ENUMERATION IN GRAPHS

G.J. Bezem and J. van Leeuwen

RUU-CS-87-7

May 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

ENUMERATION IN GRAPHS

G.J. Bezem and J. van Leeuwen

RUU-CS-87-7

May 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

ENUMERATION IN GRAPHS

G.J. Bezem and J. van Leeuwen

Technical Report RUU-CS-87-7
May 1987

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
The Netherlands

Contents

1. Introduction

The enumeration of certain kinds of subgraphs as well as the enumeration of certain subsets of the set of vertices or edges of a graph is of interest in many practical applications, for example the theory of electrical and communication networks, the reliability analysis of networks, and data storage and retrieval problems. This report surveys a number of algorithms that enumerate all subgraphs or subsets of a certain kind. It is organized as follows. In the next section of this chapter we give some definitions. In chapters 2 through 7 we discuss algorithms that enumerate all cycles, paths, spanning trees, cliques, maximal independent sets and cut-sets of a graph respectively.

1.1 Definitions

A *graph* $G = (V, E)$ consists of a set of *vertices* $V = \{v_1, v_2, \ldots, v_n\}$ and a set of *edges* $E = \{e_1, e_2, \ldots, e_m\} \subseteq V \times V$. The number of vertices will be denoted by n and the number of edges by e or m. Two vertices are *adjacent* if an edge exists between them. An edge and a vertex are *incident* if the vertex is a terminal vertex of the edge. A graph G is called *undirected* if the edges are unordered pairs of vertices and G is called *directed* otherwise. A *null graph* is a graph $G = (V, E)$ with E empty. A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. A *walk* is a sequence $v_1 e_1 v_2 e_2 \cdots v_p e_p v_{p+1}$ of vertices and edges such that e_i is an edge from v_i to v_{i+1} for each $i = 1, \ldots, p$. The length of a walk is its number of edges. A (*simple*) *path* is a walk in which no vertex appears more than once. A (*simple*) *cycle* or *circuit* is a (simple) path of which the initial and terminal vertex coincidence. In case G is directed, a path is sometimes called a *dipath* and a circuit is sometimes called a *dicircuit*. A Hamiltonian cycle (path) is a cycle (path) which contains each vertex of the graph. If a path exists between each pair of vertices, then G is *connected*, otherwise G is *disconnected*. A connected graph that contains no cycles is called a *tree*. A *spanning tree* of a connected graph G is a connected subgraph which is a tree and which contains each vertex of G. Let T be a spanning tree of a graph G. The edges of G that belong to T are called the *branches* of G with respect to T. The edges of G that do not belong to T are called the *chords* of G with respect to T. Adding a chord to T creates a cycle in T, called a *fundamental cycle* of G with respect to T. Each chord of G creates a different cycle in T, and thus there exists a one-to-one correspondence between the chords of G and the fundamental cycles of G with respect to T. A *cut-set* of G is a set C of edges with the property that the removal of the edges in C from G disconnects G, but the removal of any proper subset of C from G does not disconnect G. Deleting a branch from a spanning tree T of G partitions the vertices of G into two disjoint subsets V_1 and V_2 such that in T there exists no path from any vertex of V_1 to any vertex of V_2. Let C be the set of all edges of G that join vertices of V_1 to vertices of V_2. C is a cut-set of G and is called a *fundamental cut-set* of G with respect to T. There exists a one-to-one correspondence between the branches of T and the fundamental cut-sets of G with respect to T. Let s and t be vertices of G. An *$s-t$ cut-set* of G is a cut-set C with the property that the removal of the edges of C from G results in a graph in which there is no path from s to t.

If there exists an edge between each pair of vertices of G, then G is a *complete graph*. A *clique* of G is a complete subgraph that is not contained in any other complete subgraph of G. An *independent set* I of G is a subset of V such that no vertex in I is adjacent to any other vertex in I. A *maximal independent set* is an independent set to which no vertex can be added without losing the independency property. Finally we define an operation on sets of subgraphs. Let H_1 and H_2 be sets of subgraphs of a graph G. The *Cartesian product* $H_1 \times H_2$ of H_1 and H_2 is the set of all subgraphs that are obtained as the union of an element of H_1 and an element of H_2. For definitions not mentioned in this section we refer to [32] or [21].

1.2 The time complexity of enumeration algorithms and counting problems

If the time complexity of the algorithms discussed is available, then it will be given. The time complexity generally is expressed as a function of n, e and x, where x is the number of objects that are listed. This number can be exponentially large in n. In that case, the amount of time required by the algorithm to list all objects is not polynomial in n, although the time complexity of the algorithm may be polynomial in x. If we are only interested in the number x, then it is not always necessary to list all objects. For example, the number of spanning trees of a graph can be obtained by evaluating a determinant. The problem of determining the number of all objects of a certain kind is called an *enumeration* or *counting problem.* Some enumeration problems can be solved in polynomial time, even if the number to be counted is exponential, but many enumeration problems are not solvable in polynomial time. If an enumeration problem corresponds to a $NP-$complete problem, then a suitable formulation of the enumeration problem is $NP-$hard (for the definitions of $NP-$complete and $NP-$hard and for related terminology we refer to Garey and Johnson ([28])). For example, the problem of determining the number of Hamiltonian circuits corresponds to the problem of determining whether a graph has a Hamiltonian circuit. If we have computed the number of Hamiltonian circuits of a graph, then we can easily decide whether the graph has a Hamiltonian circuit. On the other hand, if we could decide in polynomial time whether a graph has a Hamiltonian circuit, it does not necessarily mean that we are able to compute the number of all Hamiltonian circuits in polynomial time. Thus, some enumeration problems may be harder than the corresponding $NP-$complete problem. In [91] and [92] Valiant introduces a new class of polynomial time equivalent problems, the *$\#P-$complete* problems. $\#P$ is the class of functions that can be computed by counting the number of accepting computations of a nondeterministic Turing machine of polynomial time complexity. A function f is called *$\#P-$complete* if f is in $\#P$ and every function g in $\#P$ can be reduced to f by a polynomial time reduction. Valiant ([92]) and Provan and Ball ([69]) give some counting problems that are $\#P-$complete, such as the problem of determining the number of all cliques, all $s-t$ cut-sets, all Hamiltonian circuits or all (simple) paths from vertex s to vertex t of a graph. In this report we do not consider the theory of $\#P-$complete problems in any further detail.

2. Cycles

The algorithms for enumerating all cycles of a graph can be divided into four classes, depending on their underlying approach. These four classes are the following.

1) search space algorithms
2) backtrack algorithms
3) algorithms using the powers of the adjacency matrix
4) algorithms using the line digraph

Algorithms for directed graphs can also be used for undirected graphs as follows. From an undirected graph G we obtain a directed version $\vec{G}$ by replacing each edge of G by two edges of opposite direction. For each cycle of G we have two dicircuits in $\vec{G}$, one in each direction. In addition we have created e dicircuits of length 2, so if $c(G)$ is the number of cycles of G then $c(\vec{G}) = 2c(G) + e$. After we have found all dicircuits of $\vec{G}$, we have to do some additional work to get the cycles of G. On the other hand, algorithms for undirected graphs can not be used for directed graphs by taking the undirected version G of $\vec{G}$, for the number of cycles of G can be exponentially larger than the number of dicircuits of $\vec{G}$. Consider for example the directed graph in fig. 1. This graph has no dicircuits at all, whereas its undirected version has 2^k cycles.

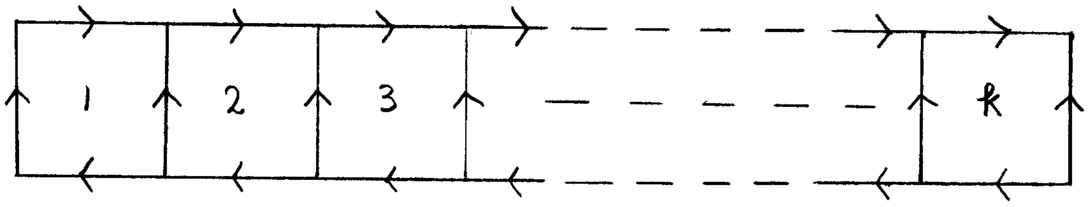

figure 1

In the next four sections we shall discuss the different classes of algorithms. Some of the algorithms assume that G has no self-loops or parallel edges. These and other simplifications of G can be made in any case. Self-loops can be listed and then deleted from G, because they are never contained in a cycle of length greater than 1. Parallel edges can be replaced by one edge, and edges in series can also be replaced by one edge.

2.1 Search space algorithms

Algorithms using this approach search for circuits in an appropriate search space containing all circuits in some representation. The efficiency of such an algorithm, of course, depends on the size of the search space with respect to the number of circuits, and the effort it takes to decide whether an element of the search space is a cycle. The algorithms of LaPatra and Myers ([43]), Char ([13]) and Chan and Chang ([12]) search for cycles in the set of all permutations of the vertices of a graph. Choosing this search space does not yield a very efficient algorithm. A more promising search space that has been studied widely is the cycle vector space of an undirected graph. The cycle vector space $C(G)$ of an undirected graph G is defined as the set of all cycles of G together with the empty set and the set of edge-disjoint unions of cycles. $C(G)$ is a vector space over F_2, the field of integers modulo 2, with the addition of any two elements in $C(G)$ defined as the ring sum of the set of edges of the elements, i.e., if C_1 and C_2 are elements of $C(G)$ then the edges of $C_1 \oplus C_2$ are those edges that are either in C_1 or in C_2 but not in both. Several algorithms which search for cycles in the cycle vector space compute all the elements of $C(G)$ and test whether an element is a cycle. The algorithms differ in their way of representing and computing the elements of $C(G)$. We outline a number of approaches.

Let T be a spanning tree of the undirected graph G. The set of fundamental cycles $C_1, \dots, C_\mu$ of G induced by the set of chords $c_1, \dots, c_\mu$ of G with respect to T is a basis of $C(G)$. The circuit matrix B of G (with respect to T) is the $\mu \times e$ - matrix in which the rows correspond to the fundamental cycles and the columns correspond to the edges of G, with $B_{ij} = 1$ if edge e_j is in fundamental cycle C_i and $B_{ij} = 0$ otherwise. Using this circuit matrix Maxwell and Reed ([52]) generate a listing of all the elements of $C(G)$ by simply enumerating all non-trivial combinations of fundamental cycles in terms of their edges and store them in a $(2^\mu - 1) \times e$ - matrix F. A row of F represents a cycle if and only if no other row is contained in it. After setting up this matrix F, Maxwell and Reed test which row is a cycle by comparing each row with every other row. A lower bound on the time for this algorithm is $2^{2\mu}$ and moreover, it needs $O(e . 2^\mu)$ space. Rao and Murti ([70],[71]) show that space can be saved. Let $S = \bigoplus_{i \in I} C_i$ be an element of $C(G)$ for some index set $I \subseteq \{1, \dots, \mu\}$. Instead of listing the edges, S can also be represented by a μ - vector X with $X_j = 1$ if $j \in I$ and $X_j = 0$ otherwise. Define a $\mu \times (n-1)$ - matrix F as follows. The rows of F correspond to the chords $c_1, \dots, c_\mu$ and the columns of F correspond to the branches $b_1, \dots, b_{n-1}$ of T. If the unique path in T between the two vertices incident to edge c_i contains branch b_j then $F_{ij} = 1$, and else $F_{ij} = 0$. Use the vector representation of S together with the matrix F to compute the edges of S. We now decide whether S is a cycle as follows. Let the edges of S contain the branches $b_1, \dots, b_s$ and the chords $c_1, \dots, c_k$ and let $m = s + k$. If S is a cycle then a tree T' can be obtained from T by adding one of the chords $c_1, \dots, c_k$ and deleting a branch that is not an edge of S. This procedure is repeated until a spanning tree containing $m-1$ edges of S is found, or until it is not possible after adding a chord to obtain a new tree without deleting a branch that is an edge of S. In the last case S is not a cycle. In the first case S is a cycle, because if S was an edge-disjoint union of cycles, then the $m-1$ edges of S in the spanning tree should contain a cycle, which is not possible. Rao and Murti find all cycles of G by applying this procedure to all linear combinations of fundamental cycles. A lower bound on the time is 2^μ and the space required is only $O(n . \mu)$.

Gibbs ([29]) gives another algorithm that computes all elements of $C(G)$. He uses 4 sets S, Q, R and R^*, and a set of fundamental cycles $C_1, \dots, C_\mu$. The algorithm proceeds in stages. After stage $(i-1)$ of the algorithm, S contains all cycles found so far, Q contains all linear combinations of

$C_1, \ldots, C_{i-1}$, and the sets R and R^* are empty. In stage i all linear combinations of C_i with elements of Q are considered. If $T \in Q$ and $T \cap C_i = \emptyset$, then $T \oplus C_i$ is an edge-disjoint union of cycles and is placed in R^*. If $T \cap C_i \neq \emptyset$, then $T \oplus C_i$ is placed in R. In that case $T \oplus C_i$ is a cycle or an edge-disjoint union of cycles.

Lemma. *If $T \oplus C_i \in R$ is a union of two or more edge-disjoint cycles, then there is an $U \in R$ with $U \subset T \oplus C_i$.*

It follows that every $V \in R$ for which there is no $U \in R$ with $U \subset V$ is a cycle, and that all new cycles "involving" C_i are obtained this way. In S are placed C_i and all $V \in R$ for which there is no $U \in R$ with $U \subset V$. All elements of R and R^* are placed in Q. Afterwards R and R^* are made empty again and we go to stage $i+1$. The algorithm ends after stage μ. It should be noted that this algorithm is a modification of an algorithm of Welch ([96]) of which Gibbs shows that it is incorrect. All linear combinations of $C_1, \ldots, C_\mu$ are formed (and stored) and in each stage all pairs of all elements of R are tested, so the algorithm has $2^{2\mu}$ as a lower bound on the running time and requires $O(e.2^\mu)$ space.

$C(G)$ has 2^μ elements, which means that algorithms that compute all elements of $C(G)$ have always 2^μ as a lower bound on the time. In most cases however, the number of cycles of G is small compared with the number of elements of $C(G)$. In fact, only the 4 undirected reduced graphs K_3, K_4, K_4-x and $K_{3,3}$ have all elements $\neq 0$ of $C(G)$ as a cycle (Mateti and Deo, [50], [51]). To get a better time bound it is necessary to reduce the size of the search space. Let $C_1, \ldots, C_\mu$ be a cycle basis of G. The intersection graph G^c is defined as follows. The vertices of G^c (called 'nodes') are the cycles $C_1, \ldots, C_\mu$ of G and two nodes C_i and C_j $(i \neq j)$ are adjacent if $C_i \cap C_j \cap E(G) \neq \emptyset$. Mateti and Deo ([50]) call G^c a cycle graph of G.

Lemma. *If $C = \bigoplus_{i \in I} C_i$ is a cycle of G, then the subgraph of G^c induced by the nodes $C_i, i \in I$, is connected.*

Using this lemma Mateti and Deo give an algorithm that generates all connected induced subgraphs of G^c and tests whether the ring sum of the cycles corresponding to the nodes of the subgraph found is a cycle. However, not all connected induced subgraphs of G^c necessarily correspond to a cycle of G. The efficiency of this algorithm depends on how the connected induced subgraphs of G^c are generated and on the number of cycles of G with respect to the number of connected induced subgraphs of G^c, and this depends on the cycle basis chosen. This problem would be solved if we could choose a cycle basis such that there exists a 1-1 correspondence between the cycles of G and the connected induced subgraphs of G^c. However, in [83] Syslo shows that there are graphs for which such a cycle basis does not exist. In [84] Syslo shows that in case G is planar it is possible to modify the algorithm somewhat and to construct an efficient cycle vector space algorithm. He takes the boundaries of the interior regions of G as a cycle basis. A subgraph of G^c that corresponds to a cycle of G is called a 'feasible subgraph'. If F is a feasible subgraph, $v \in V(F)$ and the subgraph of G^c induced by $V(F) - v$ is also feasible, then v is called a 'feasible node'.

Lemma. *Let G be biconnected and let F be a feasible subgraph of G^c, then $v \in V(F)$ is feasible if and only if v is an exterior node of F and the cycle C_v of G corresponding to v has exactly one nontrival (i.e. different from a vertex) common path with the cycle $C_F = \bigoplus_{v \in V(F)} C_v$.*

The algorithm Syslo proposes consists of the following steps. Construct the plane representation of G and construct G^c using the interior regions of G. Mark the nodes of G^c as feasible or infeasible. Use a depth-first search to remove the feasible nodes one by one and mark the nodes of the obtained subgraph again as feasible or infeasible. This algorithm generates all feasible subgraphs of G^c (and thus all cycles of G) in $O(n)$ time for each subgraph, and so the overal complexity of the algorithm is $O(n.c)$, where c is the number of cycles.

2.2 Backtrack algorithms

The basic idea of all algorithms using a backtracking procedure is similar. Consider the vertices as numbered $1, \ldots, n$. Choose one vertex v_1 and build an elementary path with start vertex v_1. If we have build a path $v_1, \ldots, v_k$ and no extension is possible, then delete v_k from the path and continue with extending the path $v_1, \ldots, v_{k-1}$ until no unexplored paths with start vertex v_1 are left. Then choose another start vertex and repeat the procedure, excluding v_1. At given moments we test whether the path yields a new cycle. The question is how to choose the start vertex, how to choose the extension of the current path, and at what moment do we test for new cycles. In [87] Tiernan builds paths $v_1, \ldots, v_k$ such that $v_1 \le v_i, 2 \le i \le k$. When no extension is possible and an edge from v_k to v_1 exists, then the cycle $v_1, v_2, \ldots, v_k, v_1$ is enumerated, v_k is deleted and the algorithm continues with $v_1, v_2, \ldots, v_{k-1}$. This procedure is repeated with each vertex as start vertex. Since each cycle contains a smallest vertex v, it is enumerated exactly once, namely when v is the start vertex. The algorithm of Tiernan uses backtracking without any restriction. Each path $v_1, \ldots, v_k$ with $v_1 \le v_i$ for all $2 \le i \le k$ is generated (although some bookkeeping is performed to avoid a simple path to be generated more than once), but clearly not every path generated yields a cycle. Tarjan ([86]) gives a worst-case example in which the algorithm of Tiernan needs time exponential in n to enumerate $2n$ cycles. He also shows that it can be made much more efficient when we store the information about vertices of which we know that they are not contained in a cycle together with the current path. For this purpose he adds a marking mechanism to the algorithm of Tiernan. He uses two stacks, a point stack and a marked stack. The point stack keeps track of the path being build. The marked stack keeps track of vertices v that are not to be chosen as an extension of the current path, because they are in the current path or it is known that every path from v to start vertex s intersects the current path in a vertex other than s. When the current path is extended with vertex v, then v is placed at the top of the point stack and at the top of the marked stack. The path can only be extended with vertices that are neither on the point stack nor on the marked stack, and that are not smaller than the start vertex. When no extension from a vertex v is possible, then v is deleted from the point stack (at this moment v is the top of the point stack). If no cycle containing v has been found, that means every path from v to s intersects the current path in a vertex other than s, or if there is no path from v to s at all, then the marked stack does not change, so vertices that are not contained in a cycle together with the current path are stored temporarily. As long as they are stored on the marked stack they cannot be chosen as an extension of the current path, which avoids unnecessary searches. On the other hand, if a cycle containing v has been found, then *all* the vertices that are above v on the marked stack, including v, are deleted from that stack. Even the vertices of which we know that they are not contained in a cycle with s are deleted, so we still have to do unnecessary work if we arrive again at these vertices. Yet the algorithm of Tarjan has a better time bound than the algorithm of Tiernan. It requires $O(n + e)$ space and $O(n.e.(c + 1))$ time, where c is the number of cycles, which means a polynomial time bound per cycle.

As we have seen, Tarjan's algorithm is not yet optimal. In [39] Johnson gives a better version of the algorithm of Tarjan, by improving the marking mechanism (called 'blocking' by Johnson). Instead of using a marked stack, he uses a list $B(v)$ for each vertex v. At each moment, $B(v)$ contains those vertices w that are blocked and not in the point stack, and for which an edge from w to v exists. Choose a start vertex s. As in the algorithm of Tarjan, when a vertex is used as an extension of the current path, it is placed at the top of the point stack and it becomes blocked. When no extension from vertex v is possible, v is deleted from the stack. If no cycle containing v has been found, then v stays blocked and is added to the list $B(w)$ for each w for which an edge vw exists. If a cycle containing v has been found, then a procedure $UNBLOCK(v)$ is called, which 'unblocks' v and recursively calls $UNBLOCK(w)$ for each $w \in B(v)$. This means that once a vertex w has been blocked, it stays blocked until a vertex v is found such that there exists a path from w to v with all vertices blocked and not in the point stack, and with v contained in a cycle with start vertex s. When such a vertex v is found, w must be unblocked, because the path from w to v together with a part of the cycle containing v might be part of a cycle containing w and v, which is not yet enumerated. Another improvement over the algorithms of Tiernan and Tarjan is the way in which Johnson chooses his start vertex. Both Tiernan and Tarjan repeat their procedure with each vertex of G as start vertex. Johnson first computes the strongly connected components of G. The first start vertex s is the smallest vertex of these strong components. Let K be the component containing s. The algorithm computes all cycles in K containing s. Then the strongly connected components of the subgraph of G induced by $\{s+1, s+2, \ldots, n\}$ are computed and the next start vertex is the smallest one of these components. This procedure is repeated until $s = n$. The algorithm proposed by Johnson requires $O(n + e.(c + 1))$ time and $O(n + e)$ space.

The backtrack algorithms decribed so far all choose a start vertex s and then generate all cycles having s as the smallest vertex. During this computation, other cycles might be discovered but are not enumerated at this stage because they do not contain s (as the smallest vertex). Szwarcfiter and Lauer ([81]) use a modification of Johnson's blocking system in an algorithm that enumerates cycles as soon as they occur during the building of the elementary path. For each vertex v, the list $B(v)$ now contains those vertices u for which $uv \in E$ and the exploration of edge uv has not led to a new cycle. In addition Szwarcfiter and Lauer keep a *position* vector, with position $(v) = j$ if v is the j^{th} vertex from the bottom of the stack and position $(v) = n+1$ if v is deleted from the stack. The basic idea is still the same, namely to build elementary paths by extending the current path with vertices that are not marked. A vertex becomes marked when it is placed on the point stack. When it is deleted from the stack, vertex v is unmarked only if a new cycle containing v has been found. Otherwise it stays marked until a vertex z_1 is deleted from the stack such that a new cycle containing z_1 was found and there exists a path $z_k z_{k-1} \cdots z_1$ with $z_k = v$ and $z_{i+1} \in B(z_i)$ for $k < i \leq 1$. Let v_{k-1} be at the top of the stack and let us explore the edge $v_{k-1}v_k$ with v_k marked. The current path can not be extended with v_k. If v_k is not on the stack then no cycle can be generated. In that case v_{k-1} is inserted in $B(v_k)$. If v_k is still on the stack, then a cycle is found but this is not necessarily a new cycle.

Lemma. *A cycle is a new cycle if and only if at least one of its vertices has never been deleted from the stack.*

Swarcfiter and Lauer keep a variable q local to the recursive procedure, and q equals the position of the top-most vertex in the stack that has been deleted. If position $(v_k) \leq q$, then a new cycle is found. Otherwise a duplicate cycle is found, which means that edge $v_{k-1}v_k$ has not led to a new elementary cycle, so v_{k-1} is inserted in $B(v_k)$. If no extension can be made from a vertex v, then v is deleted from the stack and the algorithm backs up one vertex. The algorithm works on the strongly connected components of the problem graph. Szwarcfiter and Lauer show that when started with a vertex of maximal indegree of a strongly connected component, the algorithm generates all cycles of this component in $O(e)$ time per cycle for any cycle except for the first enumerated, whose time bound is $O(n + e)$. Thus the overal compexity of the algorithm is $O(n + c.e)$, and it requires $O(n + e)$ space.

Szwarcfiter and Lauer were not the first to generate cycles as soon as they occur somewhere during the generation of the elementary paths. In [95] Weinblatt already did so. Vertices and edges of the current path are stored on a stack. As an extension of the current path we choose an edge that has not yet been explored. So each edge is explored exactly once. When we arrive at a vertex v where we have been before, then there are two cases. If v is still in the stack, we have found a new cycle which is then enumerated. If v is not in the stack anymore, the algorithm starts searching in the cycles that have been enumerated so far, for a path that starts in v and terminates in a vertex that is still on the stack. All paths thus found together with a part of the stack form new cycles. In any case we go back to the last vertex on the stack from which an edge originates that has not yet been explored. The vertices and edges on the stack that are passed, are deleted from the stack. If the stack is empty, we select another start vertex by choosing a vertex that has not yet been examined. Although each edge is explored only once, yet the searching for paths in cycles already enumerated requires much time. Tarjan ([86]) gives an example in which the algorithm requires time exponential in n to enumerate $2n + 2$ cycles.

A slightly different approach was chosen by Read and Tarjan ([74]). First they repeatedly do a depth-first search, so the graph is divided into a set of depth-first search trees which together span the vertices of the graph, and a set of non-tree arcs. These non-tree arcs can be cycle arcs, forward arcs, and cross arcs. During the depth-first searches, the vertices are numbered $1, \ldots, n$. Tree edges and forward arcs join smaller to larger numbered vertices, and cycle arcs and cross arcs join larger to smaller vertices ([85]).

Lemma. *Each cycle arc is an edge of at least one cycle and each cycle must end in a cycle arc.*

After the depth-first searches, Read and Tarjan divide the graph into strongly connected components, and delete the edges between the components. Each vertex with an entering cycle arc is marked as a start vertex. For each start vertex s they repeat the following procedure to build an elementary path. From the last vertex v in the current path they search for a vertex w such that $vw \in E$ and a path $w = w_1, w_2, \ldots, w_k = s$ that avoids the current path (except at s) exists. After such a path is found, the vertices $w_1, w_2, \cdots$ are added one by one to the current path, until a vertex w_i is entered from which more than one edge originates. At this moment the whole procedure is called recursively. The adding of $w_1, \ldots, w_{i-1}$ is done without recursion. After the recursive procedure is terminated, all vertices after v in the current path are deleted and another vertex w' is searched such that $vw' \in E$ and a path from w' to s that avoids the current path exists. If no such path is found, v is deleted from the current path and the recursive procedure which was called to examine paths starting in v is terminated. This algorithm has a time bound of $O(n + e.(c + 1))$ and requires $O(n + e)$ space, which is both the same as the algorithms of Johnson ([39]) and Szwarcfiter and Lauer ([81]).

2.3 Algorithms using the powers of the adjacency matrix

These algorithms compute the powers of (a modification of) the adjacency matrix. The *adjacency matrix* of a graph G is a $n \times n$ - matrix A in which A_{ij} is the number of edges joining vertex i to vertex j. If G is undirected, A is a symmetric matrix. It is well known that the element $(A^p)_{ij}$ of A^p is the number of walks of length p from vertex i to vertex j. However, these walks are not necessarily simple, it is possible that the same vertex or the same edge appears more than once. Consider the edges of G to be labeled $e_1, \ldots, e_b$. Let Z be the $n \times n$ - matrix in which Z_{ij} is the formal sum of the edges joining vertex i to vertex j. Z is called the variable adjacency matrix of G. $(Z^p)_{ij}$ contains the sum of all walks of length p, but again these walks are not necessarily (simple) paths or (simple) cycles. Enumerating the cycles of G by simply generating the powers of Z is not very efficient because of these "non-simple" terms. It is more efficient to eliminate non-simple terms as soon as they occur in the computation of Z^p. The difference between the algorithms is the way in which they eliminate the walks that are not paths or cycles. Let D_1 be a $n \times n$ - diagonal matrix with $(D_1)_{ii} = Z_{ii}$ for $1 \le i \le n$, and let C_1 be defined by $C_1 = Z - D_1$. Then C_1 contains all paths of length 1 except the self-loops. Suppose C_q is a matrix with $(C_q)_{ij}$ is the sum of all paths of length q for $i \neq j$ and $(C_q)_{ii} = 0$ for

$1 \le i, j \le n$. Compute $C_q C_1$. Let P be a path of length q between the vertices v_i and v_k, and let there be an edge from v_k to v_j. If v_j is not in P, then the path of length $q+1$ from v_i to v_j is a term of $(C_q C_1)_{ij}$. If $v_i = v_j$, then the cycle of length $q+1$ containing v_i is a term of $(C_q C_1)_{ii}$. If v_j is in P and $v_i \ne v_j$, then we have neither a cycle nor a path. We call such a walk a *flower* ([40]). A flower from v_i to v_j consists of a path from v_i to v_j together with a cycle containing v_j, where the path and the cycle have no vertex in common except v_j. So if v_j is in P and $v_i \ne v_j$, then a flower from v_i to v_j is a term of $(C_q C_1)_{ij}$. Defining D_1 and C_1 as above, a basic algorithm to enumerate all cycles of a graph involves the following steps.

```
step 0:  $q = 0$
step 1:  $q = q + 1$; compute $C_q C_1$
step 2:  use the terms of the diagonal elements of $C_q C_1$ to enumerate all cycles of length $q+1$
step 3:  compute $C_{q+1}$ by setting all diagonal elements of $C_q C_1$ to zero and eliminating the flowers
step 4:  if $C_{q+1} = 0$, then the algorithm terminates, else goto step 1
```

Ponstein ([67]) and Yau ([99]) both use this basic algorithm to enumerate rooted dicircuits. Each cycle of length q is enumerated q times. To eliminate the flowers in step 3, Ponstein computes also the matrix $C_1 C_q$. He shows that a walk from vertex i to vertex j is a path if and only if it occurs as a term in both $(C_1 C_q)_{ij}$ and $(C_q C_1)_{ij}$. He computes C_{q+1} by taking $(C_{q+1})_{ij}$ as the sum of the terms that occur in both $(C_1 C_q)_{ij}$ and $(C_q C_1)_{ij}$ for $i \ne j$, and $(C_{q+1})_{ii} = 0$ $(1 \le i, j \le n)$. Yau eliminates flowers by testing whether any term in $(C_q C_1)_{ij}$ contains as a factor any cycle that has been enumerated so far and if so, reducing this term to zero. It should be noted that in [99] Yau computes all Hamiltonian cycles, so his algorithm terminates if C_n has been computed. In fact, our basic algorithm certainly terminates if $q = n-1$, because there are no cycles of length greater than n. Yau and Ponstein both generate edge-sequences to enumerate the cycles of G. Kamae ([40]) and Danielson ([20]) use almost the same basic algorithm, but they generate vertex-sequences. In the definition of D_1 and C_1 Kamae uses the adjacency matrix A instead of the variable adjacency matrix Z. In step 2 he uses a matrix A_{q+1} (instead of using the diagonal elements of $C_q C_1$) to enumerate all cycles of length $q+1$. A_{q+1} is defined by $(A_{q+1})_{ij} = (C_q)_{ji}(C_1)_{ij}$. Since $(C_q)_{ji}$ equals the number of paths from vertex j to vertex i and $(C_1)_{ij}$ equals the number of edges from i to j, $(A_{q+1})_{ij}$ equals the number of cycles of length $q+1$ containing an edge from vertex i to vertex j. Now suppose row i_1 is the first row of A_{q+1} which includes non-zero entries, and let i_2 be the first column of row i_1 which is non-zero. There exists at least one cycle of length $q+1$ containing edge $i_1 i_2$. Repeating this procedure with row i_2 we get a sequence $i_1 i_2 i_3$ which might be part of a cycle. Repeating this procedure again we finally get a sequence $i_1 i_2 \cdots i_{q+1}$. If the element $i_{q+1} i_1$ of A_{q+1} is non-zero, we have the vertex sequence of a cycle of length $q+1$. In any case we delete i_{q+1} from the sequence and look for the next non-zero entry in row i_q. Continuing this backtrack procedure we proceed until all cycles of length $q+1$ containing vertex i_1 are enumerated. Using the vertex sequences we can obtain the edge-sequences of the cycles. At last we obtain a new matrix A'_{q+1} by decreasing $(A_{q+1})_{ij}$ by 1 for each time edge ij occurs in one of the edge-sequences. To enumerate all cycles of length $q+1$, we repeat the whole procedure using A'_{q+1} instead of A_{q+1}, until we get an all zero matrix. In step 3 of the basic algorithm C_{q+1} is computed by setting all diagonal elements of $C_q C_1$ to zero and decreasing $(C_q C_1)_{ij}$ by one for each flower (with $q+1$ edges) from vertex i to vertex j. Danielson ([20]) defines the internal node product of a path P to be the product of the successive vertices of P but without the terminal vertices. For example, if $v_1 v_2 v_3 v_4 v_5$ is a path in G, then the corresponding internal node product is $v_2 v_3 v_4$. He also introduces another modification of the adjacency matrix by defining a $n \times n$ - matrix C_1 in which $(C_1)_{ij} = j$ if an edge connects i to j and $(C_1)_{ij} = 0$ otherwise. Danielson assumes the graph to be simple, i.e. without self-loops or parallel edges, but these restrictions are not necessary (see also the introduction of this chapter). As there are no self-loops, the main diagonal elements of A and C_1 are all zero, and as there are no parallel edges, the entries of A are all zero or one. The meaning of C_q in his version of the basic algorithm is a matrix in which $(C_q)_{ij}$ contains the sum of all internal node products of (simple) paths of length q from vertex i to vertex j. In step 0 Danielson sets q to 1 and computes $C_1 A$. He uses the terms of the diagonal elements to enumerate the cycles of length 2. Then he computes C_2 by setting the main diagonal elements of $C_1 A$ to zero. The rest of the algorithm is the same, except that he computes $C_1 C_q$ instead of $C_q C_1$. The order in which the matrices C_1 and C_q appear in the product is important now, because internal node products are computed. If there is a path of length q from vertex i to vertex k represented by internal node product $l_1 \cdots l_{q-1}$ and an edge connects vertex k to vertex j, then there exists a path of length $q+1$ from i to j with internal node product $l_1 \cdots l_{q-1} k$. Computing $C_q C_1$ however, yields the product $l_1 \cdots l_{q-1} j$ as a term of $(C_q C_1)_{ij}$, which is not an internal node product of a path of length $q+1$ from i to j. On the other hand, an edge from i to l_1 must exist just as a path of length q from l_1 to j represented by internal node product $l_2 \cdots l_{q-1} k$, so computing $C_1 C_q$ yields $l_1 l_2 \cdots l_{q-1} k$ as a term of $(C_1 C_q)_{ij}$. To eliminate flowers Danielson tests in step 3 of the basic algorithm whether the terms of the entries of row i contain vertex i. If so, the term is set to zero.

2.4 Algorithms using the line digraph

Let G be a directed graph without loops or parallel edges, and define the line digraph $L(G)$ of G as follows. The vertices of $L(G)$ correspond to the edges of G, so if $v_i v_j$ is an edge of G then v_{ij} is a vertex of $L(G)$. An edge from v_{ij} to v_{kl} exists in $L(G)$ if $j = k$, that means if the terminal vertex of the edge in G corresponding to v_{ij} is the same as the initial vertex of the edge corresponding to v_{kl}. From the definition of $L(G)$ we can see that there is a one-to-one correspondence between the cycles of length k of G and the cycles of length k of $L(G)$, for $2 \le k \le n$. Since G has no self-loops, each edge of $L(G)$ corresponds to a simple walk of length 2 in G, to a rooted 2-cycle if it lies on a cycle in $L(G)$ and to a 2-path otherwise. Enumerate all rooted cycles of length 2. Delete the edges from $L(G)$ that correspond to a rooted 2-cycle, and call the resulting graph $L_2(G)$. $L_2(G)$ has no cycle of length less than 3. Let $L_2^{(2)}(G)$ be the line digraph of $L_2(G)$. There is a one-to-one correspondence between the k-cycles of G and the k-cycles of $L_2^{(2)}(G)$ for $3 \le k \le n$. A simple walk of length 3 in G corresponds to an edge in $L_2^{(2)}(G)$. A non-simple walk of length 3 in G must contain a cycle of length

2, which corresponds to a cycle of length 2 in $L(G)$. Since we have deleted all 2-cycles from $L(G)$, an edge of $L_2^{(2)}(G)$ must correspond to a simple walk of length 3, to a rooted 3-cycle if it lies on a 3-cycle in $L_2^{(2)}(G)$ and to a 3-path otherwise. Enumerate all the rooted cycles of length 3, delete the edges that correspond to a rooted 3-cycle from $L_2^{(2)}(G)$ and call the resulting graph $L_3(G)$. $L_3(G)$ has no cycles of length less than 4. To obtain all rooted cycles of G we repeat the procedure until we get some null graph. This approach to the cycle enumeration problem is due to Cartwright and Gleason ([11]). Note that instead of searching for cycles in G, they search for cycles of length k in a graph that has no cycles of length less than k.

2.5 Hamiltonian cycles

In this section we describe some algorithms that enumerate Hamiltonian cycles. In section 2.3 we already discussed an algorithm of Yau ([99]) for enumerating all Hamiltonian cycles using the powers of the adjacency matrix, but this algorithm also generates all other cycles of the graph. Of course, each algorithm discussed in chapter 2 can be used to enumerate all Hamiltonian cycles by simply generating all cycles and testing which cycle is Hamiltonian. We now describe two algorithms that enumerate all Hamiltonian cycles only. Both are basically backtrack algorithms. Roberts and Flores ([75]) use an unrestricted backtrack procedure. Consider the vertices as numbered $1, \ldots, n$. Roberts and Flores use the combinatorial matrix M, with the entries $i_1, i_2, \cdots$ in column j representing the vertices at which an edge terminates that originates in vertex j. The columns of this matrix are recognized as a double linked adjacency list. Thus the j^{th} column of M can be seen as a list of vertices i for which an edge ji exists. Choose a start vertex s. As in section 2.2 a path is built, using the combinatorial matrix M. Let vertex j be the last vertex of the current path, and let i_1 be the first entry of column j. The path is extended with vertex i_1. This procedure is repeated until a vertex v is entered which is contained in the current path. If v equals s and the length of the current path equals n, then a Hamiltonian cycle is output. The algorithm proceeds with the next entry after v of the column corresponding to the last vertex of the current path. If the current path can not be extended with an entry of the column corresponding to the last vertex of the current path, then the algorithm backs up one vertex. This procedure is repeated until all possibilities are exhausted. As a Hamiltonian cycle contains each vertex of the graph, we can choose any vertex as the start vertex. This algorithm does not make use of any special property of Hamiltonian cycles, like for example the property that if a vertex v has only one edge wv entering, then that edge is required in the cycle. (When we arrive at vertex w we have to choose edge wv as the extension of the current path.) This and other properties have been used by Rubin ([76]) to reduce the number of possibilities that are considered. The basic idea of the algorithm is still that of building paths, starting with vertex s. Each time after the current path has been extended with a vertex, the edges of the graph are classified into three sets D, R and U. D contains "deleted edges", i.e. edges which can not be in any Hamiltonian path containing the current path. R contains "required edges", i.e. those edges which must be contained in every Hamiltonian path containing the current path. U contains the "undecided edges", edges which can not yet be classified. The classification of the edges as "deleted" or "required" is performed by applying a number of rules, which may be applied in any order and until no decision can be made about the remaining edges. During this classification two directed edges *vw* and *wv* are considered as an undirected edge. The rules are the following.

- If a vertex has only one directed edge entering (leaving), then that edge is required.
- If a vertex has only two edges incident, then both edges are required.
- If a vertex has a required directed edge entering (leaving), then all incident undirected edges are assigned the direction leaving (entering) that vertex.
- If a vertex has a required undirected edge incident and all other incident edges are leaving (entering) the vertex, then the required edge is assigned the direction entering (leaving) the vertex.
- If a vertex has two required edges incident, then all undecided edges incident may be deleted.
- If a vertex has a required directed edge entering (leaving), then all undecided directed edges entering (leaving) may be deleted.
- Delete any edge which forms a closed cycle with required edges, unless it completes the Hamiltonian cycle.

After the classification of the edges into the sets R, D and U a number of failure rules are verified to decide whether the current path possibly is contained in a Hamiltonian cycle. Those failure rules are the following.

- Fail if any vertex becomes isolated.
- Fail if any vertex has only one incident edge.
- Fail if any vertex has no directed edge entering (leaving).
- Fail if any vertex has two required edges entering (leaving).
- Fail if any vertex has three required edges incident.
- Fail if any set of required edges forms a closed cycle other than a Hamiltonian cycle.

If no failure rule applies, then the current path possibly is part of a Hamiltonian cycle. The successors of the last vertex of the path are listed and the path is extended with the first successor listed. If any of the failure rules applies, then the current path can not be part of a Hamiltonian cycle. In that case, and in case that all extensions of the last vertex have been examined, the last vertex of the current path is deleted and the path is extended with the next listed successor of the preceeding vertex. If the current path contains all vertices of the graph and a successor of the last vertex equals the start vertex, then a Hamiltonian cycle is enumerated, and the algorithm proceeds as if all extensions of the last vertex have been examined. The algorithm ends when all possible extensions of the start vertex have been explored.

2.6 Algorithms for enumerating cycles of fixed length

In this section we only consider algorithms for enumerating triangles or quadrangles. In [36] Itai and Rodeh give several methods for finding a triangle in a graph. Some can be used to enumerate all triangles. The method we discuss here uses rooted spanning trees. Let T be a rooted spanning tree of a connected graph G.

Lemma. *In G exists a triangle which contains an edge of T if and only if in G exists a non-tree edge (x, y) for which $(father(x), y) \in E(G)$.*

The algorithm to enumerate triangles consists of the following steps. Find a rooted spanning tree T of G. For each non-tree edge (x, y) check (in both directions) whether $(father(x), y) \in E(G)$. If so, a triangle $(x, y, father(x))$ is output. When all triangles containing an edge of T have been found, the edges of T are deleted from G. The resulting graph G' may be disconnected. For each connected component of G' we repeat the above procedure. When each component left has at most two edges, all triangles have been enumerated. However, duplications may arise. If $father(x) = father(y)$ and $(father(x), y) \in E$ then the algorithm outputs the triangles $(x, y, father(x))$ and $(y, x, father(y))$, which are the same. To avoid this problem we number the vertices $1, 2, \ldots, n$. When we find a triangle $(x, y, father(x))$, we test whether $father(x) = father(y)$. If $father(x) \neq father(y)$ then the triangle is output, otherwise the triangle is output only when $x < y$. Other duplications of the triangle $(x, y, father(x))$ must have the form $(x, father(x), y)$, $(father(x), x, y)$, $(y, father(x), x)$ or $(father(x), y, x)$. The first and the second possibility will not be checked (and thus will not be output) because the edge $(x, father(x))$ is a tree-edge. As $(x, y, father(x))$ is a triangle, (x, y) is a non-tree edge, so $father(y) \neq x$ and the third possibility will not occur. As $father(father(x)) \neq x$, the fourth possibility will not occur either. The complexity of the modified algorithm is the same as the complexity of the original algorithm of Itai and Rodeh. It requires $O(e^{\frac{3}{2}})$ time. In [18] Chiba and Nishizeki present another algorithm for listing all the triangles of a simple undirected graph. The vertices are considered as numbered $1, \ldots, n$ in such a way that $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$. Observing that each triangle containing vertex v_i corresponds to an edge joining two neighbours of v_i, they first mark all the vertices u adjacent to v_i (for the current index i). For each marked vertex u and each vertex w adjacent to u they test whether w is marked. If so, a triangle v_i, u, w is listed. After this test is completed for each marked vertex u, they delete v_i from G and repeat the procedure with v_{i+1}. Starting with v_1, this algorithm lists all triangles without duplication in $n-2$ steps. To compute the time complexity of this algorithm, Chiba and Nishizeki use the *arboricity* of G, defined as the minimum number of edge-disjoint spanning forests into which G can be decomposed, and denoted with $\alpha(G)$ ([32]).

Lemma. *If G has n vertices and e edges, then* $\sum_{uv \in E} \min\{d(u), d(v)\} \leq 2.\alpha(G).e$

Using this lemma they show that the time complexity of the algorithm is $O(\alpha(G).e)$. If G is planar then $\alpha(G) \leq 3$ ([32]) and $e \leq 3n-6$, so the algorithm runs in $O(n)$ time in case G is planar. In [18] Chiba and Nishizeki also present an algorithm for listing all quadrangles of a simple undirected graph. The quadrangles are not listed individually, but as triples $(v, w, \{u_1, u_2, \ldots, u_l\})$, $l \geq 2$, where $u_1, u_2, \ldots, u_l$ are all adjacent to v and w. A triple $(v, w, \{u_1, \ldots, u_l\})$ represents $\frac{1}{2}l(l-1)$ quadrangles (v, u_i, w, u_j) with $1 \leq i, j \leq l$ and $i \neq j$. As in the case of listing triangles, the vertices are numbered $1, \ldots, n$ in such a way that $d(v_1) \geq d(v_2) \geq \cdots \geq d(v_n)$. For each vertex w within distance 2 from the current vertex v_i the vertices $\{u_1, \ldots, u_l\}$ are stored in a list $U(w)$. Starting with $i = 1$, the algorithm adds vertex u to the list $U(w)$ for each u adjacent to v_i and each $w \neq v_i$ adjacent to u. For each vertex w with $|U(w)| \geq 2$ the triple $(v_i, w, U(w))$ is listed. To avoid duplicate quadrangles, v_i is deleted from G and the procedure is repeated with v_{i+1}. The algorithm ends after the n^{th} iteration. It requires $O(\alpha(G).e)$ time and $O(e)$ space.

3. Paths

In this chapter we discuss algorithms for enumerating all paths of a graph G or for enumerating special paths. The simplifications of G mentioned in chapter 2 can also be made in this case. Many of the algorithms for enumerating all cycles of a graph can be used for listing paths too. Among those algorithms are backtrack algorithms, algorithms using the powers of the adjacency matrix as well as the algorithm using the line digraph of G.

3.1 Backtrack algorithms

As we have seen in section 2.2, the algorithm of Tiernan ([87]) for enumerating all cycles, builds all elementary paths $v_1, v_2, \ldots, v_k$ with $v_1 \leq v_i$ for $2 \leq i \leq k$. So if we choose the smallest vertex (of a given numbering) as v_1, the algorithm enumerates all paths of G starting with v_1. Instead of choosing another vertex as the start vertex, (as does the algorithm of Tiernan), we first choose another numbering of the vertices, such that the smallest vertex of this numbering is different from v_1. Repeating this procedure until each vertex has been the smallest vertex of some numbering and thus start vertex, gives us all paths of G. The algorithms for enumerating all cycles of Tarjan ([86]), Johnson ([39]) and Szwarcfiter and Lauer ([81]) are all backtrack algorithms with some kind of restriction to the number of paths that are build, so they can not be used to enumerate the paths of a graph. The algorithm of Read and Tarjan ([74]) can be modified to list all paths from a set S of start vertices to some set F of finish vertices. First a backward search is made to eliminate useless startvertices. For each start vertex s the same procedure is repeated except that "s" must be replaced by "a vertex in F". The time bound of this modified version of the algorithm is still $O(n + e + e.p)$ where p is the number of paths to be enumerated. In [42] Kroft presents an unrestricted backtrack algorithm for enumerating all paths from start vertex s to terminal vartex t, which is similar to the algorithm of Roberts and Flores for the enumeration of all Hamiltonian cycles ([75]). For each vertex v, let $A(v)$ denote the adjacency list of v. As in section 2.2, the algorithm of Kroft builds elementary paths starting in s. The path which is build so far is stored on a stack. The algorithm consists of a recursive procedure. On entering the procedure, let v be the element at the top of the stack. The procedure searches for the first vertex w of $A(v)$ which is not on the stack already. (*) If such vertex w is found, then w is added to the stack. If $w = t$, then the stack represents a new path from s to t. The path is output and w is deleted from the stack. If $w \neq t$, then the procedure is called recursively. After returning from this recursive call, and in case $w = t$, the procedure searches for the next vertex of $A(v)$ after w which is not in the stack already. The procedure is repeated from (*) until no such vertex is found. At that moment, v is deleted from the stack and the procedure terminates. When the procedure is called for the first time, the stack contains only vertes s. The algorithm terminates when the stack is empty.

3.2 Algorithms using the powers of the adjacency matrix

From the algorithms described in section 2.3, the algorithms of Ponstein ([67]), Yau ([99]) and Danielson ([20]) can be used to enumerate all paths of any length. In the version of the basic algorithm of Ponstein and Yau, the matrix element $(C_q)_{ij}$ contains all paths of length q from vertex i to vertex j. In Danielson's version of the basic algorithm, the matrix element $(C_q)_{ij}$ contains all internal node products of paths of length q from vertex i to vertex j. The algorithm of Kamae ([40]) described in section 2.3 can not be used to enumerate the paths of a graph. In his version of the basic algorithm he uses the adjacency matrix instead of the variable adjacency matrix, and the matrix element $(C_q)_{ij}$ gives us only the number of paths of length q from vertex i to vertex j.

In [65] Paz already used the same modification C of the adjacency matrix as Danielson. He considers graphs without self-loops (or parallel edges), so $C_{ii} = 0$ for each i, $1 \le i \le n$. A term of an element C_{ij}^q represents a path P_{ij} of length q from vertex i to vertex j, and contains each vertex of P_{ij} except vertex i. Paz generates all paths of length q by computing C^q. If a term of C_{ij}^q has a vertex appearing more than once or if it contains vertex i, then the path represented by that term contains a cycle and the term is deleted from C_{ij}^q. Paz shows that if we are only interested in all paths of length q between two vertices i and j, it is not necessary to compute C^q. Define D_{ij}^0 to be an $1 \times n$ - matrix $(d_i)_{i=1,\ldots,n}$ with $d_i = i$, $d_j = j$ and $d_k = 0$ for all $k \neq i, j$. Define recusively $D_{ij}^q = D_{ij}^{q-1}.C$ for $q = 1, 2, \cdots$. In this case too, if a term of an entry of D_{ij}^q has a vertex appearing more than once, it is deleted. The i^{th} entry of D_{ij}^q contains all paths of length q from vertex j to vertex i, and the j^{th} entry of D_{ij}^q contains all paths of length q from vertex i to vertex j. A term representing a path now contains all the vertices of the path. Thus, to compute all paths of length q between two vertices i and j, we only have to multiply a $1 \times n$ - matrix with a $n \times n$ - matrix q times, instead of multiplying two $n \times n$ - matrices q times.

3.3 Algorithms using the line digraph

In section 2.4 we have seen that an edge of the graph $L_k^{(k)}(G)$ corresponds to a simple walk of length k in G, to a rooted cycle of length k if it lies on a cycle of length k in $L_k^{(k)}(G)$ and to a path of length k otherwise. So we can use the algorithm of Cartwright and Gleason ([11]) to enumerate all paths of G.

3.4 Algorithms using regular expressions or Gaussian elimination

In the theory of automata, regular expressions over an alphabet $\{s_0, s_1, \ldots, s_k\}$ are defined as follows ([34]).

(a) $\emptyset, \lambda, s_0, \ldots, s_k$ are regular expressions, where $\emptyset$ represents the empty set, λ represents the set whose only member is the empty word and $s_0, \ldots, s_k$ represent singleton sets containing one letter of the alphabet.

(b) If P and Q are regular expressions representing the sets p and q then $P + Q$, PQ and P^* are regular expressions, where $P + Q$ represents the set $p \cup q$, PQ represents $\{\sigma\tau \mid \sigma \in p, \tau \in q\}$ and P^* is the same set as $\lambda + P + P^2 + \cdots$.

(c) Every regular expression may be constructed from the expressions listed in (a) by a finite number of applications of the operations listed in (b).

In [79] Sloane gives a description of the McNaughton - Yamada algorithm ([55]), used for the enumeration of the walks of a graph G. The vertices of G are labeled $v_1, v_2, \ldots, v_n$ and the edges are labeled $e_1, e_2, \ldots, e_m$. Let α_{ij}^k denote the set of all walks from v_i to v_j which do not pass through any intermediate vertex v_p with $p > k$, for $0 \le k \le n$ and $1 \le i, j \le n$. Define α_{ij}^0 for all i, j $(1 \le i, j \le n)$ as follows. If $i \ne j$, then α_{ij}^0 is the (formal) sum of the edges joining v_i to v_j, and $\alpha_{ij}^0 = \varnothing$ if there is no edge from v_i to v_j. If $i = j$, then α_{ij}^0 is the sum of λ and the edges from v_i to itself. The α_{ij}^0 are regular expressions over the alphabet $\{e_1, e_2, \ldots, e_m\}$. Having defined α_{ij}^0 we can compute α_{ij}^k in terms of regular expressions for each $k = 1, 2, \ldots, n$, according to the following rules.

(1) If $k \ne i$ and $k \ne j$, then $\alpha_{ij}^k = \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1}(\alpha_{kk}^{k-1})^* \alpha_{kj}^{k-1}$.
(2) If $i \ne j$ and $k = i$, then $\alpha_{ij}^i = (\alpha_{ii}^{i-1})^* \alpha_{ij}^{i-1}$.
(3) If $i \ne j$ and $k = j$, then $\alpha_{ij}^j = \alpha_{ij}^{j-1}(\alpha_{jj}^{j-1})^*$.
(4) If $i = j = k$, then $\alpha_{ii}^i = (\alpha_{ii}^{i-1})^*$.

Note that this algorithm computes *each* walk from v_i to v_j, that means, an edge can occur any number of times in the same walk. To avoid this problem we have to leave out the term $(\alpha_{kk}^{k-1})^*$ in (1) and we have to replace (2), (3) and (4) by one rule.

(1)′ If $k \ne i$ and $k \ne j$, then $\alpha_{ij}^k = \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1}\alpha_{kj}^{k-1}$.
(2)′ If $k = i$ or $k = j$, then $\alpha_{ij}^k = \alpha_{ij}^{k-1}$.

This reduces the size of the set considerably, but α_{ij}^k still contains walks in which edges or vertices appear more than once. It is clear that simply comparing elements of α_{ik}^{k-1} with elements of α_{kj}^{k-1} to avoid non-simple walks during the computation of α_{ij}^k requires much time, so this approach does not yield an efficient algorithm. However, in [25] Fratta and Montanari use the same algorithm to enumerate all simple paths of a directed graph by Gaussian elimination. The paths are represented by the sequence of their internal vertices. Let S_{ij} be a set of (not necessarily all) simple paths from vertex i to vertex j, for each $i, j = 1, \ldots, n$, $i \ne j$. Define the simple multiplication $S_{ij}S_{mn}$ of two sets S_{ij} and S_{mn} to be the set of paths obtained by concatenating each path of S_{ij} with each path of S_{mn}, where non-simple paths are deleted, if $j = m$, and to be the empty set otherwise. Define a path algebra on the sets S_{ij} as follows. The addition of sets is defined as the union of sets (with $\varnothing$ as the zero element). The multiplication of sets is defined as the simple multiplication described above (with $\{\lambda\}$ as the unity element, where λ is the path of zero length). Define t_{ij} to be the set containing edge ij if an edge from i to j exists, and to be the empty set otherwise. Let T_{ij} be the set of all simple paths from i to j for $i, j = 1, \ldots, n$, $i \ne j$. Fratta and Montanari show that the sets T_{ij} are the unique solution of the following system of linear equations in the path algebra.

$$T_{ij} \;=\; \sum_{k=1,\,k\neq i,j}^{n} t_{ik}T_{kj} + t_{ij} \qquad (i,j = 1,\ldots,n;\, i \neq j) \qquad (*)$$

Let T_{ij}^{k+1} denote the set of all simple paths from i to j, which do not pass through any vertex greater than k. Then, using this notation, the algorithm consists of the following steps. Set $T_{ij}^{1} = t_{ij}$ for $i, j = 1, \ldots, n, i \neq j$. Compute T_{ij}^{k+1} for each $i, j, k = 1, \ldots, n$ according to the following rules.

(1) If $k \neq i \neq j \neq k$, then $T_{ij}^{k+1} = T_{ij}^{k} + T_{ik}^{k}T_{kj}^{k}$.
(2) If $k = i$ or $k = j$, but $i \neq j$, then $T_{ij}^{k+1} = T_{ij}^{k}$.

Finally, set $T_{ij} = T_{ij}^{n+1}$ for each $i, j = 1, \ldots, n, i \neq j$.

Theorem. *The above algorithm is the solution by Gaussian elimination of system (*).*

The complexity of the algorithm depends on the complexity of the computation of T_{ij}^{k+1}. For this purpose, Fratta and Montanari use binary trees. To each binary tree B an initial vertex i and a terminal vertex j is associated, as well as a number k and two sets V^+ and V^-, where $V^+ \cup V^-$ contains all vertices not smaller than k. B represents the set S_B of all paths from i to j passing through V^+ and possibly passing through some vertices not in V^-. A leave of B contains a set of paths from i to j passing through the same set of vertices. If $S_B = \emptyset$, then B consists of the root containing *NIL*. If $S_B \neq \emptyset$ and $k = 1$, then B consists of the root containing all paths in S_B, i.e. all paths from i to j passing through the vertices in V^+. If $S_B \neq \emptyset$ and $k \neq 1$, then B consists recursively of a root having a left and a right subtree. To both subtrees the same vertices i and j are associated, as well as the number $k-1$. For the left subtree, V^+ is the same as the set V^+ of B, and $k-1$ is added to V^-. For the right subtree, V^- is the same as the set V^- of B and $k-1$ is added to V^+. Thus the left subtree represents all paths from i to j passing through the vertices in V^+ and possibly passing through some vertices not in $V^- \cup \{k-1\}$. The right subtree represents all paths from i to j passing through the vertices in $V^+ \cup \{k-1\}$ and possibly passing through some vertices not in V^-. Note that if $V^+ = \emptyset$, then V^- contains all vertices l with $l \geq k$, and thus $S_B = T_{ij}^{k}$. The sum of two trees B_1 and B_2 is a tree B, defined recursively as follows. If B_1 and B_2 both consists of the root only, then B consists of a root, containing the union of the sets of B_1 and B_2 (or *NIL* if both B_1 and B_2 contain *NIL*). If B_1 (B_2) is *NIL* and B_2 (B_1) is not *NIL*, then B equals B_2 (B_1). Otherwise the left (right) subtree of B is the sum of the left (right) subtrees of B_1 and B_2. Let B_1 be a tree with corresponding vertices i and l, and number k. Let B_2 be a tree with corresponding vertices l and j, and number k. The product of B_1 and B_2 is a tree B defined recursively as follows. If B_1 and B_2 both consists of the root only, then B consists of a root containing the concatenation of the elements of the set represented by B_1 with the elements of the set represented by B_2 (or *NIL* if both B_1 and B_2 contain *NIL*). If $i \neq k$ and $j \neq k$, then the left subtree of B is the product of the left subtrees of B_1 and B_2, and the right subtree of B is the sum of the product of the left subtree of B_1 and the right subtree of B_2, and the product of the right subtree of B_1 and the left subtree of B_2. If

$i = k$ or $j = k$, then the left subtree of B is the product of the left subtrees of B_1 and B_2, and the right subtree of B is *NIL*. Having defined the sum and the product of trees, we now can describe the computation of T_{ij}^{k+1}. To represent T_{ij}^{1} we create a tree consisting of the root only, with the corresponding sets $V^+ = \emptyset$ and $V^- = V$. If an edge from i to j exists, the root contains an empty list (we represent a path by a list of its internal vertices). If no edge exists, the root contains *NIL*. To compute T_{ij}^{k+1}, a new tree is created, where the left subtree is the tree representing T_{ij}^{k} and the right subtree is the product of the trees representing T_{ik}^{k} and T_{kj}^{k}. Fratta and Montanari claim that the algorithm can be implemented to run in a time which is output bounded.

3.5 Hamiltonian paths

The algorithm of Roberts and Flores ([75]) decribed in section 2.5 can be used to generate all Hamiltonian paths of G by repeating the procedure with each vertex as the start vertex. Each path of length $n-1$ that is generated by the algorithm is a Hamiltonian path. But, as we have seen before, this way of searching spends much time in considering possibilities that do not lead to a Hamiltonian path. Another algorithm discussed in section 2.5 which can be used to enumerate all Hamiltonian paths of G is the algorithm of Rubin ([76]). Before applying the algorithm we first obtain a graph G' from the problem graph G by adding a new vertex to G and connecting this vertex to every other vertex by two edges of opposite direction. There is a one-to-one correspondence between the Hamiltonian cycles of G' and the Hamiltonian paths of G. Applying the algorithm of Rubin to G' gives us all Hamiltonian cycles of G' and thus all Hamiltonian paths of G. If we are only interested in the Hamiltonian paths of G between two specified vertices s and t, we change G into a graph G' by adding a vertex v and two edges tv and vs to G. Now there is a one-to-one correspondence between the Hamiltonian cycles of G' and the Hamiltonian paths of G from s to t, so applying the algorithm of Rubin gives us all the Hamiltonian paths of G from s to t.

4. Spanning trees

Most of the algorithms for the enumeration of all spanning trees of a graph G belong to one of the three following classes.

1) backtrack algorithms
2) algorithms based on the fusion of vertices
3) algorithms based on the exchange of edges

The three classes will be discussed in section 4.1, 4.2 and 4.3. Some of the algorithms based on the contraction of vertices or on the exchange of edges also use a backtracking procedure. In section 4.4 through 4.7 we will describe a few algorithms which do not belong to one of these classes: an algorithm using the fundamental cutset matrix, an algorithm using the incidence matrix, an algorithm considering subgraphs with $n-1$ edges and an algorithm combining trees of subgraphs. It should be noted that most of the algorithms enumerate spanning trees of undirected graphs. For this reason, throughout this chapter, we let G be an undirected graph, unless explicitly stated otherwise. Of course, when we discuss spanning trees, we consider connected graphs only.

4.1 Backtrack algorithms

The basic idea of the algorithms discussed in this section is to generate all spanning trees containing a given partial tree T, but not containing any edge of a set A of edges. (A partial tree is a subgraph that does not contain any cycles, but may be disconnected.) Initially, T is some partial tree T_0 and A is empty. Let e be an edge not in T and not in A that does not create a cycle in T. Add e to T and call this procedure recursively to generate all spanning trees containing $T \cup e$, but not containing A. After returning from this recursive call, delete e from T and call the procedure again recursively to generate all spanning trees containing T, but not containing $A \cup e$. After returning from this recursive call, the algorithm backs up one edge, that means, if f is the last edge that has been added to the partial tree (which resulted in T), delete f from T and continue with the enumeration of all the spanning trees containing $T-f$, but not containing $A \cup f$. The procedure terminates when all spanning trees containing the initial partial tree T_0 have been enumerated. The choice for T_0 depends on the algorithm. In [74] Read and Tarjan choose for T_0 the subgraph of G containing all the bridges of the graph. They keep two variables e and B local to the recursive procedure. On entering the procedure, e becomes an edge not in the partial tree and is added to the partial tree T. Then B is computed as the set of edges that are not in T and that join vertices already connected in T. So B contains the edges that form cycles with edges already in T. All edges in B are deleted from G. Now the recursive procedure is called to produce spanning trees containing T and e. The recursion terminates when the partial tree is a spanning tree. After returning from the recursive call, the edges of B are added to G and e is deleted from the partial tree and from G. Now B is computed as the set of bridges which are not yet in the partial tree. All edges in B are added to the partial tree. Then the recursive procedure is called again to generate spanning trees containing T but not containing e. The recursion terminates when the partial tree is a spanning tree. After returning from this second call, all edges in B are removed from the partial tree, e is added to the graph and the procedure terminates. Before calling the recursive procedure for the first time, the algorithm tests whether G is connected. If G is not connected, then G contains no spanning trees and the algorithm terminates. This algorithm requires $O(n + e + e.t)$ time, where t is the number of spanning trees, and it requires $O(n + e)$ space.

The same basic idea was used in [27] by Gabow and Myers to enumerate all spanning trees (arborescences) of a directed graph G, rooted in a vertex r. Let T be the partial tree build up so far. To extend T, Gabow and Myers choose edges in such a way that T grows depth-first. For this purpose they use a stack F. F contains all edges from vertices in T to vertices not in T. T is extended with the edge $e = zv$ at the top of the stack. The edge e is added to T and deleted from the stack F. Each edge vw with $w \notin T$ is placed on the stack and each edge wv on the stack with $w \in T$ is deleted from that stack to avoid the creation of cycles, but the locations of these edges in the stack are stored. This procedure is called recursively to generate all spanning trees containing $T \cup e$ as described above. The recursion terminates when the current partial tree is a spanning tree. After returning from the recursive call each edge vw with $w \notin T$ is deleted from the stack and the edges wv with $w \in T$ are restored in F using the old locations. F is now exactly the same as it was just after e was deleted from it. Then e is removed from T and from G and stored on a stack FF local to the recursive procedure for enumerating all spanning trees containing T. The procedure continues by extending T with the topmost edge f of the stack F, and is called recursively to enumerate all spanning trees containing $T \cup f$, but not containing e. This is repeated until the edge for extending T that was processed is a bridge of the (current) graph G. At this moment all spanning trees containing T have been enumerated. Each edge in FF is now deleted from FF, added to G and placed on the stack F. The algorithm backs up one edge as described above. Initially, T contains only vertex r, and the algorithm terminates when all spanning trees containing r have been enumerated. The running time of this algorithm is $O(n + e + e.t)$ and the space required is $O(n + e)$. This algorithm finds all spanning trees of a directed graph. Gabow and Myers show that if G is an undirected graph, the algorithm can also be used to enumerate all spanning trees of G by giving each edge both directions, and so making G directed. The root r can be chosen arbitrarily. The running time is only $O(n + e + n.t)$.

4.2 Algorithms based on the fusion of vertices

Let $b = v_1v_2$ be an edge of the graph G. The spanning trees of G can be classified into those which contain b and those which do not contain b. If b is a bridge, this last class is empty. We define G_1 to be the graph obtained from G by the fusion of v_1 and v_2 into one vertex v_b, and G_2 to be the graph obtained from G by deleting edge b. Each spanning tree of G containing edge b can be obtained as a spanning tree of G_1, in which vertex v_b is expanded to edge b. Each spanning tree of G not containing edge b can be obtained as a spanning tree of G_2. This procedure is repeated recursively to generate the spanning trees of G_1 and G_2.

In [56] Minty uses this basic idea in a rather informal description of an algorithm to enumerate all maximal forests of a (not necessarily connected) graph G. First all self-loops and all bridges are deleted from G. Call the resulting graph G'. The maximal forests of G' are generated by choosing an edge b (that cannot be a bridge, because G' has no bridges), obtaining the graphs G_1 and G_2, and repeating this procedure with G_1 and G_2. The recursive procedure terminates when only null graphs are left. After the maximal forests of G_1 and G_2 have been found, vertex v_b is eventually expanded to edge b as described above, and the bridges of G are added to each maximal forest of G', which gives us the maximal forests of G. This algorithm allows G to have parallel edges, but treats them separately. McIlroy ([54]) treats parallel edges together. The vertices are numbered $1, \ldots, n$. Let $S \subset V(G)$ and $i \notin S$, then the attachment set A_{Si} of vertex i with respect to S is the set of edges connecting i to elements of S. A_{ij} is the attachment set of the vertices i and j, i.e. the set of all edges between them. Let G_{ij} be the graph obtained by the deletion of the elements of A_{ij} from G and the fusion of the vertices i and j. Let $T(G)$ denote the set of all spanning trees of G. Then, following the basic idea, we have

$$T(G) = T(G_{ij}) \times A_{ij} \cup T(G - A_{ij}) .$$

To generate the spanning trees, McIlroy gives a recursive procedure with three parameters G, S and B. G is the current graph. S is the set of vertices that has been fused into one vertex. B contains the vertices that are adjacent to elements of S, but that are not in S itself. Initially, G is the problem graph, S contains only vertex 1 and B contains all neighbours of vertex 1. Let i_1 be the first element of B. The procedure computes A_{Si_1}, removes A_{Si_1} from G and removes vertex i_1 from B. Let G' denote the graph $G - A_{Si_1}$, let S' denote $S \cup \{i_1\}$ and let B' denote $B \cup$ {neighbours of i_1 that are not in S}. The procedure is called recursively with G', S' and B'. The recursive procedure terminates when the current set B is empty or the current set S contains each vertex of the original graph G. In the latter case a number of spanning trees is output as the Cartesian product of the attachment sets. After returning from the recursive call of the procedure, the algorithm continues with the next element of B. Note that the attachment set A_{Si_1} was removed from G, which avoids duplication of spanning trees. The algorithm repeats the procedure for each element of B. In [74] Read and Tarjan give a worst case example in which the algorithm of McIlroy has a running time exponential in e.

A better algorithm for the enumeration of all spanning trees based on the fusion of vertices was presented in [97] by Winter. The vertices of G are labeled as follows. Compute a spanning tree T of G. Choose a leaf of T, label it n and remove it from T. Then choose a leaf of the resulting tree T', label it $n-1$ and remove it from T'. Repeat this procedure until all vertices are labeled. A labeled graph is said to be *properly labeled* if there exists a spanning tree which can be used to generate the given labeling as described. Let n_i be a vertex adjacent to vertex n of the properly labeled graph G. Let G^{n_i} denote the graph obtained from G as follows. For each $j = n-1, n-2, \ldots, n_i$, all edges from j to n are removed from G. The remaining edges incident with n are made incident with n_i and vertex n is removed from G. G^{n_i} is said to be obtained from G by *proper contraction* of n into n_i. Let $T(G)$ be the set of all spanning trees of G. Let $A(n) = \{n_1, \ldots, n_t\}$ be the set of vertices adjacent to n. For each $k = 1, \ldots, t$, let P^{n_k} denote the set of spanning trees of G containing an edge from n to n_k, but not containing any edge from n to n_j, for $j = k+1, \ldots, t$. There exists a one-to-one correspondence between the elements of P^{n_k} and the spanning trees of G^{n_k}. The sets $P^{n_1}, P^{n_2}, \ldots, P^{n_t}$ form a partition of $T(G)$. Each P^{n_k} can be further partitioned using proper contractions of vertex $n-1$ into a vertex adjacent to $n-1$ in G^{n_k}. Proceeding in this manner finally gives us all spanning trees of G. Let G^{β_k} be the graph obtained from G by k consecutive proper contractions, i.e. β_k is a sequence of vertices $r_n r_{n-1} \cdots r_{n-k+1}$, where n was contracted into r_n to obtain G^{r_n} from G, $n-1$ was contracted into r_{n-1} to obtain $G^{r_n r_{n-1}}$ from G^{r_n}, etc.. G^{β_k} is a properly labeled multigraph with $n-k$ vertices. Let $E(j, i)$, for $j = 2, \ldots, n-k$ and $i = 1, \ldots, j-1$, denote the set of edges between j and i in G^{β_k}. $E(j, i)$ is arranged as a simple list. Let $EE(j)$, for $j = 2, \ldots, n-k$, denote the set of the $E(j, i)$, $i = 1, \ldots, j-1$, that are not empty. The algorithm consists of a recursive procedure. On entering the procedure, if $n-k \neq 2$, the vertex with the greatest label r_{n-k} adjacent to $n-k$ in G^{β_k} is selected. The graph $G^{\beta_k r_{n-k}}$ is obtained from G^{β_k} by a proper contraction of $n-k$ into r_{n-k} as follows. For each $i < r_{n-k}$ such that $E(n-k, i) \in EE(n-k)$, the set $E(r_{n-k}, i)$ is examined. If $E(r_{n-k}, i) \neq \emptyset$, then the position of the last element of $E(r_{n-k}, i)$ is stored on a stack and $E(n-k, i)$ is added at the end of $E(r_{n-k}, i)$. If $E(r_{n-k}, i) = \emptyset$, then $E(n-k, i)$ is adden to it and $E(r_{n-k}, i)$ is added to $EE(r_{n-k})$. k is increased by 1 and the procedure is called recursively. After returning from the recursive call, k is decreased by 1 and G^{β_k} is restored as follows. For each $i < r_{n-k}$ such that $E(n-k, i) \in EE(n-k)$, $E(r_{n-k}, i)$ is examined. If the first element of $E(r_{n-k}, i)$ equals the first element of $E(n-k, i)$, that means $E(r_{n-k}, i)$ was empty before $E(n-k, i)$ was added to it, then $E(r_{n-k}, i)$ is made empty again and is removed from $EE(r_{n-k})$. Otherwise, the elements of $E(n-k, i)$ are removed from $E(r_{n-k}, i)$, using the position which is stored on the stack. The top of the stack is deleted. Then the vertex with the next greatest label adjacent to $n-k$ is selected and the above procedure is repeated until all vertices adjacent to $n-k$ have been processed. At that moment the procedure terminates. If $n-k = 2$ on entering the procedure, then the elements of the Cartesian product $E(n, r_n) \times E(n-1, r_{n-1}) \times \cdots \times E(2, 1)$ are all spanning trees of G and are output. Before calling the procedure for the first time, the vertices of G are properly labeled, the sets $E(j, i)$ and $EE(j)$ are generated, and k is set to 0. The time complexity of this algorithm is $O(n + e + n.t)$, where t is the number of spanning trees, and the space complexity is $O(n^2)$.

4.3 Algorithms based on the exchange of edges

In [53] Mayeda and Seshu give an algorithm which, starting from a spanning tree T_0 (called the reference tree), enumerates all spanning trees by exchanging edges in T_0 with edges not in T_0. Consider the edges of G as numbered $1, \ldots, e$. For each spanning tree T and edge $e_i \in T$, let $S_i(T)$ denote the fundamental cut-set defined by e_i with respect to T. Let T_0 be the reference tree. Without loss of generality we may assume that the edges of T_0 are numbered $1, \ldots, n-1$. Let e_i be an edge of T_0. Replacing e_i by an edge of $S_i(T_0)$ gives another spanning tree. For $i = 1, \ldots, n-1$ define F^i to be the set of all spanning trees obtained by replacing e_i by elements of $S_i(T_0)$. The elements of the sets F^i, $1 \le i \le n-1$, form exactly the set of all spanning trees that differ from T_0 in one edge. Now let $T \in F^i$ and let e_j be an edge of T such that $1 \le j \le n-1$. So e_j is an edge of T and T_0. Replacing e_j by an element of $S_j(T)$ gives another spanning tree. Repeating this procedure for each edge $e_j \in T$, for every $T \in F^i$ and for each i, $1 \le i \le n-1$, gives us all spanning trees that differ from T_0 in two edges. However, generation of duplicate spanning trees is possible with this procedure. Some duplications can be avoided by replacing e_j in $T \in F^i$ by elements of $S_j(T) \cap S_j(T_0)$ only. We define $F^{i_1 i_2 \cdots i_k}$ recursively to be the set of all spanning trees that are obtained by replacing edge e_{i_k} by the elements of $S_{e_{i_k}}(T) \cap S_{e_{i_k}}(T_0)$ in each spanning tree $T \in F^{i_1 i_2 \cdots i_{k-1}}$, where $\{i_1, i_2, \ldots, i_k\}$ is a subset of $\{1, 2, \ldots, n-1\}$. The algorithm of Mayeda and Seshu consists of the following steps. Generate a spanning tree T_0 (and assume the edges of G to be numbered such that $T_0 = \{e_1, \ldots, e_{n-1}\}$) . Generate F^{i_1} for $i_1 = 1, \ldots, n-1$. Then, recursively, having obtained the sets $F^{i_1 i_2 \cdots i_{k-1}}$ where $1 \le i_1 < i_2 < \cdots < i_{k-1} \le n-1$, generate the sets $F^{i_1 i_2 \cdots i_k}$ for each i_k, $i_{k-1} < i_k \le n-1$. The algorithm terminates when no new sets can be obtained. Spanning trees in different sets are different, but within a set duplications may arise. Define an ordered set $\{e_{i_1}, e_{i_2}, \ldots, e_{i_k}\}$ of edges to be an *M-sequence* if for each r, $1 \le r \le k$, the subset $\{e_{i_1}, e_{i_2}, \ldots, e_{i_r}\}$ is a connected subgraph. Mayeda and Seshu prove the following theorem.

Theorem. *If $T_0 = \{e_1, e_2, \ldots, e_{n-1}\}$ is the reference tree in M-sequence form, then the spanning trees in the set $F^{i_1 i_2 \cdots i_k}$ where $i_1 < i_2 < \cdots < i_k$, generated from $F^{i_1 i_2 \cdots i_{k-1}}$ by the algorithm, are all distinct.*

Thus, started with a reference tree in M-sequence form, the algorithm of Mayeda and Seshu generates all spanning trees without duplication. In [74] Read and Tarjan remark that this algorithm can be implemented to run in $O(n + e + n.e.t)$ time and $O(n.e)$ space, where t is the number of spanning trees. Finally, Mayeda and Seshu remark that a similar procedure can be obtained by using fundamental circuits instead of fundamental cut-sets.

A completely different approach was chosen by Gabow in [26] to generate weighted spanning trees in order of increasing weight. For every edge e of a graph G, let $w(e)$ denote the weight of e. The weight of a subgraph of G is the sum of the weight of its edges. Let T be a spanning tree of G. A *$T-$exchange* is a pair of edges (e, f) with $e \in T$, $f \notin T$ and $T-e \cup f$ is a spanning tree of G. The weight of T-exchange (e, f) is $w(f) - w(e)$. Gabow proves the following theorem.

Theorem. *Let T be a minimum weight spanning tree of G and let e be an edge of T. Let (e, f) be a T-exchange having the smallest weight of all T-exchanges (e, f'). Then $T-e \cup f$ is a minimum weight spanning tree of the graph $G-e$.*

From the theorem it follows that if T is a minimum weight spanning tree and (e, f) is a minimum weight T-exchange, then $T-e \cup f$ is a spanning tree with the next smallest weight. This observation is the basis of the algorithm. The spanning trees are represented by a father array F, where a vertex v has been chosen as root vertex and for each vertex $w \neq v$, $F(w)$ is the father of w. The trees are numbered in order in which they are generated. The algorithm proceeds in stages. After stage $j-1$, the first $j-1$ trees $T_1, T_2, \ldots, T_{j-1}$ have been enumerated. The remaining trees have been partitioned into $j-1$ disjoint sets $P_i^{j-1} = \{T_k \mid k > j-1; e_1, e_2, \ldots, e_r \in T_k; e_{r+1}, \ldots, e_s \notin T_k\}$, $1 \leq i \leq j-1$, where r and s vary for each set P_i^{j-1}. Note that we have not labeled or numbered the edges, so with $e_1, e_2, \ldots, e_r$ we mean the edges that have been computed in the previous stages. Each set P_i^{j-1} is represented by a tuple $(w, X_i^{j-1}, F_i, IN, OUT)$, where w, X_i^{j-1}, F_i, IN and OUT have the following meaning. IN represents the set of edges that must be contained in all trees of P_i^{j-1}, so $IN = \{e_1, e_2, \ldots, e_r\}$. OUT represents the set of edges that must not be contained in any tree of P_i^{j-1}, so $OUT = \{e_{r+1}, \ldots, e_s\}$. F_i is the father array of T_i. For each edge $g \in T_i - IN$, the list X_i^{j-1} contains the smallest T_i-exchange (g, h) with $h \in G-OUT$. Finally, w is the weight of the smallest T_i-exchange in X_i^{j-1}. In stage j, the algorithm finds the index i of the set P_i^{j-1} with the smallest weight w. Then T_j is computed by replacing e by f in T_i, where (e, f) is an element of X_i^{j-1} having weight w. F_j is computed and T_j is output. The sets P_k^j, $1 \leq k \leq j$, are formed as follows.

(1) $P_k^j = P_k^{j-1}$ for each $k \neq i$, $1 \leq k \leq j-1$.
(2) $P_i^j = \{T_k \mid k > j; e_1, \ldots, e_r, e \in T_k; e_{r+1}, \ldots, e_s \notin T_k\}$.
(3) $P_j^j = \{T_k \mid k > j; e_1, \ldots, e_r \in T_k; e_{r+1}, \ldots, e_s, e \notin T_k\}$.

Or, written as tuples, $P_i^j = (w', X_i^j, F_i, IN \cup \{e\}, OUT)$, where X_i^j is computed by removing T_i-exchange (e, f) from X_i^{j-1}, and $P_j^j = (w', X_j^j, F_j, IN, OUT \cup \{e\})$, where X_j^j is computed by removing (e, f) from X_i^{j-1} and adding the smallest T_j-exchange (f, h) with $h \in G-OUT \cup \{e\}$ to X_i^{j-1}. In both cases w' is computed as the minimum weight of the elements of P_i^j and P_j^j respectively. The algorithm ends when each list X is empty. To start the algorithm, we generate a minimum weight spanning tree T_1 and we have only one set P_1^1, containing all spanning trees of G, except T_1, and represented by the tuple $(w, X_1^1, F_1, \varnothing, \varnothing)$. Of course, each bridge of G is contained in every spanning tree of G. If B is the set of all bridges of G, we can start with the tuple $(w, X_1^1, F_1, B, \varnothing)$ representing P_1^1. The algorithm runs in $O(t.e)$ time and $O(t + e)$ space, where t is the number of spanning trees of G. The above algorithm is a modification of another algorithm of Gabow, which is also presented in [26] and enumerates the K smallest weight trees in $O(K.e.\alpha(e, n) + e.\log e)$ time, where α is Tarjan's inverse of Ackermann's function and very slow-growing. In [41] Katoh, Ibaraki and Mine show that the time required for finding the K smallest weight trees can slightly be reduced. They present an algorithm which is similar to Gabow's algorithm, but requires only $O(K.e + \min(n^2, e.\log\log n))$ time.

4.4 An algorithm using the fundamental cut-set matrix

Let T_0 be a spanning tree of a graph G and let e_i be a branch of T_0. Let S_i be the fundamental cut-set defined by e_i with respect to T_0. Define the fundamental cut-set matrix with respect to T_0 to be a $(n-1) \times e$ - matrix Q_f. The rows of Q_f correspond to the branches of T_0 and the columns of Q_f correspond to the edges of G, where $(Q_f)_{ij} = 1$ if edge e_j is contained in the fundamental cut-set corresponding to e_i, and $(Q_f)_{ij} = 0$ otherwise. It is well known ([21]) that Q_f can be partitioned into two submatrices Q_t and I_{n-1}, where I_{n-1} is the identity matrix of order $n-1$. That is,

$$Q_f = \left[\, Q_t \mid I_{n-1} \, \right].$$

The rows of Q_t correspond to the branches of T_0 and the columns of Q_t correspond to the chords of G with respect to T_0. In [72] Rao and Murti use this matrix Q_t to enumerate all spanning trees of G. Let B be the set of branches of T_0 and let C be the set of chords of G. Let B_1 and B_2 be two disjoint sets of B such that $B_1 \cup B_2 = B$. Let C_2 be a subset of C such that $|C_2| = |B_2|$. Rao and Murti prove the following theorem.

Theorem. *The edges of the set $B_1 \cup C_2$ form a spanning tree of G if and only if the submatrix of Q_t formed by the rows corresponding to elements of B_2 and the columns corresponding to the elements of C_2, is nonsingular.*

Let the edges of G be numbered such that the edges $1, 2, \ldots, e-(n-1)$ correspond to the chords of G and the edges $e-n+2, \ldots, e$ correspond to the branches of T_0. To generate all spanning trees, Rao and Murti compute all possible combinations of $n-1$ edges of the graph G. Each combination is represented as an array M with e entries. The i^{th} entry of M is 1 if e_i is contained in the combination, and is 0 otherwise. The submatrix Q_m of Q_t corresponding to M is computed by taking the columns of Q_t corresponding to the nonzero entries of the first $e-(n-1)$ positions of M and by taking the rows of Q_t corresponding to the zero elements of the trailing $(n-1)$ positions of M. After computing Q_m they test whether combination M is a tree by testing if Q_m is nonsingular.

4.5 An algorithm using the incidence matrix

Let G be a graph without self-loops. Let the vertices of G be numbered $1, \ldots, n$ and let the edges of G be numbered $1, \ldots, e$. The *incidence matrix B* of G is a $n \times e$ - matrix with $B_{ij} = 1$ if vertex v_i is incident with edge e_j, and $B_{ij} = 0$ otherwise. Consider each row of B as a vector of an e-dimensional vector space over F_2, the field of integers modulo 2. It is well known ([21]) that the rows of B are not linearly independent. However, if G is connected, then any combination of $n-1$ rows of B is a linearly independent system. Thus, any $(n-1) \times e$ - matrix of B specifies G completely. Such a submatrix of B is called a *reduced incidence matrix* of G. Let G be connected and let B_f be a reduced incidence matrix. For $i = 1, \ldots, n-1$, let P_i denote the i^{th} row of B_f. For each $j = 1, \ldots, e$, let E_j

be an e-vector with a 1 at position j and a 0 at position i, $i \neq j$. $E_1, \ldots, E_e$ form a basis of the vector space. For each $i = 1, \ldots, n-1$, P_i can be written as $P_i = E_{i_1} + E_{i_2} + \cdots + E_{i_{n_i}}$. Define a $n_i \times e$ - matrix Ω_i in which row j corresponds to E_{i_j}. The *Cartesian product* Π_k of $\Omega_1, \ldots, \Omega_k$, for $k = 1, \ldots, n-1$, is defined as the set of vectors which are obtained as the sum of any possible combination of k vectors E_p, each one taken from a different set $\Omega_1, \ldots, \Omega_k$. Clearly, the number of elements of Π_k equals $n_1 n_2 \cdots n_k$. Each element X of Π_k represents a set of edges. The *norm* of X is the number of edges of the set it represents. Define the *normalized Cartesian product* Π_k^{norm} as the subset of Π_k consisting of the elements of Π_k which have norm k and which do not appear an even number of times in the set Π_k.

Theorem. *Π_{n-1}^{norm} is a set of vectors representing all possible spanning trees of G.*

From the theorem it follows that we can enumerate all spanning trees of G by computing Π_{n-1}^{norm}. This approach is due to Piekarski ([66]).

4.6 An algorithm considering subgraphs with n-1 edges

To enumerate all spanning trees of a graph G with n vertices, Char ([14]) gives a simple algorithm for generating systematically subgraphs with $n-1$ edges and testing which subgraph is a spanning tree. Consider the vertices as numbered $1, 2, \ldots, n$. Although G is undirected, we consider an edge as starting in v_i and terminating in v_j, or vice versa, depending upon whether it is written as ij or ji. Let T be a spanning tree and let v be a vertex of G. In T there exists a unique path from v_i to v for each vertex v_i. Vertex v is called a reference vertex. Choosing v_n as the reference vertex, a spanning tree can be represented by an array T with $n-1$ elements, where $T(i) = j$ if edge ij is contained in the path from v_i to v_n. On the other hand, let T be an array with $n-1$ elements having the property that $1 \leq T(i) \leq n$ and an edge from i to $T(i)$ exists for each i, $1 \leq i \leq n-1$. Char proves the following theorem.

Theorem. *T represents a spanning tree if and only if for each i, $1 \leq i \leq n-1$, either*
(a) $T(i) = v_n$.
(b) $T(i) > i$.
(c) There exists a path $i = l_1 l_2 \cdots l_k = v_n$ such that $T(l_j) = l_{j+1}$ for each j, $1 \leq j \leq k-1$.

An edge from i to $T(i)$ is said to pass the test for tree compatibility if a path $i = l_1 l_2 \cdots l_k = l$ exists from i to a vertex greater than i, such that $T(l_j) = l_{j+1}$ for each j, $1 \leq j \leq k-1$. To decide whether an array T as described above is a spanning tree, we scan T from left to right as long as the edges from i to $T(i)$ pass the test for tree compatibility. T is not a spanning tree if, for some i, the path starting in i that we are building reaches again vertex i. Now let array T represent a spanning tree. Change the last k columns of T and let T' be the result, still having the property as described above. Then we have the following lemma.

Lemma. *To decide whether T' is a spanning tree, it is sufficient to test the last k columns for tree compatibility.*

To generate the arrays T, Char first numbers the vertices as follows. The number n is given to a vertex of maximum degree. Let α be the set of vertices which are not connected to v_n. The number $n-1$ is given to a vertex not in α, but connected to a maximum number of vertices in α. Let β be the set of vertices out of α which are not connected to v_{n-1}. Then the number $n-2$ is given to a vertex not in β, but connected to a maximum number of vertices in β. Repeat this procedure until no residue is left and each vertex is numbered or connected to a numbered vertex. Number the remaining vertices in any way. For each vertex v_i, $1 \le i \le n-1$, a list B_i is used, containing the vertices that are adjacent to v_i. Those vertices are stored in cyclic order, starting with the greatest vertex adjacent to v_i. Initialize T by giving $T(i)$ the value of the first element of B_i for each i, $1 \le i \le n-1$. Because of the numbering chosen, $T(i) > i$ for each i, so from the theorem it follows that T is a tree. Commencing with $i = n-1$, we let $T(i)$ pass through the list B_i, choosing only the elements $B_i(j)$ such that the edge from i to $B_i(j)$ passes the test for tree compatibility. When we arrive again at the first element of B_i, $T(i-1)$ becomes the next vertex in B_{i-1} that is adjacent with vertex $i-1$ such that the corresponding edge passes the test for tree compatibility, and the rotating recommences with $i = n-1$. Each time when $T(i)$ is given a new value for some i, we have tested the edge involved for tree compatibility. By the lemma, we do not have to test the elements 1 through $i-1$ of T. At the same time, $T(j)$ equals the first element of B_j for each j, $i+1 \le j \le n-1$, and thus $T(j) > j$. So each time when $T(i)$ is given a new value for some i, we have found a new spanning tree. The algorithm terminates when each element $T(i)$ of T is again equal to the first element of B_i. This algorithm was analysed by Jayakumar and Thulasiraman ([37]). The complexity is $O(e + n + n.(t + t_0))$, where t is the number of spanning trees and t_0 is the number of subgraphs considered that did not correspond to a spanning tree. Jayakumar and Thulasiraman show that $t_0 \le n^2.t$, so the complexity of Char's algorithm is $O(e + n + n^3.t)$. It is interesting that, tested on a number of randomly generated graphs, they have found that Char's algorithm is superior to the algorithms of Minty (see section 4.2) and Gabow and Myers (see section 4.1), and that Char's algorithm becomes more and more efficient as the number of spanning trees increases.

4.7 An algorithm combining trees of subgraphs

In this section we describe a rather theoretical method for the enumeration of all spanning trees of a graph, due to Hakimi and Green ([31]). First we have to go through some definitions. Let H be a subgraph of G and let e be an edge of G. We define $\frac{\delta H}{\delta e}$ to be the graph $H-e$ if e is an edge of H, and 0 otherwise. If $H(G)$ is a set of subgraphs $H_1, H_2, \dots, H_r$, then $\frac{\delta H(G)}{\delta e}$ is the set of subgraphs $\frac{\delta H_i}{\delta e}$, $i = 1, \dots, r$. If $H(G)$ and $F(G)$ are sets of subgraphs of G, then the ring sum of $H(G)$ and

$F(G)$, $H(G) \oplus F(G)$, is defined as the set of subgraphs that are either in $H(G)$ or in $F(G)$, bot not in both. If F is a subgraph of G and $F = e_1 e_2 \cdots e_s$, then we define $\frac{\delta H(G)}{\delta F}$ to be the set $\frac{\delta H(G)}{\delta e_1} \oplus \frac{\delta H(G)}{\delta e_2} \oplus \cdots \oplus \frac{\delta H(G)}{\delta e_s}$. If $F_1, F_2, \ldots, F_s$ is a set of subgraphs of G, then we define $\frac{\delta^s H(G)}{\delta F_1 \delta F_2 \cdots \delta F_s}$ to be the set $\frac{\delta}{\delta F_1}[\frac{\delta^{s-1} H(G)}{\delta F_2 \cdots \delta F_s}]$. Let G_1 and G_2 be two subgraphs of G, having sets of vertices V_1 and V_2 respectively. Let $\{v_1, v_2, \ldots, v_q\} \subseteq V_1$ and $\{v'_1, v'_2, \ldots, v'_q\} \subseteq V_2$. Generate a new graph G^* by "superimposing" $\{v_1, \ldots, v_q\}$ upon $\{v'_1, \ldots, v'_q\}$ as follows. The set of vertices of G^* is the set $V_1 \cup V_2 - \{v'_1, \ldots, v'_q\}$. The set of edges of G^* consists of the edges of G_1, the edges of G_2 that are joining vertices of $V_2 - \{v'_1, \ldots, v'_q\}$, and if $e = vw$ is an edge of G_2 with $v = v'_i$ for some i, $1 \le i \le q$ (and $w = v'_j$ for some j, $1 \le j \le q$), then $e = v_i w$ $(v_i v_j)$ is an edge of G^*. Let $T(G)$ denote the set of all spanning trees of the graph G. Represent a spanning tree by a list of its edges. Let $P_{12}, P_{23}, \ldots, P_{q-1,q}$ be paths in G_1 between pairs of vertices $v_1 - v_2, v_2 - v_3, \ldots, v_{q-1} - v_q$ and let $P'_{12}, P'_{23}, \ldots, P'_{q-1,q}$ be paths in G_2 between pairs of vertices $v'_1 - v'_2, v'_2 - v'_3, \ldots, v'_{q-1} - v'_q$. Hakimi and Green prove the following theorem.

Theorem. $T(G^*) = \dfrac{\delta^{q-1}[T(G_1) \times T(G_2)]}{\delta(P_{12} \cup P'_{12})\delta(P_{23} \cup P'_{23})\delta(P_{q-1,q} \cup P'_{q-1,q})}$.

This theorem can be used to compute the spanning trees of a graph G as follows. Generate two subgraphs G_1 and G_2 of G, such that G can be obtained by superimposing a set of vertices $\{v_1, \ldots, v_q\}$ upon the same set of vertices of G_2. Compute the spanning trees of G_1 and G_2, and compute $P_{12}, \ldots, P_{q-1,q}$ and $P'_{12}, \ldots, P'_{q-1,q}$. Generate all spanning trees of G by applying the theorem. To compute the spanning trees of G_1 and G_2 we can use the same procedure or another algorithm for enumerating spanning trees. In general, the number of spanning trees of G_1 and G_2 will be much smaller than the number of spanning trees of G. Still we have to store all the spanning trees of at least one of the subgraphs G_1 and G_2, and this may require a lot of space.

4.8 An algorithm for enumerating the subtrees of a tree

In [77] Ruskey presents an algorithm to enumerate all the subtrees of a tree T with root r, lexicographically with respect to a given ordering of the vertices of T. Define a heap-labeling of a tree T with root r to be a numbering of its vertices with the integers $\{1, \ldots, n\}$ such that children receive larger numbers than their parents. Let T be a heap-labeled tree with root r. T is represented by an array *par*, with *par*(i) is the parent of vertex i and *par*$(r) = 0$. The subtrees are represented by an array *sub*, with *sub*$(i) = 1$ if vertex i is in the subtree and *sub*$(i) = 0$ otherwise. We generate a subtree T_k by scanning the array *sub* of the subtree T_{k-1} that lexicographically precedes T_k, and so is the last subtree enumerated. We scan the array *sub* of T_{k-1} from right to left. Each 1 encountered is changed into a 0. If we encounter a 0 in position l and if the parent of l is in T_{k-1} (i.e. *sub*(*par*(l)) $= 1$), then we change *sub*(l) into a 1. The modified array *sub* now represents T_k. If the parent of l is not in T_{k-1}, we continue scanning *sub*. The algorithm starts with the subtree T_0, containing the root only. As T is heap-labeled, the array *sub* of T_0 contains a 1 in position 1, and a 0 in the positions k with $2 \leq k \leq n$. The algorithm terminates when *sub* contains all 1's. It is possible to speed up the algorithm by making use of information of the previous iteration. Suppose l is the position of *sub* that was changed into a 1 and caused us to enumerate T_{k-1}. To generate T_k we might start scanning *sub* in a position $L(l) \neq n$. Let $L(l)$ be the greatest j such that $par(j) = m$ for some $m \leq l$. As T is heap-labeled, $par(l) < l$, so $L(l) \geq l$. Let $i > L(l)$, then $i > l$ and $par(i) > l$. Since we know that the positions $l+1, \ldots, n$ of *sub* are all 0, it follows that $sub(i) = 0$ and $sub(par(i)) = 0$. So we do not have to consider the positions i with $i > L(l)$, and we start scanning *sub* in position $L(l)$. Note that the values $L(l)$ for each l, $1 \leq l \leq n$ can be computed before starting the generation of the subtrees. Better values for $L(l)$ might be computed, depending on the ordering of the vertices of T.

5. Cliques and complete subgraphs

In this chapter we describe algorithms for enumerating all cliques of a graph and algorithms for enumerating all complete subgraphs of order l, for fixed l. The algorithms for enumerating all cliques of a graph can be divided into the six following classes.

1) algorithms intersecting complete subgraphs with adjacency lists
2) algorithms adding vertices to complete subgraphs
3) an algorithm based on edge addition
4) an algorithm based on edge removal
5) an algorithm testing combinations of vertices
6) an algorithm using the adjacency matrix

The six classes will be discussed in the sections 5.1 through 5.6. In section 5.7 we describe algorithms for listing all complete subgraphs of fixed order. The problem of enumerating all cliques of a graph is equivalent to the problem of enumerating all maximal independent sets of a graph. For a discussion see chapter 6.

5.1 Algorithms intersecting complete subgraphs with adjacency lists

The first algorithm we describe in this section is an algorithm developed by Bierstone ([7]), revised by Augustson and Minker ([4]) and further by Mulligan and Corneil ([58]). The vertices of the graph are numbered $1, 2, \ldots, n$. To represent the graph, Bierstone uses a list M_j for each vertex v_j, $1 \le j \le n$. M_j contains all vertices v_k adjacent to v_j with $k > j$. A clique is represented by the set of its vertices. The algorithm builds sets of vertices representing complete subgraphs of G. These sets of vertices are stored in an array C. Upon termination of the algorithm, all elements of C will represent cliques. The algorithm proceeds in stages. After completion of each stage, no element of C is contained in any other element. To initialize, the algorithm finds the greatest index j such that M_j is not empty. For each $v_k \in M_j$, $k = n, \ldots, j+1$, the set $\{v_k, v_j\}$ representing a complete subgraph of order 2, is added to array C. At each stage the next greatest index j such that M_j is not empty, is found. Let l be the number of complete subgraphs in C after the previous stage. The elements of C are scanned in increasing order. During the scanning of the elements of C, a set W keeps track of all vertices of M_j that have not yet been put into some element of C. So, before starting the processing with $C(1)$, W equals M_j. The processing of the elements $C(k)$, $k = 1, \ldots, l$, consists of the following steps. Compute $T = C(k) \cap M_j$. If T contains less than two vertices, the algorithm continues with the next element of C. If T contains two or more vertices, then all elements of $T \cap W$ are deleted from W. $T \cup \{v_j\}$ is a complete subgraph. We have four possibilities.

(1) $T = C(k) = M_j$. In this case, redefine $C(k) = C(k) \cup \{v_j\}$. Delete from C those elements $C(q)$ with $q > l$ and $C(q) \subset C(k)$ (if any). We do not have to check the elements $C(q)$ with $q \le l$, because after the previous stage no element of C was contained in any other element of

C, and at the current stage no element $C(q)$ with $q = k+1, \ldots, l$ is redefined before the processing of $C(k)$ is finished. As $T = M_j$, all complete subgraphs $(C(m) \cap M_j) \cup \{v_j\}$ for $k < m \leq l$ will be subgraphs of $T \cup \{v_j\}$, so the algorithm continues with the next stage.

(2) $T = C(k) \neq M_j$. This case is processed in the same way as (1), except that the algorithm continues with the next element of C.

(3) $T \neq C(k)$ and $T = M_j$. Add $T \cup \{v_j\}$ to C and delete all elements $C(q)$ with $q > l$ and $C(q) \subset T \cup \{v_j\}$ from C. As $T = M_j$, the algorithm continues with the next stage (see (1)).

(4) $T \neq C(k)$ and $T \neq M_j$. If any element of C contains $T \cup \{v_j\}$, then the algorithm continues scanning the array C, otherwise $T \cup \{v_j\}$ is added to C and all $C(q)$ with $q > l$ and $C(q) \subset T \cup \{v_j\}$ are deleted from C. The algorithm continues with the next element of C.

If the algorithm has scanned the first l elements of C, then for each vertex $v_k \in W$ (if any) the set $\{v_k, v_j\}$ is added to C and the algorithm proceeds with the next stage. If each j with M_j not empty has been processed, then C contains all cliques of G and the algorithm terminates.

Another algorithm based on the intersection of complete subgraphs with adjacency lists was presented in [18] by Chiba and Nishizeki. This algorithm consists of a recursive procedure. Consider the vertices of the graph to be numbered $1, \ldots, n$, such that $d(v_1) \leq d(v_2) \leq \cdots \leq d(v_n)$. Let $A(i)$ denote the adjacency list of v_i and let G_i denote the subgraph of G induced by $v_1, v_2, \ldots, v_i$. The recursive procedure has two parameters C and i. C is a set of vertices representing a clique of G_{i-1}. When the procedure is called for the first time, $C = \{v_1\}$ and $i = 2$. On entering the procedure, if $i = n+1$, then C is a clique of G. C is output and the recursive procedure terminates. If $i \neq n+1$, then the algorithm starts searching for cliques of G_i. If $C \cap A(i) \neq C$, then C contains vertices that are not adjacent to v_i. In that case, C is also a clique of G_i and the procedure is called recursively with C and $i+1$. After returning from this call, and in case $C \cap A(i) = C$ (i.e. v_i is adjacent to each vertex of C), the procedure continues with computing $C' = (C \cap A(i)) \cup \{i\}$. C' is a complete subgraph. If G_i has a vertex $v_j \in A(i)$, $v_j \notin C$, such that $j < i$ and $C \cap A(i) \subset A(j)$, then v_j is adjacent to all vertices of $C \cap A(i)$ and to i, so C' is not a maximal complete subgraph of G_i. After computing C', the procedure tests whether C' is maximal. If C' is not a clique of G_i, the procedure terminates. If C' is a clique of G_i, then, to avoid duplicates, the algorithm tests whether C is the lexicographically largest clique of G_{i-1} containing $C \cap A(i)$.

Lemma. *C is the lexicographically largest clique of G_{i-1} containing $C \cap A(i)$ if and only if there is no vertex $v_j \in G_{i-1}$, $v_j \notin C$, such that $(C \cap A(i)) \cup C_j \subset A(j)$, where $C_j = \{v_k \in C \mid k > j\}$.*

If C is the lexicographically largest clique of G_{i-1} containing $C \cap A(i)$, then the procedure is called recursively with C' and $i+1$. After returning from this recursive call, and in case C is not the lexicographically largest clique of G_{i-1} containing $C \cap A(i)$, the recursive procedure terminates. Chiba and Nishizeki show that this algorithm can be implemented to run in $O(\alpha(G).e)$ time, where $\alpha(G)$ is the arboricity of G (see section 2.6). It requires $O(e)$ space. This algorithm is similar to an algorithm of Tsukiyama et al. ([89]) for listing all maximal independent sets of a graph (see section 6.1).

5.2 Algorithms adding vertices to complete subgraphs

In [4] Augustson and Minker give a description of an algorithm of Bonner ([8]) to enumerate all cliques of a graph G. The algorithm builds a set of vertices C representing a complete subgraph by adding one selected vertex at the time. When no vertices can be added and the complete subgraph represented by C is not contained in another complete subgraph of G, then C is a clique. Let the vertices of G be numbered $1, 2, \ldots, n$ and let $A(v)$ denote the set of neighbours of vertex v. The algorithm uses a variable i, indicating the current level of the algorithm, and three arrays C, $CAND$ and L with the following meaning. At level i, for each k, $1 \le k \le i$, $C(k)$ contains a set of vertices representing a complete subgraph, $CAND(k)$ contains a set of vertices that are candidates to extend $C(k)$, and $L(k)$ contains the vertex to be considered for addition to $C(k)$. Initially, $i = 1$, $C(1) = \varnothing$, $CAND(1) = V(G)$ and $L(1) = 1$. At level i, the algorithm tests whether $L(i)$ is contained in $CAND(i)$. If so, then $CAND(i+1)$ is set to $CAND(i) \cap A(L(i)) - \{L(i)\}$, $C(i+1)$ is set to $C(i) \cup \{L(i)\}$ and $L(i+1)$ is set to $L(i) + 1$. If $L(i)$ is not contained in $CAND(i)$, then $L(i)$ is set to $L(i) + 1$. In both cases, the algorithm tests whether any element of $CAND(i)$ is larger than $L(i)$. If such an element is found, then the complete subgraph $C(i)$ may not yet be maximal. In that case, the algorithm proceeds with level k, where $k = i$ if $L(i)$ was not contained in $CAND(i)$, and $k = i + 1$ otherwise. If no element of $CAND(i)$ is larger than $L(i)$, then $C(i)$ can not be extended. A variable T is set to $C(i)$. If $CAND(i)$ is empty, then $C(i)$ is a clique and is output. If $CAND(i)$ is not empty, then either $C(i)$ has been enumerated before or $C(i)$ is not maximal. Let W_k denote the set of all vertices of $CAND(k)$ with numbers greater than $L(k)$. In both cases the algorithm continues with searching the greatest index $k \le i-1$ such that W_k is not a subset of T. If such k exists, then $L(k)$ is set to $L(k) + 1$ and the algorithm proceeds with level k. If no such k exists, then all cliques have been enumerated and the algorithm terminates. In [9] Bron and Kerbosch give a backtrack algorithm which is similar to Bonner's algorithm. They also build a set of vertices C representing a complete subgraph by adding one selected vertex at the time. When no vertices can be added and the complete subgraph represented by C is not contained in another complete subgraph of G, the clique C is output. The vertex v that was the last added, is deleted from C and the algorithm continues with trying to extend C with vertices other than v. This backtrack algorithm consists of a recursive procedure with two parameters NOT and $CAND$. NOT is the set of vertices that are not to be chosen as an extension of C and $CAND$ is the set of vertices that are eligible ("candidates") to extend C. When the procedure is called for the first time, NOT is empty and $CAND$ contains all vertices of the graph. Initially the set C is empty too. For vertex v let $count(v)$ be the number of vertices in $CAND$ that are not adjacent to v. On entering the recursive procedure count(v) is computed for each $v \in NOT \cup CAND$. Let $v_j \in NOT \cup CAND$ be the vertex with the minimum count computed. If $v_j \in CAND$, then C is extended with vertex $s = v_j$. If $v_j \in NOT$, then C is extended with a vertex $s \in CAND$ with s not adjacent to v_j. (*) The new sets NOT' and $CAND'$ are computed. NOT' contains those vertices of NOT that are adjacent to s. $CAND'$ contains those vertices of $CAND - \{s\}$ that are adjacent to s. If NOT' and $CAND'$ both are empty, then C is a clique and is output. If $CAND'$ is not empty, then the recursive procedure is called with NOT' and $CAND'$. After returning from the recursive call, and in case the procedure was not called recursively at all, vertex s is deleted from C and from $CAND$, and is added to NOT. Then another vertex $s \in CAND$ with s not adjacent to v_j (if any) is selected. C is extended with s and the procedure repeats from (*) with the current sets NOT, $CAND$ and C. The recursive procedure terminates when all vertices in the current set $CAND$ are adjacent to v_j. When the algorithm returns from the initial call of the procedure, all cliques of G have been enumerated.

A completely different approach is the following. Let v be a vertex of the graph G. Define the sets C_v and D_v of vertices as follows. C_v is the set containing all neighbours of v and D_v is the set of vertices that are not adjacent to v. Each clique C of G contains either vertex v or at least one element of D_v, so the cliques of G can be classified into those that contain vertex v and those that contain at least one element of D_v. This idea was used by Akkoyunlu ([1]) to enumerate all cliques of G. The graph is represented by the sets C_v and D_v for each vertex $v \in V$. For $S \subseteq V$, let $L(S)$ denote the set of all cliques which contain at least one element of S and let $E(S)$ denote the set of all cliques which contain each element of S. Clearly, $L(V)$ represents the set of all cliques of G. The algorithm divides sets of cliques into smaller subsets, using the following rules.

(1) $L(\{v\}) = E(\{v\})$,

(2) $E(S_1) \cap E(S_2) = E(S_1 \cup S_2)$,

(3) $L(S_1) \cap L(S_2) = L(S_1)$, if $S_1 \subseteq S_2$,

(4) $L(\emptyset) = \emptyset$,

(5) $E(\{v\}) \cap (\bigcap_i L(S_i)) = E(\{v\}) \cap (\bigcap_{i;v \notin S_i} L(S_i \cap C_v))$,

(6) $L(S \cup \{v\}) = E(\{v\}) \cup (L(S) \cap L(D_v))$,

(7) $E(S) = \begin{cases} S & \text{if } \bigcap_{v \in S} C_v \neq \emptyset, \\ E(S) \cap L(\bigcap_{v \in S} C_v) & \text{otherwise.} \end{cases}$

The subsets of cliques which still have to be processed are stored in a stack. Elements of the stack are expressions representing intersections between one or more L-sets and at most one E-set. Initially the stack contains only $L(V)$. The algorithm proceeds in stages. At each stage, the topmost element T is deleted from the stack. T is either of the form $\bigcap_{i \in I} L(S_i)$ or the form $E(S') \cap (\bigcap_{i \in I} L(S_i))$. In the first case, a variable W is set to $\emptyset$ and in the second case W is set to S'. The index $k \in I$ such that S_k has the fewest elements is found and a vertex $v \in S_k$ is selected. Let S denote $S_k - \{v\}$. Using the above rules, the set $\bigcap_{i \in I} L(S_i)$ can be written as follows.

$$\bigcap_{i\in I} L(S_i) = ((\bigcap_{i\in I;i\neq k} L(S_i)) \cap E(\{v\})) \cup ((\bigcap_{i\in I;i\neq k} L(S_i)) \cap L(S) \cap L(Q)) ,$$

where $Q = D_v$ if $W = \varnothing$ and $Q = D_v \cap (\bigcap_{w\in W} C_w)$ otherwise. If $S = \varnothing$ or $Q = \varnothing$, then the second part of the right-hand side is an empty set. Is $S \neq \varnothing$ and $Q \neq \varnothing$, then T with $L(S) \cap L(Q)$ substituted for $L(S_k)$ is placed in the stack. In either case the algorithm continues processing $E(S') \cap ((\bigcap_{i\in I;i\neq k} L(S_i)) \cap E(\{v\}))$ or just $(\bigcap_{i\in I;i\neq k} L(S_i)) \cap E(\{v\})$, depending on the form of T. Let J be the set $\{j \mid j\in I,\ v_j \notin S_j\}$. If $J \neq \varnothing$, then for each $j\in W$ the set $W_j = S_j \cap C_v$ is computed. Using rule (5) and (2) we get the expression $E(W \cup \{v\}) \cap (\bigcap_{j\in J} L(W_j))$. If $L(W_j) = \varnothing$ for any j, then this expression represents an empty set, otherwise it is placed on the stack. In both cases the algorithm proceeds with the next stage. If $J = \varnothing$, then we have the expression $E(W \cup \{v\})$. Compute the set $P = \bigcap_{y\in W\cup\{v\}} C_y$. If $P = \varnothing$, then $W \cup \{v\}$ is clique and is output. If $P \neq \varnothing$, then $W \cup \{v\}$ is not yet maximal and the expression $E(W \cup \{v\}) \cap L(P)$ is placed on the stack. In either case the algorithm continues with the next stage. The algorithm terminates when the stack is empty.

5.3 An algorithm based on edge addition

Let $G = (V,E)$ be a graph with a pair of non-adjacent vertices x and y, and let $G' = (V,E')$ a graph with $E' = E \cup \{xy\}$. In [62] Osteen gives an algorithm that computes the cliques of G', given the cliques of G. Let L and L' denote the set of all cliques of G and G' respectively. As $xy \notin E$, the cliques of G can be partitioned into three disjoint sets X, Y, and L_0, where X is the set of all cliques containing x but not y, Y is the set of all cliques containing y but not x, and L_0 is the set of all cliques containing neither x nor y. Let C_x be a clique of G containing x. If each vertex of $C_x - \{x\}$ is adjacent to y, then $C_x - \{x\}$ is contained in some clique of G containing y, and $C_x \cup \{y\} \in L'$. Otherwise $C_x \cup \{y\}$ is not a complete subgraph of G' and $C_x \in L'$. This observation gives us the following partition of the sets X and Y. Define X_1 to be the set $\{C_x \in X \mid \text{for some } C_y \in Y,\ C_x - C_y = \{x\}\}$, and $X_2 = X - X_1$. Define Y_1 to be the set $\{C_y \in Y \mid \text{for some } C_x \in X,\ C_y - C_x = \{y\}\}$, and $Y_2 = Y - Y_1$. Finally, define L_1 as $\{C_x \cup \{y\} \mid C_x \in X_1\} \cup \{C_y \cup \{x\} \mid C_y \in Y_1\}$ and L_2 as $X_2 \cup Y_2$.

Lemma. $L_0 \cup L_1 \cup L_2 \subset L'$.

Note that the sets L_0, L_1 and L_2 are mutual disjoint. If $C \in L'$ and $C \notin L_0 \cup L_1 \cup L_2$, then $C \in L_3'$, the set of all maximal elements of $\{(C_x \cap C_y) \cup \{x,y\} \mid C_x \in X_2,\ C_y \in Y_2\}$. As no element of L_0 and L_2 contains both x and y, the sets L_0, L_2 and L_3' are mutual disjoint and no element of L_3' is contained in an element of L_0 or L_2. However, it is possible that an element of L_3' is contained in an element of L_1. Therefore, define L_3 to be the set of elements of L_3' which are not contained in any element of L_1.

Theorem. $L' = L_0 \cup L_1 \cup L_2 \cup L_3$.

This theorem is used to compute the set L' of all cliques of $G' = (V, E')$, given the set L of all cliques of $G = (V, E)$, where $E \subset E'$. Initialize $L' = L$ and $D = E - E'$. Choose an edge $xy \in D$ and remove xy from D. Compute X, Y and L_0 by placing each $C \in L'$ in X if $x \in C$, in Y if $y \in C$, and in L_0 otherwise. Compute X_1 and Y_1 as follows. For each $C_x \in X$ and $C_y \in Y$, if $C_x - \{x\} \subset C_x \cap C_y$ then place C_x in X_1 and if $C_y - \{y\} \subset C_x \cap C_y$ then place C_y in Y_1. Initialize four sets L_1', L_1'', X_2 and Y_2 to be empty. For each $C_x \in X$, if $C_x \in X_1$ then $C_x \cup \{y\}$ is placed in L_1', otherwise C_x is placed in X_2. For each $C_y \in Y$, if $C_y \in Y_1$ and C_y is not contained in a member of L_1', then $C_y \cup \{y\}$ is placed in L_1'', otherwise C_y is placed in Y_2. The set L_1 is the union of L_1' and L_1''. For each $C_x \in X_2$ and each $C_y \in Y_2$, place $(C_x \cap C_y) \cup \{x, y\}$ in a set A. Remove from A each element that is contained in any other element of A. Remove from A each element that is contained in any element of $L_1' \cup L_1''$. Set $L' = L_0 \cup L_1' \cup L_1'' \cup X_2 \cup Y_2 \cup A$. Then choose another edge in D (if any) and repeat the procedure until no edge in D is left. At that moment, L' contains all cliques of G' and the algorithm terminates. An upperbound for the time complexity of the algorithm when only one edge is added, is approximately $\frac{1}{2}(\frac{r}{2})^4$, where r is the number of cliques of G. A lower bound on the time complexity is approximately r.

5.4 An algorithm based on edge removal

In the preceding section we discussed an algorithm of Osteen for enumerating all cliques of a graph, based on edge addition. In the same reference Osteen presents a similar algorithm, based on edge removal. Let $G = (V, E)$ be a graph with a pair of non-adjacent vertices x and y, and let $G' = (V, E')$ be a graph with $E' = E \cup \{xy\}$. Osteen computes the set L of all cliques of G, given the set L' of all cliques of G'. L' can be partitioned into four sets X, Y, L_0 and W, the sets of all cliques of G' containing x but not y, y but not x, neither x nor y, and both x and y respectively. Each clique of G' not containing both x and y is also a clique of G, and as $xy \notin E$, each clique of G' containing both x and y is not a clique of G. Thus $X \cup Y \cup L_0 = L \cap L'$. It can be shown that if $C \in L$ and $C \notin L'$, then $C \in \{M - \{x\} \mid M \in W\} \cup \{M - \{y\} \mid M \in W\}$. Let W_1 be a set containing the elements of $\{M - \{x\} \mid M \in W\}$ that are not properly contained in any element of Y, and similary let W_2 be a set containing the elements of $\{M - \{y\} \mid M \in W\}$ that are not properly contained in any element of X.

Theorem. $L = L_0 \cup X \cup Y \cup W_1 \cup W_2$.

This theorem is the basis of the following algorithm to compute all cliques of $G = (V, E)$, given all cliques of $G' = (V, E')$ with $E \subset E'$. Initialize $L = L'$ and $D = E - E'$. Choose an edge $xy \in D$ and remove xy from D. Compute four sets X, Y, W and K as follows. For each $C \in L$, if $x \in C$ but not $y \in C$ then place C in X, if $y \in C$ but not $x \in C$ then place C in Y, if $x \in C$ and $y \in C$ then place C in W, and place C in K otherwise. For each $C \in W$, if $C - \{x\}$ is not contained in any element of Y then $C - \{x\}$ is added to K, and if $C - \{y\}$ is not contained in any element of X then $C - \{y\}$ is added to K. Set $L = K \cup X \cup Y$. Choose another edge in D (if any) and repeat the procedure until no edge in D

is left. At that moment L contains all cliques of G and the algorithm terminates. An upper bound for the time complexity of the algorithm when only one edge is removed, is approximately $\frac{1}{4}(r')^2$, where r' is the number of cliques of G'. A lower bound on the time complexity is approximately r'.

5.5 An algorithm testing combinations of vertices

In [19] Corneil presents an algorithm for generating all cliques of a graph G of order k, with $k = n, \ldots, 1$. The algorithm described here contains some additions to the original algorithm of Corneil, made by Mulligan ([57]). The algorithm uses an array C of vertices, representing the subgraph under examination, and a "working" graph G_1. Both G and G_1 are represented by its adjacency matrix. The cliques that are generated have to be stored. When a complete subgraph has been found, it is necessary to test whether it is not contained in a clique already generated. The algorithm searches for complete subgraphs of order k as follows. First G_1 is set equal to G. All edges incident to vertices of degree less than $k-1$ and all edges contained in less than $k-2$ triangles in G_1 are deleted from G_1. This is repeated until no more deletions can be made. If G_1 is not empty, then choose a vertex v of minimum degree in G_1. All vertices adjacent to v in G_1 are stored in an array M. Then a subroutine ([15]) is called to compute a combination of $k-1$ vertices out of the elements of M. This combination is placed in C. If C represents a complete subgraph of order $k-1$, then $C \cup \{v\}$ represents a complete subgraph of order k. Add v to C. C is compared with all cliques of order greater than k to determine whether C is contained in any of them. If C is not contained in a clique of order greater than k, then C is a clique and is added to the list of cliques. Edges incident with vertices in C of degree $k-1$ in G and edges that are contained in $k-2$ triangles of G are deleted from G. Then, and in case C was not a clique or even a complete subgraph, a new combination of $k-1$ vertices out of the elements of M is computed and examined. This is repeated until all possible combinations have been examined. The edges incident with v are deleted from G_1 and v becomes a vertex of minimum degree of the new G_1. Then all cliques of order k containing the new vertex v are generated as described above, and this procedure is repeated until all cliques of order k have been enumerated. The algorithm enumerates the cliques of G of order k, with k decreasing. If G has a maximal clique of order $k \neq n$, it is useless to search for cliques of order greater than k. From results by Erdös and Rényi ([23]) it follows that the number of complete subgraphs of order k in a random graph of n vertices and e edges is approximately

$$C(k) = \frac{\binom{n}{k}\binom{maxe - maxk}{e - maxk}}{\binom{maxe}{e}},$$

where $maxe = \frac{1}{2}n(n-1)$ and $maxk = \frac{1}{2}k(k-1)$. If $C(k)$ becomes less than 1.0 as k increases, G has probably no clique of order greater than k. Let k_0 be the minimum number k such that $C(k) < 1.0$. The algorithm starts with searching for complete subgraphs of order $k = k_0$. If a complete subgraph of order k_0 is found, then k is increased by 1 and the algorithm starts searching for larger complete subgraphs. This is repeated until the algorithm finds a number m such that G has a complete subgraph of order $m-1$, but no complete subgraph of order m. Then all cliques of order $m-1$ are generated. When all cliques of order k (for any k) are enumerated as described above or if G has no cliques of order k, then k is decreased by 1. If the current graph G is empty or $k < 2$, then all cliques of G have been enumerated and the algorithm terminates. Otherwise the algorithm continues with enumerating all cliques of order k of the current k.

In [57] Mulligan compares the algorithms of Bierstone, Bron and Kerbosch, and Corneil. Based on practical results, the algorithm of Bron and Kerbosch appears to be the most efficient in most of the cases.

5.6 An algorithm using the adjacency matrix

In [33] Harary and Ross give the following algorithm to enumerate all cliques of order $l \geq 3$ of a graph. Let the vertices of G be numbered $1, 2, \ldots, n$, and let A denote the adjacency matrix of G. Throughout this section, a clique is defined as a maximal complete subgraph of order $l \geq 3$. For two $n \times n$ - matrices B and C we define the elementwise product $B \times C$ to be the $n \times n$ - matrix D with $D_{ij} = B_{ij}C_{ij}$. Consider the matrix $A^2 \times A$. If the entry i, j of $A^2 \times A$ equals 0, then there exists no clique C such that v_i and v_j both are contained in C. Otherwise, there exists at least one clique containing both v_i and v_j. Consequently, if the i^{th} row of $A^2 \times A$ consists entirely of zeros, then v_i is not contained in any clique. Let $M(G)$ be the submatrix obtained from $A^2 \times A$ by deleting the rows and columns corresponding to vertices that are not contained in any clique. Let V be a vertex contained in at least one clique. Let $r(v)$ denote the sum of the elements of the row of $M(G)$ corresponding to v, and let $n(v)$ be the number of non-zero entries of that row (i.e. $n(v)$ is the number of vertices that are in at least one clique containing v, excluding vertex v). By a result of Festinger ([24]), the following lemma holds.

Lemma. *Vertex v is contained in exactly one clique if and only if $r(v) = n(v)(n(v)-1)$).*

The algorithm of Harary and Ross first computes $M(G)$ and $r(v)$ and $n(v)$ for each vertex v which is contained in at least one clique. Using the above lemma, they test whether G has a vertex v which is contained in exactly one clique. If no such vertices exist, then G is a graph of which each vertex is contained in at least two cliques. Otherwise, let v be a vertex contained in exactly one clique. The set C_v of vertices corresponding to the non-zero entries of the row of $M(G)$ corresponding to v together with vertex v is a clique of G and is output. Let C'_v denote the set of vertices of C_v which are contained in exactly one clique. The algorithm computes C'_v and repeats the above procedure with the graph G' induced by $V(G)-C'_v$, until a graph is obtained of which each vertex is contained in at least two cliques, or until no vertices are left. Let G now denote a graph of which each vertex is contained in at least two cliques. For each vertex x of G, let $S(x)$ be the set of vertices containing x and each vertex v_j such that the j^{th} entry of the row of $M(G)$ corresponding to x is non-zero. Let $Q(x)$ denote the union of all sets $S(w)$ such that $w \notin S(x)$. Let v be a vertex of G such that $r(v)$ is minimal. The sets $S(v)$ and $Q(v)$ are computed. The union of $S(v)$ and $Q(v)$ covers $V(G)$, but $S(v) \cap Q(v)$ may be non-empty. Harary and Ross show that there exists no clique C with $C \cap S(v) \neq \emptyset$ and $C \cap Q(v) \neq \emptyset$ which does not entirely lie in exactly one of the sets $S(v)$ and $Q(v)$. Thus, the set of cliques of G can be partitioned into the set of cliques of G_S, the subgraph of G induced by $S(v)$, and the set of cliques of G_Q, the subgraph of G induced by $Q(v)$. The subgraph G_Q is stored and the whole procedure is repeated with G_S. If no vertices are left after all vertices that are contained in exactly one clique have been deleted from the subgraph being processed, then the algorithm repeats the whole procedure with the next subgraph stored. If no subgraphs are left, then the algorithm terminates.

5.7 Algorithms for enumerating complete subgraphs of fixed order

In section 2.6 we already described a few algorithms for enumerating all triangles, i.e. all complete subgraphs of order 3. One of these algorithms is an algorithm of Chiba and Nishizeki ([18]). In the same reference they presented an algorithm for enumerating all cliques (see section 5.1) and an algorithm for enumerating all complete subgraphs of order l, for fixed l. The basic idea of this algorithm is the following. Select a vertex v_1 of G. Let G_{l-1} be the subgraph of G induced by the neighbours of v_1. Choose a vertex v_2 of G_{l-1} and let G_{l-2} be the subgraph induced by the neighbours of v_2 in G_{l-1}. If it is possible to repeat this procedure until we have a graph G_2 with at least one edge, then we have a complete subgraph of order l, containing the terminal vertices of that edge and the vertices $v_1, v_2, \ldots, v_{l-2}$. Chiba and Nishizeki represent the graph by an adjacency list for each vertex v. Instead of computing and storing the induced subgraphs G_k, they label the vertices of G_k with k. In order to find the neighbours of a vertex v in G_k quickly, the adjacency lists are rearranged such that the neighbours of v in G_k occupy the first entries of the adjacency list of v. To keep track of the vertices that may be contained in a complete subgraph of order l, the algorithm uses a stack C. At each moment, the vertices on the stack are all adjacent to each other. Initially, C is empty and all vertices are labeled l. The algorithm contains a recursive procedure with two parameters k and U. The level of recursion is $l-k$ and U is the vertex set of G_k. Let $d_k(v)$ denote the degree of a vertex v in G_k. On entering the procedure, if $k \neq 2$, $d_k(v)$ is computed for each $v \in U$. The vertices of U are numbered in decreasing order of their degree in G_k (so the vertices have a number and a label). Starting with $i = 1$, the set U' of all neighbours of v_i in G_k is computed. The vertices in U' are relabeled $k-1$. For each $u \in U'$, the neighbours of u in U' are placed in the first entries of the adjacency list of u. Vertex v_i is added to C and the procedure is called recursively with $k-1$ and U'. After returning from the recursive call, v_i is deleted from C. All vertices of U' are relabeled k. To avoid duplication, v_i is relabeled $k+1$, and v_i is moved to the element next to the last vertex with label k in the adjacency list of u, for each $u \in U'$. This procedure is repeated for each v_i, $i = 1, 2, \ldots, |U|$. The recursion terminates when k equals 2 on entering the recursive procedure. In that case, for each edge induced by a vertex pair $\{x, y\} \subseteq U$, the set of vertices $\{x, y\} \cup C$ is a complete subgraph of order l, and is printed out. The procedure is called for the first time with $k = l$ and $U = V(G)$. This algorithm lists all complete subgraphs of order l in G in $O(l.\alpha(G)^{l-2}.e)$ time and linear space, where $\alpha(G)$ is the arboricity of G (see section 2.6).

6. Maximal independent sets

Let $G = (V, E)$ be an undirected graph and let M be a maximal independent set of G. The complementary graph $\overline{G}$ of G is defined as $\overline{G} = (V, \overline{E})$, where $xy \in \overline{E}$ if and only if $xy \notin E$. In G there exist no edges between elements of M, so in $\overline{G}$ each element of M is connected to every other element of M and M is a complete subgraph of $\overline{G}$. Moreover, since M is maximal as independent set of G, M is a clique of $\overline{G}$. Thus, the enumeration of maximal independent sets of G is equivalent to the enumeration of cliques of $\overline{G}$, and each algorithm for enumerating all cliques of a graph can be used to enumerate all maximal independent sets of its complement and vice versa. In this chapter we describe two algorithms for enumerating maximal independent sets of G without using $\overline{G}$: algorithms using maximal independent sets of subgraphs and an algorithm based on Boolean arithmetic.

6.1 Algorithms using maximal independent sets of subgraphs

In [64] Paull and Unger give an algorithm to enumerate all maximal independent sets of G. Let the vertices of G be numbered $1, \ldots, n$. For each j, $1 \leq j \leq n$, let G_j be the subgraph of G induced by the vertices $1, \ldots, j$ and let M_j denote the set of all maximal independent sets of G_j. Paull and Unger give a procedure to generate M_{j+1}, given M_j. Let M' be a maximal independent set of G_{j+1}. If $j+1 \in M'$, then M' is a maximal independent set of G_j. If $j+1 \notin M'$, then $M' - \{j+1\} \subseteq M$ for some maximal independent set of G_j and $M' = (M - A(j+1)) \cup \{j+1\}$, where $A(j+1)$ is the adjacency list of $j+1$. Define L_{j+1} to be the set $\{M' \mid M' = (M - A(j+1)) \cup \{j+1\}$, for some $M \in M_j\}$, then we have the following lemma.

Lemma. $M_{j+1} \subseteq M_j \cup L_{j+1}$.

The elements of the set $M_j \cup L_{j+1}$ are all independent sets of G_{j+1}, but $M_j \cup L_{j+1}$ may contain non-maximal elements or duplications. These non-maximal elements and duplications are eliminated by comparing each element of $M_j \cup L_{j+1}$ to every other element. Thus, M_{j+1} is generated from M_j in $O(n.m_j^2)$ time, where $m_j = |M_j|$. All maximal independent sets of G are found by applying the algorithm $n-1$ times, starting with $M_1 = \{\{1\}\}$. The overall time complexity is $O(n^2.m_0^2)$, where m_0 is the number of maximal independent sets of G. The algorithm requires $O(e + n.m_0)$ space.

Tsukiyama et al. present in [89] an algorithm which is similar to the algorithm of Paull and Unger, but has a better time and space bound. Let $W \subset V$ be a set of vertices of G, let $E(W) \subset E$ be the set of edges joining vertices in W and define $G(W) = (V, E(W))$. Let $A(v)$ denote the adjacency list of v in G. Let $W_i, W_{i-1} \subset V$ such that $W_i = W_{i-1} \cup \{x\}$, and let M_{i-1} and M_i denote the set of all maximal independent sets of $G(W_{i-1})$ and $G(W_i)$ respectively. Tsukiyama et al. first show how we can generate M_i, given the set M_{i-1}. Let $A_i(v)$ denote the adjacency list of v in $G(W_i)$, and let $A = A_i(x)$. M_{i-1} can be partitioned into the following two sets.

$$M_{i-1}(x, \overline{A}) = \{M' \in M_{i-1} \mid M' \cap A = \varnothing\}$$

$M_{i-1}(x, A) = \{M' \in M_{i-1} \mid M' \cap A \neq \varnothing\}$

If $M' \in M_{i-1}(x, \bar{A})$, then since $M' \cap A = \varnothing$, M' is also a maximal independent set of $G(W_i)$. If $M' \in M_{i-1}(x, A)$, then since $M' \cap A \neq \varnothing$, $M' - \{x\}$ is a maximal independent set of $G(W_i)$. Let $M_i(\bar{x}, A)$ be the set $\{M' - \{x\} \mid M' \in M_{i-1}(x, A)\}$, then $M_{i-1}(x, \bar{A}) \cup M_i(\bar{x}, A) \subset M_i$. Now let the adjacency list A of x in $G(W_i)$ consist of the vertices $v_1, v_2, \ldots, v_p$. Tsukiyama et al. prove that $M \in M_i$ and $M \notin M_{i-1}(x, \bar{A}) \cup M_i(\bar{x}, A)$ if and only if $M = M' - A$ for some $M' \in M_{i-1}(x, A)$ and M' satisfies the following condition.

Condition. *For any* $v \in \bigcup_{y \in M' \cap A} A_{i-1}(y)$,

(i) *if* $v \notin A$, *then* v *is adjacent to some vertex of* $M' - A$ *in* $G(W_{i-1})$, *or*

(ii) *if* $v \in A$ *and* $v = v_j$ *for some* j, $1 \leq j \leq p$, *then in* $G(W_{i-1})$ v *is adjacent to some vertex of* $M' - A$ *or some* $v_k \in M' \cap A$ *with* $k > j$.

Let $M'_i(x, \bar{A})$ denote the set of all $M' - A$, with $M' \in M_{i-1}(x, A)$ and M' satisfies the above condition.

Theorem. $M_i = M_{i-1}(x, \bar{A}) \cup M'(x, \bar{A}) \cup M_i(\bar{x}, A)$.

Note that the three sets in the right-hand side of the equation are disjoint. This theorem is the basis of the algorithm. The vertices of G are numbered $1, \ldots, n$. For each i, $1 \leq i \leq n$, W_i is defined as the set of vertices $\{1, \ldots, i\}$. The algorithm consists of a recursive procedure with one parameter i, denoting vertex i. The algorithm uses a global variable M. Each time the recursive procedure is called with actual parameter i, M contains some maximal independent set of $G(W_i)$. On entering the recursive procedure, if $i < n$ and $M \cap A_{i+1}(i+1) = \varnothing$, then the procedure is called recursively with $i+1$. After returning from this call, the procedure terminates. Otherwise, if $i < n$ and $M \cap A_{i+1}(i+1) \neq \varnothing$, then M is set to $M - \{i+1\}$, and the procedure is called recursively with $i+1$. After returning from this call, M is reset to $M \cup \{i+1\}$, and the procedure tests whether M satisfies the above condition. If M satisfies the condition, then M is set to $M - A_{i+1}(i+1)$ and the procedure is called recursively with $i+1$. After returning from this last recursive call, M is reset to $M \cup A_{i+1}(i+1)$. Then, and in case M did not satisfy the condition, the procedure terminates. If $i = n$ on entering the procedure, then the recursion terminates. In that case M contains a maximal independent set of G and is output. Initially, M equals V, the vertex set of G. The recursive procedure is called for the first time with actual parameter 1. Tsukiyama et al. prove that this algorithm can be implemented to run in $O(n.e.m_0)$ time and $O(n + e)$ space, where m_0 is the number of maximal independent sets of G. In [18], Chiba and Nishizeki show that the running time of the algorithm can be reduced to $O(\alpha(G).e.m_0)$, where $\alpha(G)$ is the arboricity of G (see section 2.6), by numbering the vertices $1, \ldots, n$ such that $d(v_1) \leq d(v_2) \leq \cdots \leq d(v_n)$. The algorithms of Paull and Unger and of Tsukiyama et al. are described by Lawler in [45] and by Lawler, Lenstra and Rinnooy Kan in [46].

6.2 An algorithm based on Boolean arithmetic

Consider each vertex of a graph G as Boolean variable. Define a Boolean arithmetic on the set of vertices as follows. The Boolean sum $v + w$ of two vertices denotes the operation of including vertex v or vertex w or both. The Boolean product $v.w$ denotes the operation of including both v and w. The complement $\bar{v}$ of vertex v denotes the operation of not including v. Note that $\overline{v+w} = \bar{v}.\bar{w}$ and $\overline{vw} = \bar{v} + \bar{w}$. An edge between two vertices v and w of G is represented by the product $v.w$. Define $\phi = \sum_{vw \in E} v.w$. ϕ represents the operation of including one or more pairs of adjacent vertices. Let $\bar{\phi}$ denote the complement of ϕ. $\bar{\phi}$ can be written as $f_1 + f_2 + \cdots + f_k$, where f_i is a Boolean product of the complements of vertices for $i = 1, \ldots, k$. $\bar{\phi}$ denotes the operation of not including any pair of adjacent vertices, and as $\bar{\phi} = f_1 + f_2 + \cdots + f_k$, this is equal to the operation of including one or more products f_i. Including a product $f_i = \bar{v}_1.\bar{v}_2 \cdots \bar{v}_p$ equals the operation of not including any vertex v_j, $1 \le j \le p$. Thus, the set $V - \{v_j \mid \bar{v}_j \text{ appears in } f_i\}$ is a maximal independent set for each i, $1 \le i \le k$, and each maximal independent set corresponds to a Boolean product f_i for some i, $1 \le i \le k$. This approach is due to Deo ([21]).

7. Cut-sets

Let G be a graph with n vertices. As in the case of cycles, we define the cut-set vector space $S(G)$ as the set of all cut-sets of G together with the empty set and the set of edge-disjoint unions of cut-sets. $S(G)$ is a vector space over F_2, the field of integers modulo 2, with the addition of any two elements of $S(G)$ defined as the ring sum of the sets of edges of the elements. Let T be a spanning tree of G. The set of fundamental cut-sets $C_1, C_2, \ldots, C_{n-1}$ with respect to T is a basis of $S(G)$. All cut-sets of G can be enumerated by generating a listing of the elements of $S(G)$ and testing which element is a cut-set. A listing of the elements of $S(G)$ can be generated by taking all possible combinations of fundamental cut-sets. This approach is not very efficient. Since $S(G)$ has 2^{n-1} elements, computing all elements of $S(G)$ requires $\Omega(2^{n-1})$ time. The algorithms we describe in this chapter have a different approach. In section 7.1 we describe backtrack algorithms, in section 7.2 we describe an algorithm using Gaussian elimination and in section 7.3 we describe an algorithm using the path matrix.

7.1 Backtrack algorithms

Let $G = (V, E)$ be an undirected, connected graph without self-loops or multiple edges. For $X \subseteq V$, define $E(X)$ to be the set of all edges of G joining vertices of X, and let $G(X)$ be the subgraph of G induced by X. For $X, Y \subseteq V$, wtih $X \cap Y = \emptyset$, define $E(X, Y)$ to be the set of all edges of G joining vertices of X to vertices of Y. Let s and t be two vertices of G. For each $s-t$ cut-set C there exists a set $X \subset Y$ such that $s \in X$, $t \in \bar{X} = V - X$, and $C = E(X, \bar{X})$. Let S and T be disjoint subsets of V with $s \in S$ and $t \in T$. Define $\Lambda(S \mid T)$ as the set of all $s-t$ cut-sets $E(X, \bar{X})$ such that $S \subset X$ and $T \subset \bar{X}$. Clearly, $\Lambda(\{s\} \mid \{t\})$ is the set of all $s-t$ cut-sets of G. The following lemma of Tsukiyama et al. ([90]) is the basis of the algorithms of this section.

Lemma. *For any* $v \in V - (S \cup T)$, *we have* $\Lambda(S \mid T) = \Lambda(S \cup \{v\} \mid T) + \Lambda(S \mid T \cup \{v\})$.

For each $v \in V$, $A(v)$ denotes the adjacency list of v in G. For any $W \subseteq V$, $A(W)$ denotes the set $\bigcup_{v \in W} A(v)$ and $A^+(W) = A(W) - W$. In [90], Tsukiyama et al. prove the following observations. If $G(S)$ is connected and $G(W)$ is a maximal connected component of $G(\bar{S})$ with $t \in W$, then $\Lambda(S \mid T)$ is non-empty if and only if $T \subset W$. If $G(S)$ is connected and $G(\bar{S})$ is not, and there exists a maximal connected component $G(W)$ of $G(\bar{S})$ with $T \subset W$, then $\Lambda(S \mid T) = \Lambda(\bar{W} \mid T)$ and $G(\bar{W})$ is connected. Finally, if $G(S)$ and $G(\bar{S})$ are both connected and $A^+(S) - T = \emptyset$, then $\Lambda(S \mid T) = \{E(S, \bar{S})\}$. Tsukiyama et al. give the following algorithm based on these observations. The algorithm consists of a recursive procedure with two parameters S and T, denoting disjoint subsets of V with $s \in S$ and $t \in T$. At each call of the procedure, $G(S)$ and $G(\bar{S})$ are connected. On entering the procedure, if $A^+(S) - T = \emptyset$, then a new $s-t$ cut-set $E(S, \bar{S})$ is output and the procedure terminates. Otherwise, choose a vertex $v \in A^+(S) - T$. If $G(\bar{S} - \{v\})$ is connected, then the procedure is called with $S \cup \{v\}$ and T. After returning from this recursive call, the procedure is called again recursively with S and $T \cup \{v\}$. After returning from this second call, the procedure terminates. If $G(\bar{S} - \{v\})$ is not connected, then the algorithm computes a set of vertices W such that $G(W)$ is a maximal connected component of $G(\bar{S}-\{v\})$ with $t \in W$. If $T \subset W$, then the procedure is called recursively with $\bar{W}$ and T. After returning from this recursive call, and in case $T \not\subset W$, the procedure is called again with S and $T \cup \{v\}$. After returning from this call, the procedure terminates. The first time the procedure is called, S equals $\bar{W}$, where W is a set of vertices such that $G(W)$ is a maximal connected component of $G(V-\{s\})$ with $t \in W$, and T equals $\{t\}$. The time complexity of this algorithm is $O((n+e)(n-\log c)c)$, where c is the number of $s-t$ cut-sets of G. The space complexity is $O(n+e)$. This algorithm of Tsukiyama et al. is an other implementation of an algorithm of Jensen and Bellmore ([38]). The algorithm of Jensen and Bellmore builds a rooted tree T. The leaves of T represent cut-sets of G. To make distinction between the representation of G and the representation of T, the vertices of T are called 'nodes' and the edges of T are called 'branches'. Let the vertices of G be numbered $1, \ldots, n$. During the construction of T, the nodes are numbered $0, 1, 2, \cdots$ and the branches are labeled jT or jF, where j is the number of a vertex of G and the labels of the branches have the following meaning. If $E(X, \bar{X})$ is an $s-t$ cut-set of G, then X consists exactly of the vertices j of G such that branch jT is contained in the unique path from the root to the leave representing $E(X, \bar{X})$. The vertices j of G such that branch jF is contained in the path from the root to the leave representing $E(X, \bar{X})$ are contained in $\bar{X}$, but the entire set $\bar{X}$ consists of $V-X$. For each node i of T, let P_i be the unique path from the root to i. To node i, four sets of vertices T_i, F_i, Y_i and Z_i are associated. T_i contains all vertices j of G such that branch jT is contained in P_i. F_i contains all vertices j of G such that branch jF is contained in P_i. Y_i equals the set $V-(T_i \cup F_i)$ and Z_i is the set of vertices $j \in Y_i$ that are adjacent to a vertex of T_i. The algorithm first creates a tree with three nodes 0, 1 and 2, a branch from 0 to 1 labeled sT and a branch from 1 to 2 labeled tF. The nodes 0 and 1 are marked 'scanned' and node 2 is marked 'unscanned'. The algorithm proceeds in stages. At each stage, the unscanned node with the greatest index i is chosen and marked 'scanned'. The sets T_i, F_i, Y_i and Z_i are computed. If $Z_i = \emptyset$, then a new $s-t$ cut-set $E(T_i, \bar{T}_i)$ is output, and the algorithm proceeds with the next stage. Otherwise, a vertex $v \in Z_i$ is chosen. If $G(\bar{T}_i-\{v\})$ is connected, then two new nodes k and $k+1$ are created, where $k-1$ is the number of nodes in the current tree. The nodes k and $k+1$ are marked 'unscanned'. A branch from i to k, labeled vT, and a branch from i to $k+1$, labeled vF, are added to the tree. Then the algorithm proceeds with the next stage. If $G(\bar{T}_i-\{v\})$ is not connected, then a set of vertices W_i is computed such that $G(W_i)$ is a maximal connected component of $G(\bar{T}_i-\{v\})$ with $t \in W_i$. If $F_i \subset W_i$, then a new node k and a branch from i to k labeled vT are added to the tree, where $k-1$ is the number of nodes in the current tree. Compute the set $W'_i = \bar{T}_i - W_i - \{v\}$. Let W'_i be the set $\{w_1, w_2, \ldots, w_p\}$. New nodes $k+1, k+2, \ldots, k+p$ are added to the tree, as well as a branch from $k+i-1$ to $k+i$ labeled w_iT, for each i, $i = 1, \ldots, p$. The nodes $k+1, k+2, \ldots, k+p-1$ are marked 'scanned', and node $k+p$ is marked 'unscanned'. Finally, a node $k+p+1$ and a branch from i to $k+p+1$ labeled vF are added to the tree. Node $k+p+1$ is marked 'unscanned' and the algorithm proceeds with the next stage. If $F_i \not\subset W_i$, then a new node k and a branch labeled vF are added to the tree. Node k is marked 'unscanned' and the algorithm proceeds with the next stage. If no unscanned vertices are left, then the algorithm terminates.

Define an edge to be $s-t$ *path isolated* if it does not belong to any $s-t$ cut-set. Assume that G has no $s-t$ path isolated edges. In [90] Tsukiyama et al. prove the following lemma, using the above notation.

Lemma. *Let G(S) and $G(\bar{S})$ both be connected and let $\{v_1, v_2, \ldots, v_j\}$ be the set of vertices of $A^+(S)-T$ which are not an articulation vertex of $G(\bar{S})$. Then*

$$\Lambda(S \mid T) = \{E(S, \bar{S})\} + \sum_{i=1}^{j} \Lambda(S \cup \{v_i\} \mid T \cup \{v_1, v_2, \ldots, v_{i-1}\}) \text{ and}$$

$$\Lambda(S + \{v_i\} \mid T + \{v_1, v_2, \ldots, v_{i-1}\}) \neq \emptyset.$$

This lemma is the basis of another algorithm of Tsukiyama et al., presented in [90] and consisting of a recursive procedure with two parameters S and T. The procedure has two local variables *CAND* and T'. S, T, T' and *CAND* are sets of vertices. On entering the procedure, $E(S, \bar{S})$ is a new $s-t$ cut-set and is output. T' becomes the empty set and *CAND* becomes the set of vertices of $A^+(S)-T$ which are not an articulation vertex of $G(\bar{S})$. Choose a vertex $v \in CAND$ (if any) and delete v from *CAND*. The procedure is then called recursively with $S \cup \{v\}$ and $T \cup T'$. After returning from this recursive call, T' is set to $T' \cup \{v\}$. Then another vertex $v \in CAND$ is chosen (if any) and this procedure is repeated until $CAND = \emptyset$. Before calling the procedure for the first time, all $s-t$ path isolated edges are deleted from G. Then the procedure is called with $S = \{s\}$ and $T = \{t\}$. The time complexity of this algorithm is $O((n + e)(c + 1))$ and the space complexity is $O(n^2)$. Tsukiyama et al. show that space can be reduced to $O(n + e)$. To obtain all cut-sets of G without duplication, they give the following algorithm. Choose a vertex $v \in V$. Add v to S, a variable representing a set of vertices. Choose a vertex $t \in A^+(S)$. Let G' be the graph obtained from G by shrinking S into a vertex s. Using the previous described algorithm, enumerate all $s-t$ cut-sets of G'. After all $s-t$ cut-sets of G' have been enumerated, add t to S. Choose another vertex $t \in A^+(S)$ and repeat this procedure until $S = V$. This algorithm has a time complexity $O((n + e)(c + 1))$, where c is the number of all cut-sets of G.

7.2 An algorithm using Gaussian elimination

In [48] Martelli gives a regular algebra which can be used for the enumeration of all cut-sets of a graph. The algebra C consists of a set R with the operations sum and multiplication. An element s of R is a set of sets of edges such that no element of s is a superset of some other element of s. If s and t are elements of R, then $s + t$ is defined as the set containing the union of each element of s with each element of t, where all elements which are a superset of some other element are deleted. The zero element is the set $\{\emptyset\}$. The multiplication st is defined as the set obtained by taking the union of s and t, and deleting all elements which are a superset of some other element, where $e = \emptyset$ is the unity element. In [49] Martelli uses this algebra for the following algorithm. Consider the vertices of the graph as numbered $1, 2, \ldots, n$. Define a $n \times n$ - matrix A as follows. If an edge from i to j exists, then $A_{ij} = \{\{ij\}\}$, else $A_{ij} = \{\{\emptyset\}\}$. The elements of A belong to the algebra defined above. The closure of

A is $A^* = \sum_{k=0}^{\infty} A^k$ and can be obtained as the solution of the equation $Y = AY + U$, where U is a $n \times n$ - matrix with $U_{ij} = e$ if $i = j$ and $U_{ij} = \{\{\emptyset\}\}$ otherwise. For each $i, j, 1 \le i, j \le n$, A^*_{ij} represents the set of all $i-j$ cut-sets. As $e + s = e$ in the algebra C, $A^* = U + A + A^2 + \cdots + A^{n-1}$ and can be computed by Gaussian elimination as follows. Set $A^0_{ij} = e$ if $i = j$, and $A^0_{ij} = A_{ij}$ otherwise. Compute A^k_{ij} for each $k = 1, 2, \ldots, n$ and each $i, j, 1 \le i, j \le n$ according to the following rules.

(1) If $i \ne k$ and $j \ne k$, then $A^k_{ij} = A^{k-1}_{ij} + A^{k-1}_{ik} A^{k-1}_{kj}$.
(2) If $i = k$ or $j = k$, then $A^k_{ij} = A^{k-1}_{ij}$.

The elements of A^n_{ij} contain all $i-j$ cut-sets. Note that a similar procedure was used by Fratta and Montanari in [25] to enumerate all simple paths in a graph (see section 3.4). The complexity of the algorithm of Martelli depends upon the complexity of the computation of A^k_{ij}. A direct implementation of the operations sum and multiplication as defined above is not very efficient. For the case that G is undirected, Martelli gives several theorems which he uses for a more efficient implementation. Define V^k to be the set $\{1, 2, \ldots, k\}$. Define G^k_{ij}, $k = 1, \ldots, n$ to be the subgraph of G induced by $V^k \cup \{i, j\}$, and define G^0_{ij} to be the subgraph of G induced by the vertices i and j. Martelli shows that every A^k_{ij}, $k = 0, 1, \ldots, n$, computed by the algorithm gives the set of all $i-j$ cut-sets of G^k_{ij}. Consider the computation of A^k_{ij}. If $A^{k-1}_{ik} = \{\emptyset\}$ or $A^{k-1}_{kj} = \{\emptyset\}$, then we have $A^k_{ij} = A^{k-1}_{ij}$. If $A^{k-1}_{ij} = \{\emptyset\}$, then i and j are not connected in G^{k-1}_{ij} and thus $A^k_{ij} = A^{k-1}_{ik} \cup A^{k-1}_{kj}$. Let $A^{k-1}_{ik} \ne \{\emptyset\}$, $A^{k-1}_{kj} \ne \{\emptyset\}$ and $A^k_{ij} \ne \{\emptyset\}$, then i and j are connected in G^k_{ij}. Rewrite (1) as follows.

(1)′ If $i \ne k$ and $j \ne k$, then $A^k_{ij} = (A^{k-1}_{ij} + A^{k-1}_{ik})(A^{k-1}_{ij} + A^{k-1}_{kj})$.

Define $F^k_{ij} = (V^k_{ij}, E^k_{ij})$ as the maximal connected component of G^k_{ij} containing i and j. Let $\alpha \in A^k_{ij}$. α is a minimal $i-j$ cut-set of F^k_{ij}, and divides F^k_{ij} in exactly two connected components with sets of vertices V^i_α and V^j_α, where $i \in V^i_\alpha$ and $j \in V^j_\alpha$. For each edge in α, one terminal vertex is contained in V^i_α and the other is contained in V^j_α. Define M^i_α and M^j_α as the sets of terminal vertices of edges in α such that $M^i_\alpha \subseteq V^i_\alpha$ and $M^j_\alpha \subseteq V^j_\alpha$. Now, let $\alpha \in A^{k-1}_{ij}$ and $\beta \in A^{k-1}_{ik}$. Martelli proves the following necessary condition for a set $\alpha \cup \beta$ to be an $i-j$ cut-set: $M^i_\alpha \subseteq V^i_\beta$ and $M^i_\beta \subseteq V^i_\alpha$ (*).

Theorem. *If α and β satisfy condition (*) and if the following two conditions hold:*
(i) $V^i_\alpha \subseteq V^i_\beta$ *or* $V^i_\beta \subseteq V^i_\alpha$,
(ii) $V^j_\alpha \cap V^k_\beta \ne \emptyset$ *or an edge* kj *exists* $\in G$,
then $\alpha \cup \beta$ *is an* $i-j$ *cut-set of* F^k_{ij}.

Assume the elements $\alpha_1, \alpha_2, \cdots$ of A^k_{ij} to be ordered such that if $V^i_{\alpha_l} \subset V^i_{\alpha_m}$, then $l < m$. For $\alpha \in A^{k-1}_{ij}$ and $\beta \in A^{k-1}_{ik}$ we have the following theorem.

Theorem. *If $V^i_\alpha \subseteq V^i_\beta$, then for each $\gamma \in A^{k-1}_{ij}$ with $V^i_\gamma \not\subseteq V^i_\beta$, $\alpha \cup \beta$ is not an $i-j$ cut-set of F^k_{ij}.*

Furthermore, if $j < k$, then $A^{k-1}_{ij} + A^{k-1}_{ik}$ can be obtained by simply taking all elements $\alpha \in A^{k-1}_{ik}$ with $j \in V^k_\alpha$. Assuming the elements of A^k_{ij} and A^{k-1}_{ik} to be ordered as described above and using the theorems, Martelli gives the following algorithm to compute $A^{k-1}_{ij} + A^{k-1}_{ik}$. If $j < k$, then α is set to the first element of A^{k-1}_{ik}. If $j \in V^k_\alpha$, then α is added at the end of a variable $SUM1$. α becomes the next element of A^{k-1}_{ik}, and the procedure is repeated with the current α, until all elements of A^{k-1}_{ik} have been examined. If $j \geq k$, then α is set to the first element of A^{k-1}_{ij} and β is set to the first element of A^{k-1}_{ik}. The following conditions are checked.

(i) $M^i_\alpha \subseteq V^i_\beta$ and $M^i_\beta \subseteq V^i_\alpha$,
(ii) $V^i_\alpha \subseteq V^i_\beta$ or $V^i_\beta \subseteq V^i_\alpha$ or the subgraph induced by $V^i_\alpha \cap V^i_\beta$ is connected,
(iii) $V^j_\alpha \cap V^k_\beta \neq \emptyset$ or an edge kj exists in G.

If the three conditions hold, then $\alpha \cup \beta$ is added at the end of $SUM1$. If $V^i_\beta \not\subseteq V^i_\alpha$, then β is deleted from A^{k-1}_{ik}. If $V^i_\alpha \not\subseteq V^i_\beta$, then β becomes the next element of A^{k-1}_{ik}. The procedure is repeated with the current β until all elements A^{k-1}_{ik} are examined or until $V^i_\alpha \subseteq V^i_\beta$ for some β. In those cases, α becomes the next element of A^{k-1}_{ij}, and the whole procedure is repeated with the current α and A^{k-1}_{ik}, until all elements of A^{k-1}_{ij} are examined. At that moment, the algorithm terminates and $SUM1$ contains all elements of $A^{k-1}_{ij} + A^{k-1}_{ik}$ in order. To compute A^k_{ij}, the same algorithm is repeated to compute $SUM2 = A^{k-1}_{ji} + A^{k-1}_{jk}$, and A^k_{ij} is set to $SUM1$ followed by the reverse of $SUM2$. When applied to a complete graph, this implementation of the algorithm enumerates all cut-sets in $O(c)$ time, where c is the number of cut-sets.

7.3 An algorithm using the path matrix

Let s and t be two vertices of a graph. Let $P_1, P_2, \ldots, P_k$ be all paths from s to t. Let the edges of G be numbered $1, \ldots, e$. The *path matrix* with respect to s and t is a $k \times e$ - matrix Q with $Q_{ij} = 1$ if edge e_j is contained in path P_i and $Q_{ij} = 0$ otherwise. Let C be an $s-t$ cut-set of G. The removal of the edges of C results in a graph in which there is no path from s to t. So, if C is an $s-t$ cut-set, then C contains an edge of P_i for each i, $1 \leq i \leq k$. Let E be a k - vector obtained from C as follows. For each i, $1 \leq i \leq k$, if at least one edge of C is contained in path P_i, then E_i is set to 1, otherwise E_i is set to 0. If C is an $s-t$ cut-set, then each entry of E equals 1. Let $\{e_{i_1}, e_{i_2}, \ldots, e_{i_p}\}$ be a set of edges of G. Let X be a k - vector obtained as the logical sum of the columns of Q corresponding to $e_{i_1}, \ldots, e_{i_p}$. If each entry of X equals 1, then the removal of $e_{i_1}, \ldots, e_{i_p}$ results in a graph in which there is no path from s to t. However, $\{e_{i_1}, e_{i_2}, \ldots, e_{i_p}\}$ is not necessarily a minimal set having this property. In [60], these observations are used by Nelson, Batts and Beadles in an algorithm for enumerating all $s-t$ cut-sets of a graph. The algorithm proceeds in stages. At stage i, all combinations of i edges are computed. For each combination C, the corresponding vector X is computed. If each entry of X equals 1, then the algorithm tests whether any $s-t$ cut-set with less than i edges is contained in C. If so, C is not minimal and thus C is not an $s-t$ cut-set. Otherwise, C is an $s-t$ cut-set and is stored. The algorithm terminates after stage e.

Literature

For each chapter a list of references is given below.

1. Introduction

[21], [28], [32], [69], [91], [92].

2. Cycles

[2], [6], [11], [12], [13], [18], [20], [22], [29], [35], [36], [39], [40], [43], [44], [50], [51], [52], [61], [67], [68], [70], [71], [73], [74], [75], [76], [80], [81], [82], [83], [84], [86], [87], [95], [96], [99].

3. Paths

[11], [12], [20], [25], [42], [65], [67], [74], [75], [76], [79], [87], [99].

4. Spanning trees

[10], [14], [16], [17], [26], [27], [30], [31], [37], [41], [53], [54], [56], [59], [66], [72], [74], [77], [78], [88], [93], [94], [97], [98].

5. Cliques and complete subgraphs

[1], [4], [5], [7], [8], [9], [18], [19], [23], [24], [33], [47], [57], [58], [62], [63].

6. Maximal independent sets

[18], [21], [45], [46], [64], [87].

7. Cut-sets

[3], [38], [48], [49], [60], [69], [90].

References

[Note. References markes with an asterisk are not referred to in the text.]

[1] E.A. Akkoyunlu, *The enumeration of maximal cliques of large graphs,* Siam J. Comput., vol. 2, no. 1 (1973), pp. 1-6.

[2]* M.T. Ardon and N.R. Malik, *A recursive algorithm for generating circuits and related subgraphs,* 5th Asilomar Conf. on Circuits and Systems, Pacific Grove, California, Nov. 1971, pp. 279-284.

[3]* H. Ariyoshi, *Cut-set graph and systematic generation of separating sets,* IEEE Trans. on Circuit Theory CT-19, no. 3 (1972), pp. 233-240.

[4] J.G. Augustson and J. Minker, *An analysis of some graph theoretical cluster techniques,* Journal ACM, vol. 17, no. 4 (1970), pp. 571-588.

[5]* A.R. Bednarek and O.E. Taulbee, *On maximal chains,* Revue Roumaine de Mathématiques Pures et appliquées 11, no. 1 (1966), pp. 23-25.

[6]* A.T. Berztiss, *A k-tree algorithm for simple cycles of a directed graph,* Tech. Rep. 73-6, Dept. of Computer Sci., University of Pittsburgh, Penn., 1973.

[7] E. Bierstone, *Cliques and generalized cliques in a finite linear graph,* Unpublished Report, University of Toronto (Oct. 1967).

[8] R.E. Bonner, *On some clustering techniques,* IBM Journal of Research and Development 8, no. 1 (Jan. 1964), pp. 22-32.

[9] C. Bron and J.A.G.M. Kerbosch, *Finding all cliques of an undirected graph (Algorithm 457),* Comm. ACM, vol. 16, no. 9 (1973), pp. 575-577.

[10]* P.M. Camerini, L. Fratta and F. Maffioli, *The k-shortest spanning trees of a graph,* Int. Rep. 73-10, IEE-LCE, Politecnico di Milano, Italy, 1974.

[11] D. Cartwright and T.C. Gleason, *The number of paths and cycles in a digraph,* Psychometrika, 31 (1966), pp. 179-199.

[12] S.G. Chan and W.T. Chang, *On the determination of dicircuits and dipaths,* Proc. 2nd Hawaii Internat. Conf. System Sci., Honolulu, Hawaii (1969), pp. 155-158.

[13] J.P. Char, *Master circuit matrix,* Proc. IEE (London) 115 (1968), pp 762-770.

[14] J.P. Char, *Generation of trees, two-trees and storage of master forests,* IEEE Trans. on Circuit Theory CT-15 (1968), pp. 228-238.

[15] P.J. Chase, *Combinations of M out of N objects (Algorithm 382),* Comm. ACM, vol. 13, no. 6 (1970), p. 368.

[16]* S.M. Chase, *Analysis of algorithms for finding all spanning trees of a graph,* Ph. D. Thesis, Rep. 401, Dept. of Comput. Sci., University of Illinois, Urbana, Ill., 1970.

[17]* W.K. Chen, *Generation of trees and k-trees*, Proc. 3rd Allerton Conf. on Circuit and System Theory, 1965, pp. 889-899.

[18] N. Chiba and T. Nishizeki, *Arboricity and subgraph listing algorithms*, Siam J. Comput., vol. 14, no. 1 (1985), pp. 210-223.

[19] D.G. Corneil, *An algorithm for finding complete subgraphs of an undirected linear graph*, Unpublished Report, University of Toronto (1969).

[20] G.H. Danielson, *On finding the simple paths and circuits in a graph*, IEEE Trans. on Circuit Theory CT-15 (1968), pp. 294-295.

[21] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Inc., Englewood Cliffs, N.J., 1974.

[22]* A. Ehrenfeucht, L.D. Fosdick and L.J. Osterweil, *An algorithm for finding the elementary circuits of a directed graph*, Tech. Rep. CU-UC-024-73, Dept. of Computer Sci., University of Colorado, Colorado, 1973.

[23] P. Erdös and A. Rényi, *On random graphs I*, Publicationes Mathematicae 6 (1959), pp. 290-297.

[24] L. Festinger, *The analysis of sociograms using matrix algebra*, Human Relations, vol. 2 (1949), pp. 153-158.

[25] L. Fratta and U. Montanari, *A vertex elimination algorithm for enumerating all simple paths in a graph*, Networks 5 (1975), pp. 151-177.

[26] H.N. Gabow, *Two algorithms for generating weighted spanning trees in order*, Siam J. Comput., vol. 6, no.1 (1977), pp. 139-150.

[27] H.N. Gabow and E.W. Myers, *Finding all spanning trees of directed and undirected graphs*, Siam J. Comput., vol. 7, no. 3 (1978), pp. 280-287.

[28] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[29] N.E. Gibbs, *A cycle generation algorithm for finite undirected graphs*, Journal ACM, vol. 16 (1969), pp. 564-568.

[30]* S.L. Hakimi, *On trees of a graph and their generation*, J. Franklin Inst., vol. 270 (1961), pp. 347-359.

[31] S.L. Hakimi and D.G. Green, *Generation and realization of trees and k-trees*, IEEE Trans. on Circuit Theory CT-11 (1964), pp. 247-255.

[32] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1972.

[33] F. Harary and I.C. Ross, *A procedure for clique detection using the group matrix*, Sociometry 20 (1957), pp. 205-215.