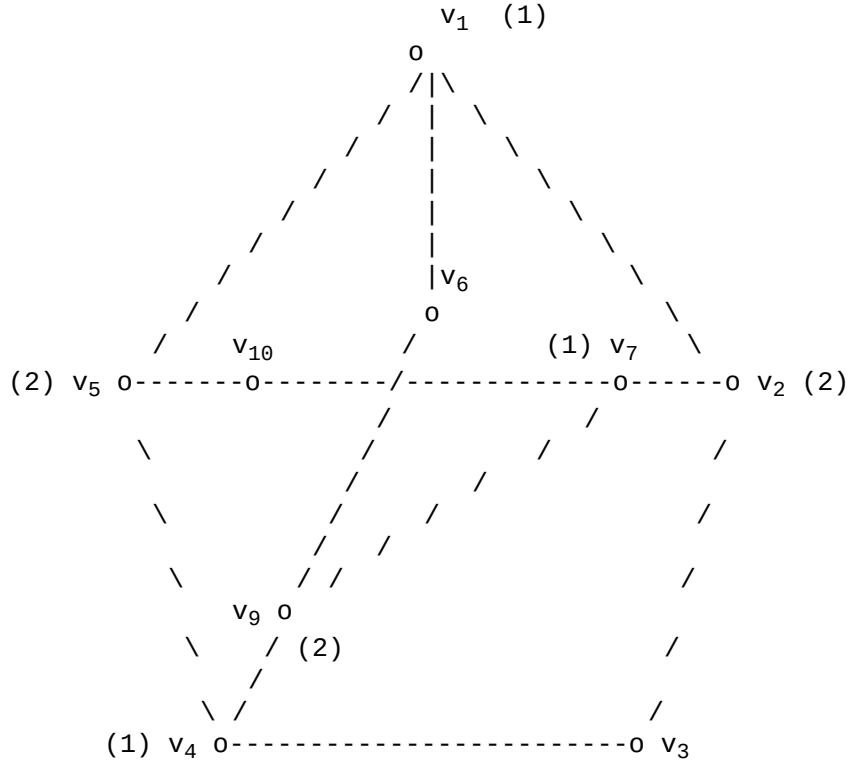
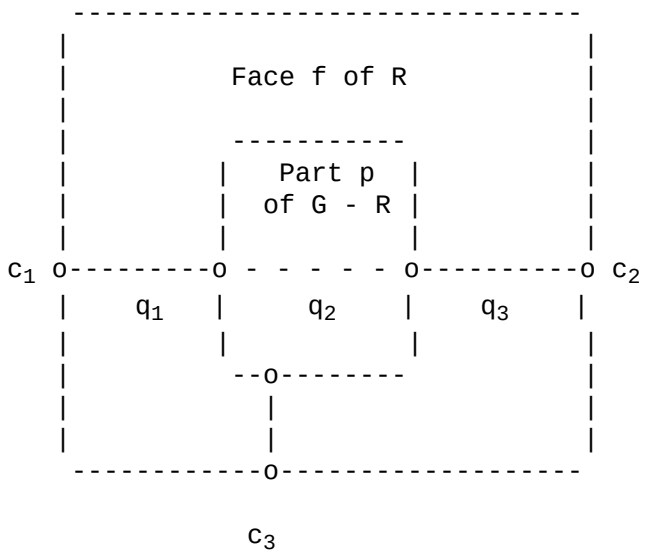


Figure 1-29. Subgraph of Petersen's Graph Homeomorhic to K(3,3).



Contact Vertices: c_1, c_2, c_3

Path q: q_1 U q_2 U q_3

Figure 1-30. Addition of Path q to R.

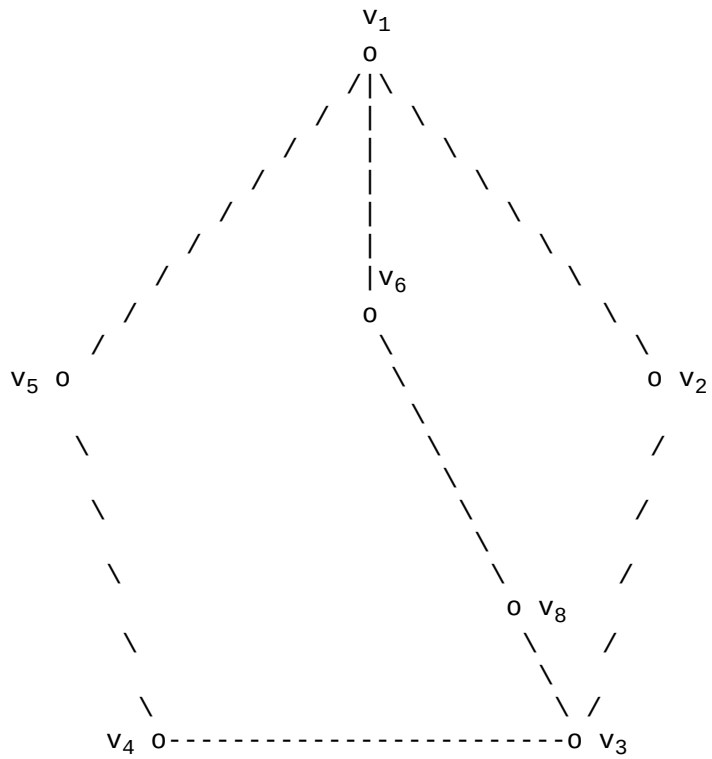
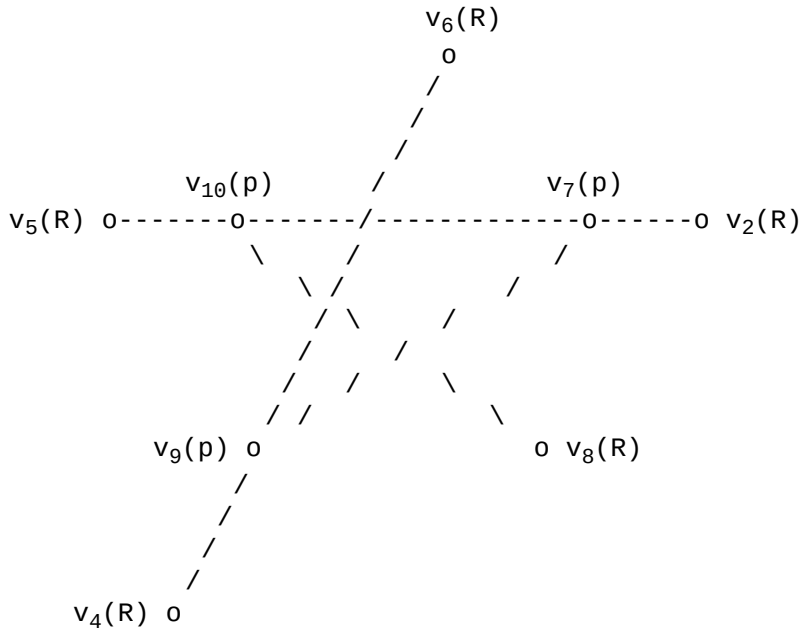


Figure 1-31. Second R Construct for Petersen's Graph.



V(p): v_7, v_9, v_{10}

Contact Vertices of p: v_6, v_5, v_4, v_8, v_2

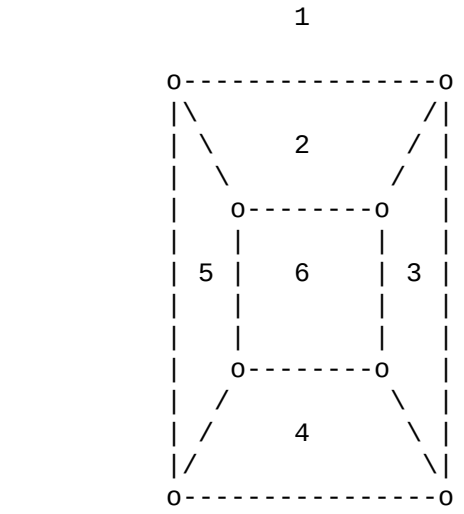
Faces of R:

f_1: {v_1, v_5, v_4, v_3, v_8, v_6}

f_2: {v_1, v_2, v_3, v_8, v_6}

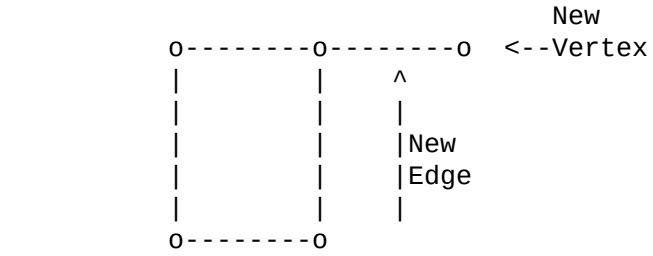
f_3: (v_1, v_2, v_3, v_4, v_5}

Figure 1-32. A Unique Part p Relative to R.

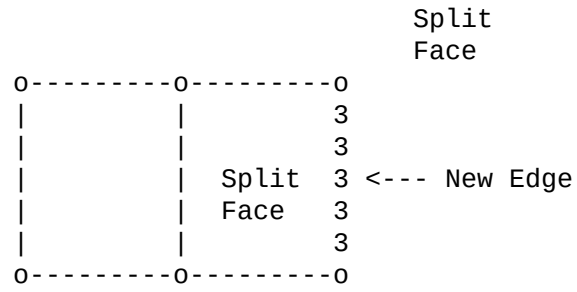


|V| = 8, |E| = 12, |F| = 6

Figure 1-33. Illustration for |V| + |F| = |E| + 2.



(a) |V| and |E| Both Increase by 1.



(b) |F| and |E| Both Increase by 1.

Figure 1-34. The Different Effects of Adding an Edge.

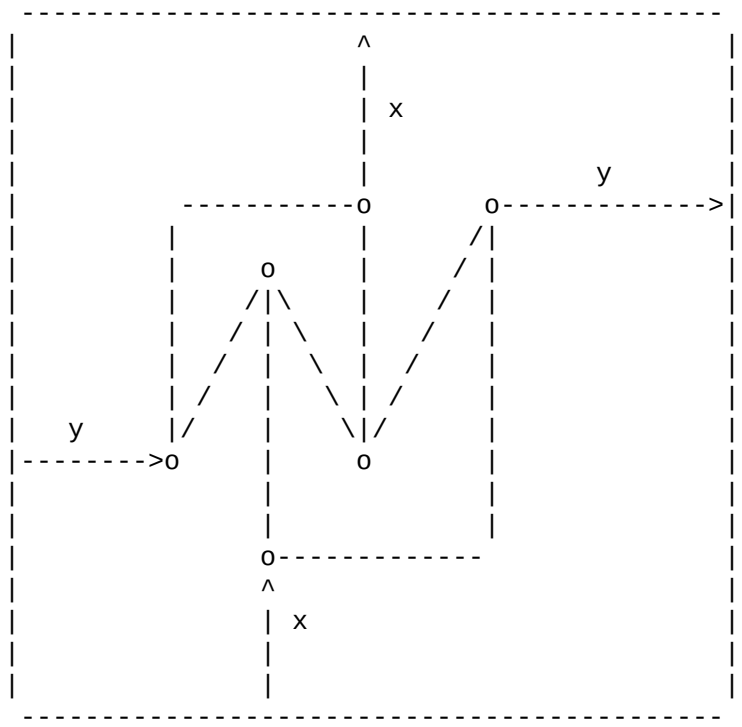
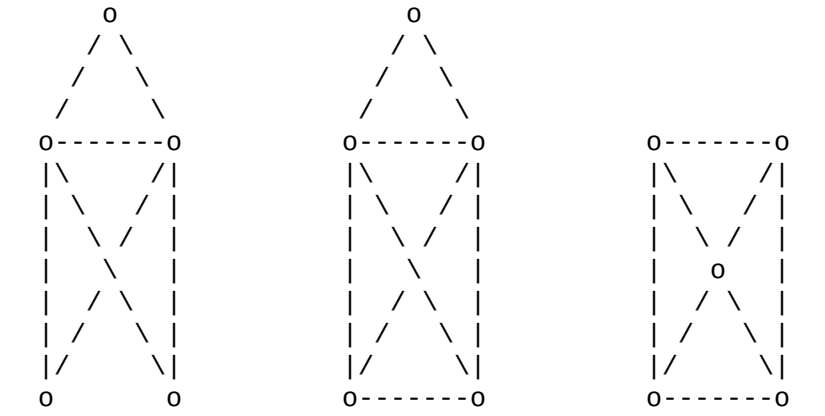


Figure 1-35. Embedding of K(3,3) on Torus.

Section 1-7



(a) Eulerian (b) Semieulerian (c) Noneulerian

Figure 1-36. Example Graphs.

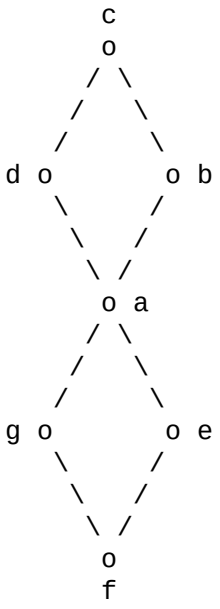


Figure 1-37. Fleury Example.

SECTION 1-8

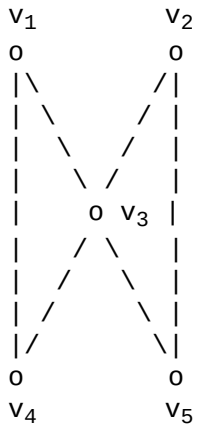
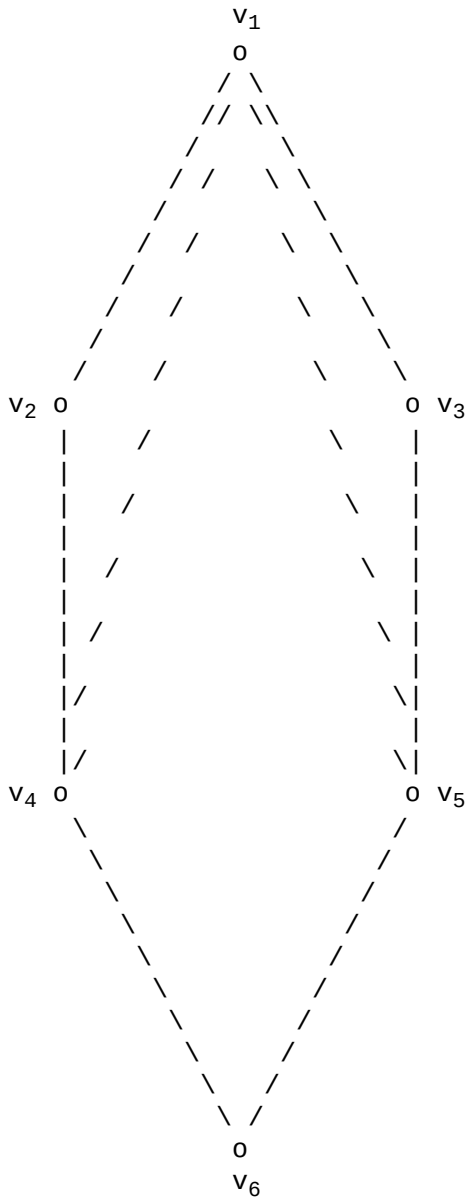
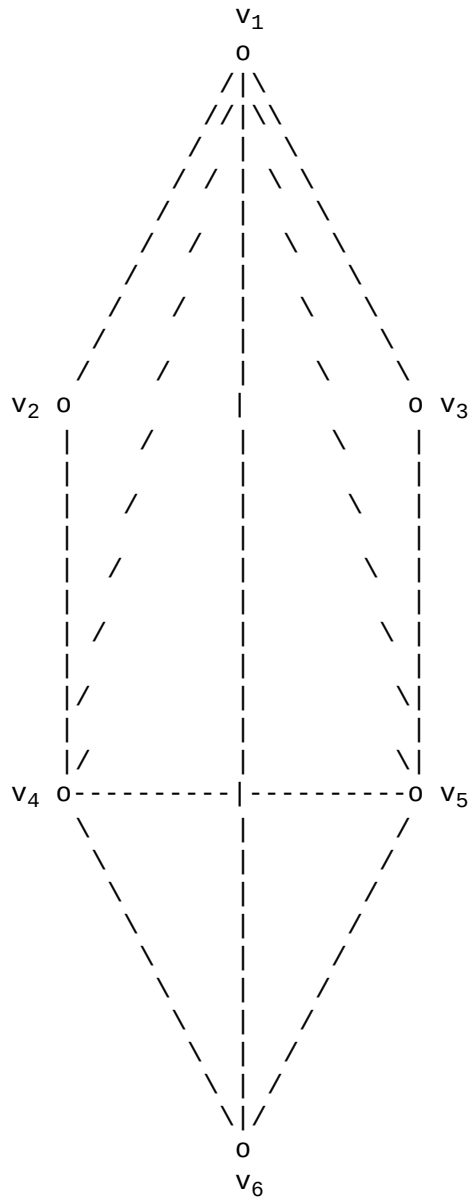


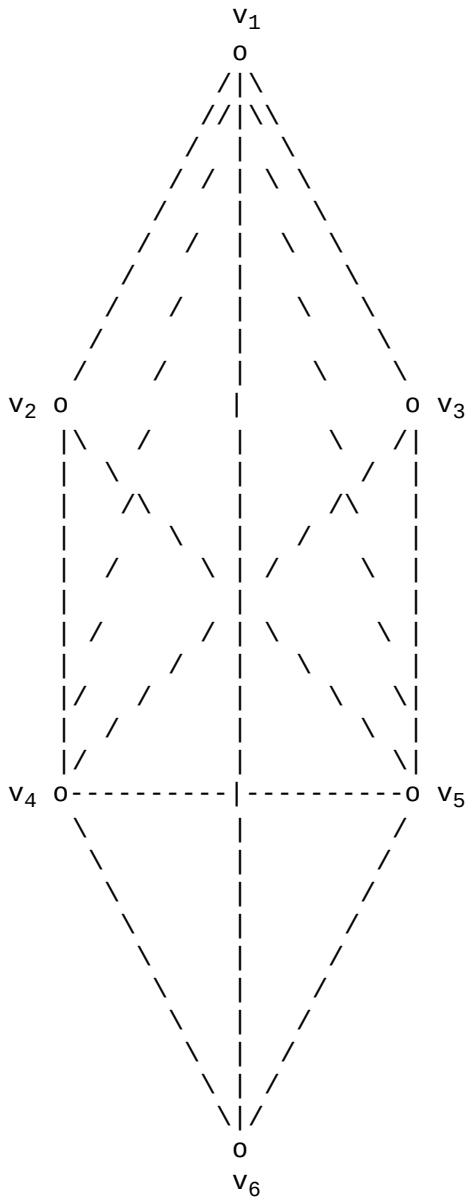
Figure 1-38. Sharpness of Ore Hamiltonian Condition.



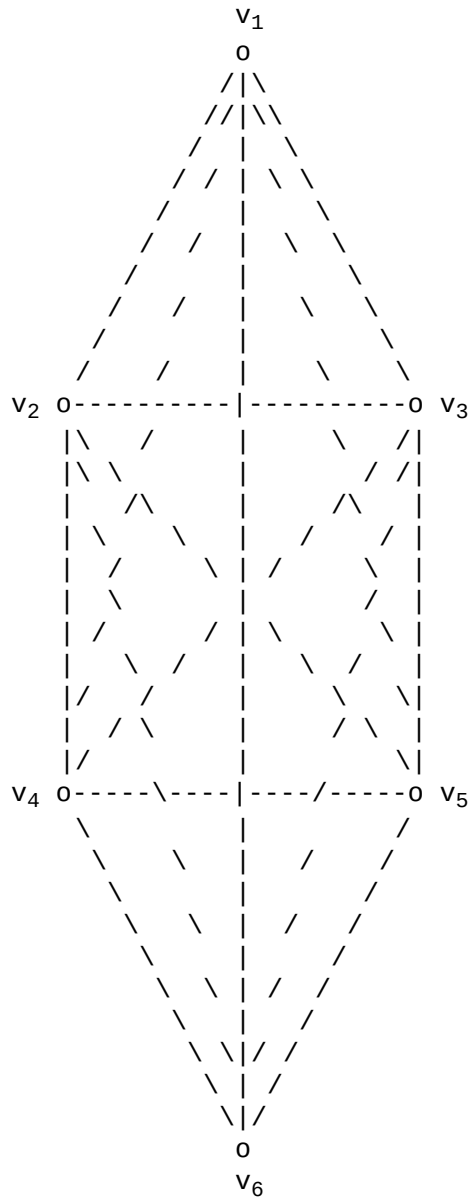
(a) Graph G whose Closure is to be Found.



(b) First Closure Iteration.



(c) Second Closure Iteration.



(d) Third Closure Iteration.

Figure 1-39. Hamiltonian Closure Condition.

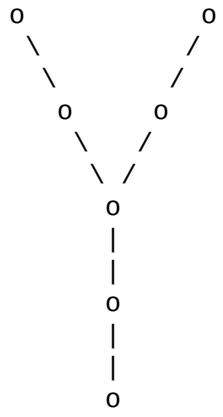


Figure 1-40. One Connected Graph with Nonhamiltonian Square.

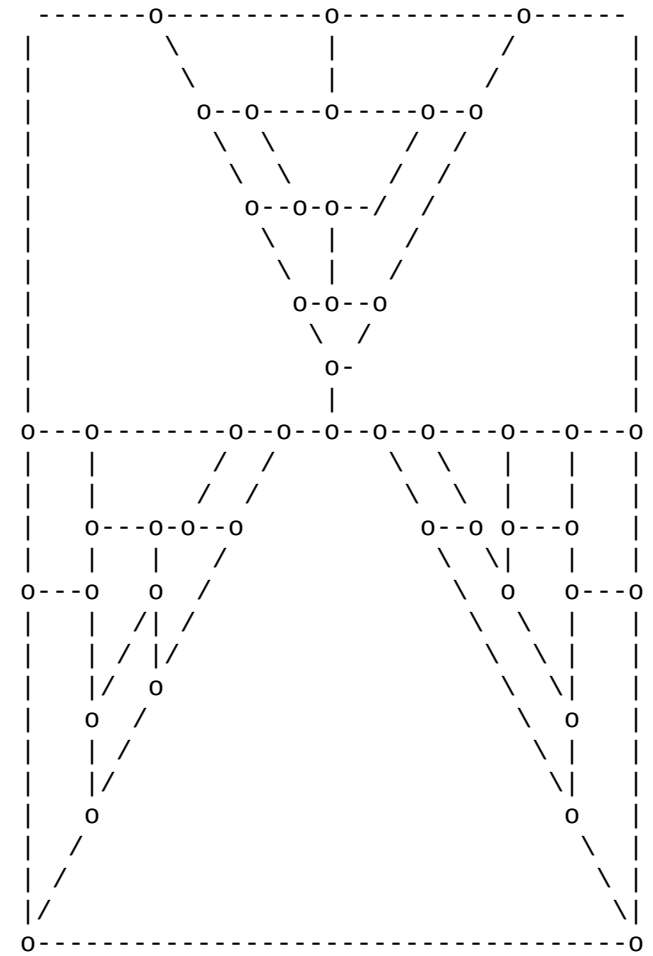


Figure 1-41. Cubic Planar Three-Connected but Nonhamiltonian.

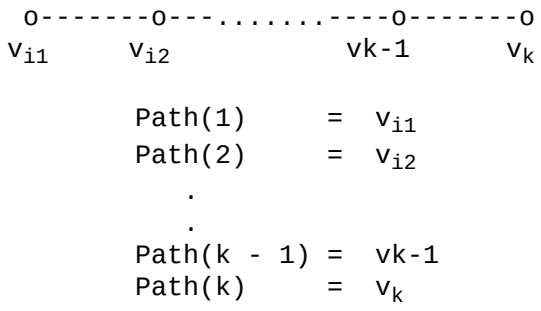
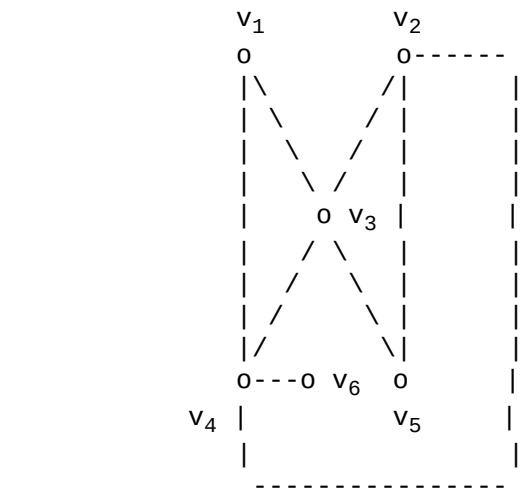


Figure 1-42. Path Notation for Backtracking.



(a) Example for Backtracking Algorithm.

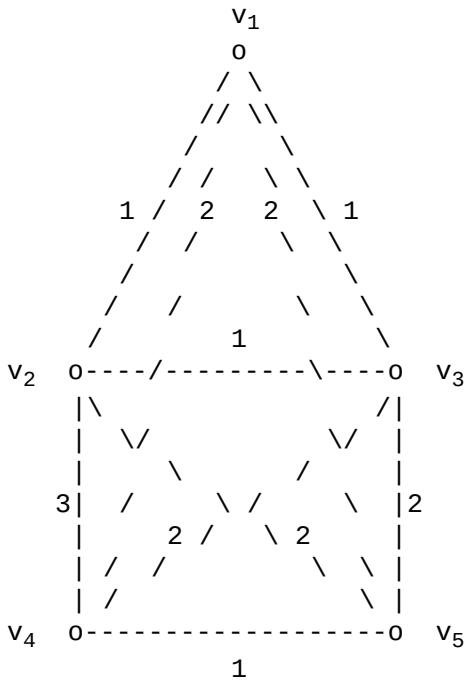
v_1
v_1 v_3
v_1 v_3 v_2
v_1 v_3 v_2 v_4
v_1 v_3 v_2 v_4 v_6 * Backup
v_1 v_3 v_2 v_4 * Backup
v_1 v_3 v_2 v_5
v_1 v_3 v_2 * Backup
v_1 v_3 v_4
v_1 v_3 v_4 v_2
v_1 v_3 v_4 v_2 v_5
v_1 v_3 v_4 v_2 * Backup
v_1 v_3 v_4 v_6
v_1 v_3 v_4 *
v_1 v_3 v_5
v_1 v_3 v_5 v_2

v_1 v_3 v_5 v_2 v_4
v_1 v_3 v_5 v_2 v_4 v_6 Hamiltonian Path

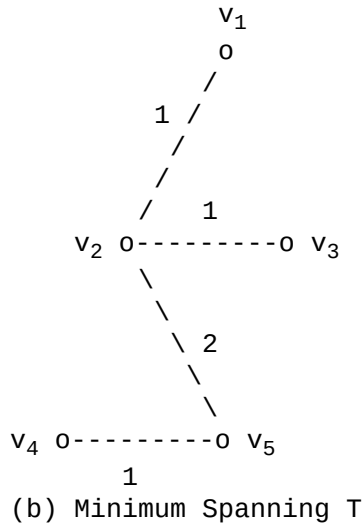
* Indicates a deadend entailing backup.

(b) Trace of Backtrack Algorithm.

Figure 1-43. Example for Find_Hamiltonian_Path.



(a) Graph G with Euclidean Weights.



(b) Minimum Spanning Tree for G.

		Length
C_0:	$v_1\ v_2\ (v_3\ v_2\ v_5)\ v_4\ v_5\ v_2\ v_1$	10
C_1:	$v_1\ v_2\ v_3\ v_5\ (v_4\ v_5\ v_2)\ v_1$	9
C_2:	$v_1\ v_2\ v_3\ v_5\ (v_4\ v_2\ v_1)$	9
C_3:	$v_1\ v_2\ v_3\ v_5\ v_4\ v_1$	7

(c) Successive Euclidean Reductions of Initial Circuit C_0.

Figure 1-44. Euclidean TSP Approximation Example.