Generating Spanning Trees

by

Malcolm James Smith
B.Sc., University of Victoria, 1985

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in the
Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. F. Ruskey, Supervisor (Dept. of C.Sc.)

Dr. D. Roelants, Supervisor (Dept. of C.Sc.)

Dr. G. MacGillivray, Outside Member (Dept. of Mathematics)

Dr. W. Myrvold, External Examiner (Dept. of C.Sc.)

University of Victoria

Supervisor: Dr. F. Ruskey and Dr. D. Roelants

Abstract

Algorithms for generating all spanning trees of a graph are presented. A series of improvements to the basic contraction-deletion algorithm results in the development of an algorithm which generates spanning trees in a Gray code order in constant amortized time. The improvements consist of removing loop edges in the graph, replacing sets of parallel edges with a single parallel edge record, and removing degree two vertices and replacing each pair of incident edges with a series edge record.

Examiners:

Dr. F. Ruskey, Supervisor (Dept. of C.Sc.)

Dr. D. Roelants, Supervisor (Dept. of C.Sc.)

Dr. G. MacGillivray, Outside Member (Dept. of Mathematics)

Dr. W. Myrvold, External Examiner (Dept. of C.Sc.)

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Dr. Frank Ruskey and Dr. Dominique Roelants for their patience, support and guidance during the development of this thesis.

In loving memory of my dear uncle, Malcolm Smith.

Chapter 1

Introduction and Definitions

The generation of all spanning trees of a graph is an important problem in the study of electrical networks [13], [5], [4]. For each spanning tree of an electrical network, the addition of the voltage source results in a unique cycle. The calculation and summation of the current around the cycle in each of the spanning trees yields the current flowing through each edge in the network. Calculating the current in the network is therefore reduced to the problem of calculating the current in a cycle and summing over all spanning trees [4]. In chemistry, the calculation of ring currents in molecules with closed cycles of atoms is determined by generating all of the spanning trees of a graph which represents the molecule [17].

Methods of generating spanning trees have been studied by several researchers, including Berger [2], Cummins [8], Gabow [9], Kamae [13], Kishi and Kajitani [15], Mayeda and Seshu [18], and Read and Tarjan [20]. Recently, Kapoor and Ramesh [14], and Shioura and Tamura [22] have provided efficient algorithms for generating spanning trees.

1.1 Definitions

A *graph* $G = (V, E)$ consists of a set of vertices V, and a multiset of edges E, such that each element $e \in E$ is an unordered pair of vertices from V. The number of vertices of G is denoted by $n(G) = |V|$, and the number of edges is denoted by $m(G) = |E|$. When G is implied by the context, then $n(G)$ and $m(G)$ are abbreviated by n and m respectively. An edge $e = (x, y) = (y, x)$ is *incident* with vertices x and y, and is said to *join* vertices x and y. The vertices x and y are the *ends* of the edge (x, y). Two vertices are *adjacent* if they are joined by an edge. A *loop* is an edge (x, x) which has identical ends. If two different edges (x, y) and (x', y') have the same pair of ends, then the edges are *parallel* edges. A graph which contains no loops and no parallel edges is a *simple graph*.

A *walk* from vertex v_0 to v_k is an alternating sequence of vertices and edges, $v_0, e_1, v_1, \ldots, e_k, v_k$, such that $e_i = (v_{i-1}, v_i)$. The length of a walk is the number of edges in the sequence. If each of the edges in a walk is distinct, then the walk is a *trail*. If each of the vertices in a walk is distinct, then the walk is a *path*. In a graph with no parallel edges, a walk is uniquely specified by the sequence of vertices. A path from vertex x to y is called an (x, y)-path. Vertex x of an (x, y)-path, P, is called the *origin* of P; vertex y is called the *terminus* of P. The origin and terminus of a walk are defined similarly.

Two vertices, $x, y \in G$, are *connected* if there is an (x, y)-path in G. A *connected graph* is a graph in which there is an (x, y)-path joining every pair of distinct vertices in the graph. If a graph is not connected, then it is a *disconnected* graph.

A *subgraph* $G' = (V', E')$ of $G = (V, E)$ consists of a subset of vertices, $V' \subseteq V$, and a subset of edges $E' \subseteq E$ such that for each edge $e \in E'$, the vertices incident with e are both included in V'. A connected subgraph $G' = (V', E')$ of G is a *connected* *component* of G if and only if none of the vertices in G' are connected to any of the vertices in $G - G'$, and $E' = \{(x, y)|x \in V', y \in V'\}$. Every disconnected graph, G, can be partitioned into a number of *connected components.*

A *cycle* is a walk $v_0, e_1, v_1, e_2, v_2, \ldots, v_k$ in which $v_0 = v_k$. A *simple cycle* consists of an (x, y)-path followed by an edge (y, x), not in the (x, y)-path, and the vertex x. Alternatively, a simple cycle can be defined as a trail in which the origin and terminus are the same vertex, and all other vertices on the walk are distinct. The *length* of a cycle is the number of edges included in the cycle. A *loop* is a simple cycle of length one. A *k-cycle* is a simple cycle of length k. A graph which does not contain any cycles is called an *acyclic* graph.

A *tree* is a connected graph which contains no cycles. A *spanning tree*, t, of a graph G, is a tree which is a subgraph of G and includes all of the vertices of G. An acyclic subgraph of G which includes all of the vertices of G but is not necessarily connected, is called a *spanning forest* [26]. When the graph G is clear from the context, a spanning tree can be denoted by the set of edges contained in the tree, and we can write $t = \{e_1, e_2, \ldots, e_{n-1}\}$. Similarly, a spanning forest is simply represented by a set of edges when the vertex set is clear from the context.

Two common graph operations which form the basis for the algorithms of Chapter 3 are edge contraction and edge deletion. The first operation, *edge contraction*, consists of removing the contracted edge $e = (u, v)$, and replacing the end points of the edge, u and v, with a single *supervertex* s. Each edge (u, x), which is incident with u, is replaced with an edge (s, x) which is incident with the supervertex s. Similarly, each edge incident with v is replaced with a corresponding edge incident with s. Since the contracted edge e is removed, it does not form a loop edge (s, s). If both u and v are adjacent to a vertex w, then the contracted graph contains two parallel edges, corresponding to the two original edges (u, w) and (v, w). The graph G with edge e

contracted is denoted $G \bullet e$.

After the contraction $G \bullet (u, v)$, the supervertex s represents the connected component $(\{u, v\}, \{(u, v)\})$. Since each supervertex (connected component) is a subgraph, let $V(s)$ and $E(s)$ represent the vertex set and the edge set of the supervertex s. Let each vertex v in the original graph be replaced with a supervertex, $v' = (\{v\}, \phi)$, which simply consists of the original vertex v. The supervertex s resulting from a contraction of edge (u, v), incident with two supervertices u and v, is $s = (V(u) \bigcup V(v), E(u) \bigcup E(v) \bigcup \{(u, v)\})$. In the algorithms of Chapter 3, it is assumed that the supervertex formed by a contraction replaces one of the supervertices incident with the contracted edge; after contraction of edge (u, v), either u or v remains as the new supervertex.

The second operation, *edge deletion*, corresponds to the removal of an edge from the graph G. If e is the deleted edge, then $G - e = (V, E - \{e\})$ denotes the graph G with edge e removed [3]. The addition of an edge to a graph or forest is also a common operation in spanning tree generation algorithms. The addition of an edge e to a graph $G = (V, E)$ is represented by the notation $G + e = (V, E \bigcup \{e\})$.

It is well known that if the connected graph G contains n vertices then each spanning tree of the graph contains exactly $n - 1$ edges (see for example [3], page 25). Figures 1.1 and 1.2 show a graph and all of its spanning trees, respectively. The number of spanning trees of a graph G is denoted $\tau(G)$. If an edge $e \in E(G)$ is included in every spanning tree of G, then e is a *bridge*. The removal of a bridge e from a connected graph G, results in a disconnected graph.

The number of spanning trees of a graph can be counted using the following recursive formula due to Cayley (see for example [3], page 33, Theorem 2.8). The proof is due to Bondy and Murty [3].

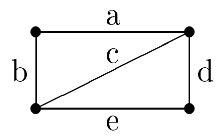

Figure 1.1: A connected graph G

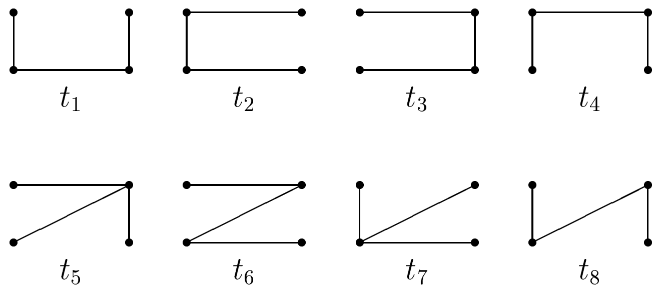

Figure 1.2: The spanning trees of G

Theorem 1.1 (Cayley) *If G is a connected graph, and e is an edge of G, then $\tau(G) = \tau(G \bullet e) + \tau(G - e)$.*

Proof: Every spanning tree of G that does not contain e is also a spanning tree of $G - e$, so $\tau(G - e)$ is the number of spanning trees of G which do not contain e. For each spanning tree t of G which contains e, there corresponds a spanning tree $t \bullet e$ of $G \bullet e$. The correspondence between spanning trees of G which contain e and the spanning trees of $G \bullet e$ is a bijection. The number of spanning trees of G which include e is therefore $\tau(G \bullet e)$, and so $\tau(G) = \tau(G \bullet e) + \tau(G - e)$. $\square$

The *tree graph*, $T(G)$, of a graph G, is defined to be the graph in which each vertex of $T(G)$ corresponds to a spanning tree of G, and two vertices t_i and t_j of $T(G)$ are adjacent if and only if their corresponding spanning trees have $n-2$ edges in common

[8]. The replacement of an edge $e_i \in t_i$ with an edge $e_j \neq e_i$, to form a new spanning tree $t_j = (t_i - e_i) + e_j$ is referred to as an *elementary tree transformation* [18]. A *Gray code* listing of the spanning trees of G is an ordering of the spanning trees such that successive spanning trees differ by a single elementary tree transformation. Figure 1.3 shows the tree graph of the graph from Figure 1.1. The sequence of spanning trees $t_1, t_2, \ldots, t_8$ in Figure 1.3 corresponds to a Gray code listing of the spanning trees of the graph from Figure 1.1.

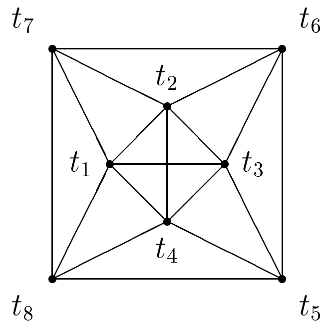

Figure 1.3: The tree graph $T(G)$

A simple path that contains every vertex of a graph is called a *Hamilton path.* A simple cycle that contains every vertex of a graph is called a *Hamilton cycle.* If a graph contains a Hamilton cycle, it is called *Hamiltonian* (see for example [3], page 53). If every edge in G is included in at least one Hamilton cycle, then G is *positively Hamiltonian.* If every edge in G can be avoided in at least one Hamilton cycle, then G is *negatively Hamiltonian.* A graph is *uniformly Hamiltonian* if it is both negatively Hamiltonian and positively Hamiltonian [12].

Cummins [8] proved that if a graph G has more than two distinct spanning trees, then the tree graph $T(G)$ is positively Hamiltonian. If P is a Hamilton path in a tree graph $T(G)$, then the sequence of vertices in P represents a Gray code listing of the spanning trees of G. After the spanning tree corresponding to the origin of P

has been generated, each successive spanning tree can be obtained from the previous spanning tree by an elementary tree transformation. Kamae [13], and later Kishi and Kajitani [15] [16] have developed algorithms for determining Hamilton cycles in a tree graph. These algorithms can be used to list the spanning trees of a graph in a Gray code order. Holzmann and Harary [12] have extended Cummins' result to prove that the tree graph $T(G)$ of a graph G is uniformly Hamiltonian if $T(G)$ contains at least two cycles. If $T(G)$ contains only a single cycle, then $T(G)$ cannot be negatively Hamiltonian since $T(G)$ contains at most one Hamilton cycle.

Since the total time required by an algorithm to generate all spanning trees of a graph increases with the number of spanning trees generated, it is useful to compare algorithms using the average time required per tree. The total time required by the algorithm, divided by the number of spanning trees generated, is referred to as the *amortized time*, or amortized cost, of the algorithm [24]. An algorithm executes in *constant amortized time* (*CAT*) if the total time required by the algorithm, divided by the number of spanning trees generated, is constant. Since only a constant amount of work is allowed (on average) between successive spanning trees, any CAT algorithm must generate the majority of new spanning trees by performing slight modifications to a previously generated spanning tree. Although the Gray code algorithms of Kamae [13] and Kishi and Kajitani [15] generate successive spanning trees by replacing a single edge, neither of these algorithms execute in constant amortized time. Recently, Kapoor and Ramesh [14], and Shioura and Tamura [22] have developed CAT algorithms. These CAT algorithms achieve the theoretical minimum amortized running time for any generation procedure, and are within a constant factor of optimal performance. Neither of these two CAT algorithms generate spanning trees in a Gray code order.

In the following chapters, several previously developed algorithms for generating all spanning trees of a graph are reviewed (Chapter 2), and a new algorithm is developed (Chapter 3). The new algorithm requires $O(n^2)$ space, and generates spanning trees in a Gray code order in constant amortized time (Chapter 4). Both of the previous CAT algorithms require $O(nm)$ space. The time complexities of the previous Gray code algorithms are unknown, but both algorithms require prohibitive space (approximately proportional to the number of spanning trees). A summary of this thesis, and some open problems, are presented in the conclusion (Chapter 5).

Chapter 2

Previous Algorithms

The execution time of combinatorial generation algorithms generally depends upon both the number of objects generated and the size of the object. In the case of spanning tree generation algorithms, execution time typically depends upon both the number of spanning trees generated and the number of vertices and edges in the graph. Let n and m represent the number of vertices and edges, respectively, in the graph for which spanning trees are being generated. The time required to generate all of the spanning trees of a graph G can be expressed as $O(f(n,m)+g(n,m)\tau(G))$, where $f(n,m)$ and $g(n,m)$ are algorithm dependent functions which depend upon the graph size, and $\tau(G)$ is the number of spanning trees in G. Since $\tau(G)$ grows very quickly with graph size, the total running time is usually dominated by the $g(n,m)\tau(G)$ term. In order to compare spanning tree generation algorithms, the amortized cost of the algorithms, $g(n,m)$, is compared.

2.1 Mayeda-Seshu Algorithm

In 1965, Mayeda and Seshu generated all spanning trees of a graph, without generating duplicates, by means of elementary tree transformations [18]. The algorithm of Mayeda and Seshu does not generate spanning trees in a Gray code order, since the backtracking of the algorithm allows successive trees to differ by more than a single elementary tree transformation [18]. Read and Tarjan [20] have studied the algorithm of Mayeda and Seshu and have determined that the algorithm operates in $O(nm)$ amortized time and requires $O(nm)$ space.

The Mayeda and Seshu algorithm (see Figure 2.1) starts with a reference spanning tree, T, of $G = (V, E)$. The graph G and the spanning tree T are global variables. The algorithm labels the edges in the reference tree $E(T) = \{e_1, e_2, \ldots, e_{n-1}\}$, and labels the remaining edges $E - E(T) = \{e_n, e_{n+1}, \ldots, e_m\}$. The edges in T are called reference edges. The reference spanning tree is the first spanning tree generated by the algorithm. Parameter min determines the set of edges, $\{e_{min}, \ldots, e_{n-1}\}$, which may be removed from the current spanning tree T. On the original call to **MS**, the parameter min must be set to 1. During this first call, the algorithm removes edge e_1 from the tree, and replaces it with each edge $e' \in E - T$ which makes $(T - e_1) + e'$ a spanning tree of G. After forming the new tree $(T - e_1) + e'$, **MS** generates all of the trees which include e' but not e_1 by the recursive call **MS**(2). After generating all trees which do not include edge e_1, **MS** generates all trees which do include e_1 by placing e_1 back into the tree, and repeating the process for edges $\{e_2, \ldots, e_{n-1}\}$. If some edge e_k cannot be replaced by any of the edges in the set $\{e_n, \ldots, e_m\}$, then e_k is restored and the algorithm continues by trying to replace edge e_{k+1}.

The amortized time complexity of the Mayeda-Seshu algorithm is determined by the loop at line 2 and the implied loop at line 4, which require $O(nm)$ time. Since it

```
    proc MS(min)
    begin
1     Print T.
2     for each e_i ∈ {e_min, ..., e_{n-1}} do
3       T ← T − e_i
4       L ← {e_j | j ≥ n, T + e_j is a tree}
5       for each e ∈ L do
6         T ← T + e
7         MS(i + 1)
8         T ← T − e
        endfor
9       T ← T + e_i
      endfor
    endproc
```

Figure 2.1: Mayeda-Seshu (**MS**) Algorithm

may not be possible to replace any of the edges in the set $\{e_{min}, \ldots, e_{n-1}\}$ with edges from the set $E - T$, the call to **MS** may result in the generation of a single tree, and so the amortized time complexity of the algorithm is $O(nm)$.

If the loop which is used to construct L is modified so that lines $6-8$ are executed whenever an acceptable edge is found, then L is not required and the loop of lines $5-8$ is not required. This implicit calculation of L reduces the space requirements from $O(nm)$ to $O(n+m)$.

2.2 Minty Algorithm

An algorithm due to Minty [19] (see Figure 2.2) has also been analyzed by Read and Tarjan [20]. The algorithm, **Minty**, maintains a partial spanning tree, T, which is grown one edge at a time. After adding an edge, e, to T, all edges which would form a cycle in T are removed from G. A recursive call then generates all spanning trees which include e and all other edges in T. After recursively generating all spanning trees which contain e, the previously removed edges are restored and e is removed from the graph. After removing e, a search for bridges is performed. Every edge which is a bridge in $G - e$ is added to T at this stage, and all of the spanning trees which do not include e are recursively generated. The algorithm searches for bridges after each edge deletion to ensure that removing any single edge does not disconnect the graph; this guarantees that each call to **Minty** results in the generation of at least one spanning tree. After the recursive call has been completed, G and T are restored to their previous condition by removing the bridges from T and restoring e to the graph G. Prior to calling the routine **Minty**, the original graph G must be checked to ensure that it is connected, and T must be initialized to include all of the bridges of G.

The **Minty** algorithm can be efficiently implemented, and can generate trees in $O(m)$ amortized time. Since each call to **Minty** either generates a spanning tree or results in exactly two recursive calls to **Minty**, the amortized cost of the algorithm is simply the cost of one call (excluding the recursive calls). The search for cycles of $T + e'$ (step 5) requires $O(m(T + e))$ time, but $T + e$ contains exactly n edges, so the search for cycles requires only $O(n)$ time. The search for bridges (step 11) requires $O(m)$ time [23]. Since all other steps in the algorithm require less than $O(m)$ time, the total time required to execute one call is $O(m)$, and the amortized cost of the algorithm is $O(m)$.

The space required by **Minty** can be determined by noting that the number of recursive calls cannot exceed m, and so the space used by the simple local variables is $O(m)$. The two local sets, B and C, can be implemented to use only $O(m)$ space total. If the space used by B and C is released immediately after restoring T and G, respectively, then no edge can exist in more than a single local B set or C set. The

proc **Minty**
begin
  if $(G = T)$ then
    Print T
  else
    Let e be any edge in $G - T$
    $T \leftarrow T + e$
    $C \leftarrow \{e' \in G \mid T + e' \text{ has a cycle}\}$
    $G \leftarrow G - C$
    Minty
    $G \leftarrow G \cup C$
    $G \leftarrow G - e$
    $T \leftarrow T - e$
    $B \leftarrow \{e' \in G - T \mid e' \text{ is a bridge of } G - T\}$
    $T \leftarrow T \cup B$
    Minty
    $T \leftarrow T - B$
    $G \leftarrow G + e$
  endif
endproc

Figure 2.2: **Minty** Algorithm

total space required by **Minty** is therefore $O(m)$.

2.3 Kamae Algorithm

In 1966, Cummins [8] noticed that the tree graph of a graph is always Hamiltonian. Algorithms developed after this time often generate spanning trees by computing the Hamilton cycle in the tree graph. In 1967, Kamae [13] developed an algorithm based on finding a Hamilton cycle in the tree graph.

The Kamae algorithm divides the set of spanning trees of a graph into a number of partitions. The partitions are defined relative to a reference spanning tree, $T = (V(G), \{e_1, e_2, \ldots, e_{n-1}\})$. For each subset $S \subseteq E(T)$, let k be the number of elements in S, and let $I = \{i_1, i_2, \ldots, i_k\}$ represent the indices of the elements of S; $e_i \in S$ if and only if $i \in I$. These subsets of $E(T)$ partition the spanning trees of G. For a given subset S, let T_S consist of the set of trees which include each of the edges $E(T) - S$ and none of the edges of S. The set T_S is called the tree section corresponding to S. Each spanning tree of G is contained in exactly one tree section.

The section graph $SG(G)$ is defined to be the graph which contains the nonempty tree sections of G as its vertices. Two tree sections, T_S and $T_{S'}$ are adjacent in $SG(G)$ if there exists trees $t \in T_S$ and $t' \in T_{S'}$ such that t and t' differ by a single edge. Figure 2.3 represents the section graph of the graph in Figure 1.1. In the left graph of Figure 2.3, the vertex representing tree section T_S is labelled S. In the right graph of Figure 2.3, the vertex representing tree section T_S is labelled with the trees within tree section T_S.

Kamae's algorithm, **Kamae**, generates the spanning trees of G one tree section at a time. The generation can be accomplished so that whenever the algorithm moves from some tree section T_S to $T_{S'}$, the last tree generated in T_S differs from the first

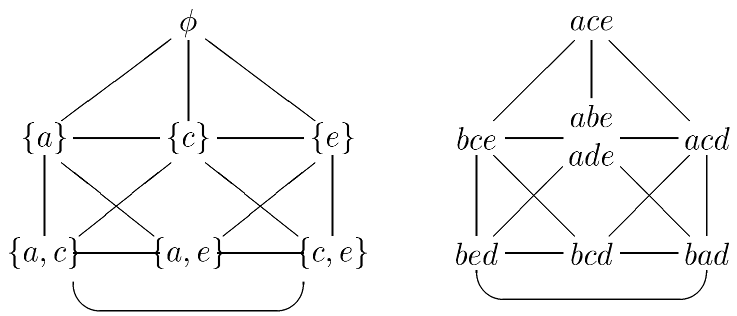

Figure 2.3: Section graph of Figure 1.2

tree in $T_{S'}$ by a single edge. If the algorithm generates the trees within a tree section in Gray code order, then the spanning trees for the entire graph are listed in Gray code order.

Kamae does not provide a complexity analysis of his algorithm, but the algorithm requires space proportional to the number of spanning trees in the largest partition [13]. Since the total number of non-empty partitions is bounded above by 2^{n-1}, we have a simple lower bound on the size of the largest partition of $\tau(G)/2^{n-1}$. Consider K_n, the complete graph on n vertices. The number of spanning trees of K_n is given by Cayley's formula, $\tau(K_n) = n^{n-2}$ [6]. K_n contains an edge between every pair of vertices. Since there are at most 2^{n-1} partitions, one of the partitions of the spanning trees of K_n must contain at least $n^{n-2}/2^{n-1} = \frac{1}{2}(\frac{n}{2})^{n-2}$ spanning trees. This bound can be increased by letting the reference spanning tree consist of the star on n vertices, $t = \{(v_1, v_2), \ldots, (v_1, v_n)\}$. A partition of the set of spanning trees in which each spanning tree contains only a single reference edge (v_1, v_2) will contain $(n-1)^{n-3}$ spanning trees, since any set of $n-2$ edges which spans the vertices $\{v_2, \ldots, v_n\}$ can be added to (v_1, v_2) to form a spanning tree in the partition. The number of sets which span the vertices $\{v_2, \ldots, v_n\}$ is simply the number of spanning trees of K_{n-1}. Using this reference tree results in a space requirement for $\Omega(\tau(G)/n)$ spanning trees.

Assuming that each spanning tree requires $\Theta(n)$ space to store, the **Kamae** algorithm has $\Omega(\tau(G))$ space requirements.

2.4 Berger Algorithm

Later in 1967, Berger [2] developed an algorithm based on the idea of growing the spanning trees from a starting vertex v_1 (see Figure 2.4). Berger's algorithm takes a partial spanning tree T, a set of vertices V_T which are in the spanning tree, and a set of edges E which are not in the spanning tree. Global variables are used to represent these three items. At step k, when new edges and vertices are added to the partial spanning tree, the length of the paths from vertex v_1 to the newly added vertices is k. Berger's algorithm may not be as efficient as the **Minty** algorithm because it is possible for a call to **Berger** to not generate any spanning trees. Algorithm **Berger** must be initially called with $T = \phi$, $V_T = v_1$ and $E = E(G)$.

The space requirements for **Berger** can be determined by noting that the total number of edges contained in all local copies of S cannot exceed m, and that the total number of vertices contained in all local copies of V' cannot exceed n. The total space required by **Berger** is therefore $O(n + m)$.

Due to the large number of subsets generated by line 6, Berger's algorithm exhibits very poor worst case performance. If G consists of the star graph with n vertices, and $n - 1$ edges of the form (v_1, v_j) for $j \in \{2, \ldots, n\}$, then line 6 will generate $2^{n-1} - 1$ subsets of S, but only S itself allows a spanning tree to be generated. If the subsets of S are generated in constant amortized time, then **Berger** executes in $O(2^n)$ time.

```
proc Berger
begin
   S ← {(u, v) ∈ E | u ∉ V_T, v ∈ V_T}
   if S = φ then
      if T spans G then
         Print T
      endif
   else
      E ← E − S
      for each ℓ ∈ {s ⊆ S | s ≠ φ, T ⋃ s is a tree} do
         V' ← {u | u ∉ V_T, (u, v) ∈ ℓ}
         V_T ← V_T ∪ V'
         T ← T ∪ ℓ
         Berger
         T ← T − ℓ
         V_T ← V_T − V'
      endfor
      E ← E ∪ S
   endif
endproc
```

Figure 2.4: **Berger** Algorithm

2.5 Kishi-Kajitani Algorithm

Kishi and Kajitani [15] generate spanning trees by decomposing the tree graph into complete subgraphs, for which Hamilton cycles are easy to generate. Kishi and Kajitani do not analyze the time or space requirements of their algorithm. The Hamilton circuit of the tree graph is built from the union of several smaller Hamilton circuits. The algorithm requires space proportional to the number of spanning trees, so this algorithm also requires a prohibitive amount of space.

The Kishi and Kajitani algorithm, **KK** starts with a pair of adjacent spanning trees, $T_{1,1}$ and $T_{1,2}$. Let $S \leftarrow \{e \mid e \in G, e \notin T_{1,1}, e \notin T_{1,2}\}$. The graph $G - S$ contains a single cycle, and the tree graph of $G - S$ is a complete graph. Let C_1 represent a Hamilton cycle in the tree graph of $G - S$ such that $T_{1,1}$ and $T_{1,2}$ are adjacent.

The algorithm now chooses two trees $T'_{1,1}$ and $T'_{1,2}$ which are adjacent in C_1. If cycle C_1 contains at least three edges, then at least one of $T'_{1,1}$ and $T'_{1,2}$ must be distinct from $T_{1,1}$ and $T_{1,2}$. Two new trees, $T_{2,1}$ and $T_{2,2}$ are now found which are adjacent to $T'_{1,1}$ and $T'_{1,2}$ respectively. These new trees both include an edge from S, which ensures that they did not appear in cycle C_1. These new trees are used as the starting point for finding another complete tree graph and another cycle C_2. The algorithm continues in this fashion until it has generated all of the spanning trees of G.

The set of cycles found by algorithm **KK** for the graph of Figure 1.1 is shown in Figure 2.5. Notice that all of the spanning trees within a cycle differ from each other by an elementary tree transformation. Using the cycles of Figure 2.5, algorithm **KK** could produce the following Gray code list of spanning trees: *abd*, *abe*, *ade*, *ace*, *acd*, *dbc*, *ebc* and *ebd*. Since the algorithm only partially traverses cycle C_k before starting cycle C_{k+1}, at least one tree in each of the cycles must be stored so that it can be listed later. The maximum length of any of the cycles found by **KK** is n, so at least $\tau(G)/n$ spanning trees will need to be stored. If each spanning tree requires $\Theta(n)$ space, algorithm **KK** requires $\Omega(\tau(G))$ space.

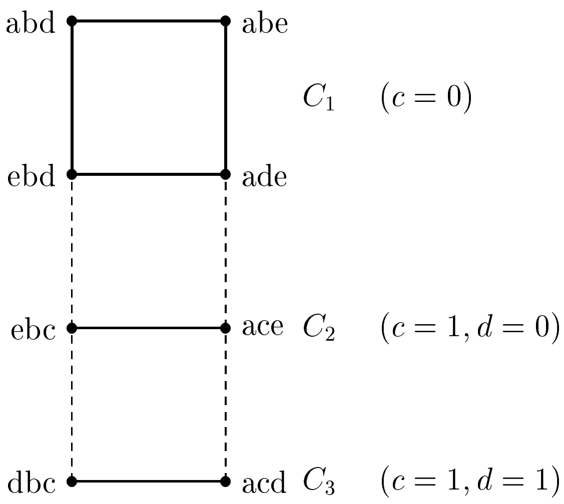

Figure 2.5: Hamilton circuit in tree graph of Figure 1.2

In order to avoid generating duplicates, an inclusion/exclusion principle is used to choose the edges from the set S. In the example in Figure 2.5, $T_{1,1} = abd$ and $T_{1,2} = abe$, so $S = \{c\}$. The cycle C_1 therefore contains all spanning trees which do not contain c. The trees in all cycles except C_1 must contain edge c. The trees in cycle C_2 do not contain edge d, while the trees in cycle C_3 do not contain e. In a graph with m edges and n vertices, each cycle would exclude exactly $m - n$ edges.

2.6 Gabow-Myer's Algorithm

Gabow and Myers [11] have developed an algorithm which requires $O(n)$ amortized time and $O(n+m)$ space (see Figure 2.6). The algorithm generates all spanning trees of a directed graph, so the spanning trees of an undirected graph are generated by treating each undirected edge as a pair of directed edges. Once a root vertex is chosen, each spanning tree of the undirected graph gives rise to exactly one spanning tree of the directed graph. For each spanning tree of the undirected graph, the choice of the root vertex determines the orientation of each of the edges in the corresponding spanning tree of the directed graph.

In 1978, at the time of its development, the Gabow-Myers algorithm, **GM**, was the fastest known spanning tree generation algorithm. Unlike the algorithms **Kamae** and **KK**, the Gabow-Myers algorithm does not generate spanning trees in a Gray code order. Algorithm **GM** uses the following global variables: G represents the current state of the graph, T represents the current partial spanning tree, F is a stack of edges directed from vertices in T to vertices not in T, and L is a copy of the last spanning tree printed. The partial spanning tree $T=(V,E)$ consists of both the vertices and edges which have been added to T. Each time an edge is added to T, the implied vertex is also added. Prior to calling **GM**, T must be initialized to contain a single root vertex r, and F must be initialized to contain all edges (r,v) in G. The local variable FF is used to record the edges which are removed from F, so that F can be restored prior to leaving **GM**.

```
    proc GM
    begin
1     if T spans G then
2        L ← T, Print L
      else
3        repeat
4           Pop an edge e from F
5           Let v ∉ T be the vertex incident with e
6           T ← T + e
7           Push each directed edge (v, w), w ∉ T onto F
8           Remove each directed edge (w, v), w ∈ T from F
9           GM
10          Restore edges removed in line 8
11          Pop the edges pushed in line 7
12          Remove e from T and G, push e onto FF
13          if there is an edge (w, v) where w is not a descendant of v in L then
14             b ← false
            else
15             b ← true
            endif
16       until b
17       Pop each edge e from FF, push e onto F, G ← G + e
      endif
    endproc
```

Figure 2.6: Gabow-Myers (**GM**) Algorithm

2.7 Kapoor-Ramesh Algorithm

The first algorithm for generating spanning trees in constant amortized time was developed by Kapoor and Ramesh [14] in 1992. The algorithm, **KR**, generates the $\tau(G)$ spanning trees of G in $O(n + m + \tau(G))$ time, and requires $O(nm)$ space. The algorithm operates by replacing edges in the current spanning tree. The algorithm does not list spanning trees in a Gray code order since multiple edges may be replaced between successive spanning trees.

An outline of algorithm **KR** is shown in Figure 2.7. Many of the details which are required to implement the algorithm in constant amortized time are omitted from Figure 2.7. These details perform such tasks as collecting parallel edges and allowing the algorithm to treat them as a single edge; removing self-loops; maintaining a version of T in which the IN edges are contracted; and maintaining data structures which allow finding particular edges in constant time. In order to obtain constant amortized time, the algorithms of Chapter 3 perform many tasks which are equivalent to those performed in **KR**. These tasks are discussed in detail in Chapter 3.

Algorithm **KR** initially generates a reference spanning tree T_0. Let the edges of T_0 be referred to as reference edges, and the edges outside of T_0 be referred to as non-reference edges. The algorithm generates all other spanning trees by systematically replacing reference edges with non-reference edges.

In order to avoid generating duplicate spanning trees, **KR** maintains an inclusion and exclusion set of edges. The inclusion set, IN, contains all of the edges which must be included in each of the spanning trees which the call to **KR** generates. The edges in set IN cannot be replaced by non-reference edges. The exclusion set, OUT, contains all of the edges which may not replace edges in the current spanning tree. The addition of an edge e to a set S is represented in the algorithm by $S \leftarrow S + e$

(lines 8, 12 and 14). Similarly, the removal of an edge e from set S is represented by $S \leftarrow S - e$ (line 10). None of the trees which are generated by the current call to **KR** will contain any of the edges in OUT.

Starting from a spanning tree T, **KR** first generates all of the spanning trees which include a new edge, f (lines $1 - 12$). The new edge, f, replaces each of the k edges $\{e_1, \ldots, e_k\}$, in turn, with a recursive call being performed for each of these new spanning trees. After generating all trees which include f, **KR** performs a recursive call to generate all of the trees which exclude f (lines $13 - 15$).

```
      proc KR
      begin
1       f ← an edge in E − T − OUT
2       Determine cycle C ⊆ T + f
3       Let S ← C − IN
4       IN ← IN + f
5       for each e ∈ S do
6         T ← T + f − e
7         Print T
8         OUT ← OUT + e
9         if E − T − OUT ≠ φ then KR
10        OUT ← OUT − e
11        T ← T + e − f
12        IN ← IN + e
        endfor
13      IN ← IN − S
14      OUT ← OUT + f
15      if E − T − OUT ≠ φ then KR
      endproc
```

Figure 2.7: Kapoor-Ramesh(**KR**) Algorithm

2.8 Shioura-Tamura Algorithm

The second algorithm for generating spanning trees in constant amortized time was developed by Shioura and Tamura [22] in 1993. The algorithm, **ST**, generates the $\tau(G)$ spanning trees of G in $O(n+m+\tau(G))$ time, and requires $O(nm)$ space. These time and space requirements are identical to those of algorithm **KR**.

Prior to calling **ST**, a depth-first search of graph G must be performed in order to determine a depth-first spanning tree T_0 of G, and to sort the vertices and edges of G. The edges of G are numbered so that $T_0 = \{e_1, \ldots, e_{n-1}\}$. The reference tree, T_0 is printed prior to the call to **ST**. Algorithm **ST** is initially called as $\mathbf{ST}(n-1)$.

An outline of algorithm **ST** is shown in Figure 2.8. During a call to **ST**, the algorithm first generates all of the spanning trees which do not include edge e_k (lines 2−6), and then generates all spanning trees which do include edge e_k. Since parameter k is initially set to $n-1$ on the first call to **ST**, and each recursive call is of the form $\mathbf{ST}(k-1)$, each of the edges in the spanning tree $T_0 = \{e_1, \ldots, e_{n-1}\}$ is selected as edge e_k.

In the constant amortized time implementation of Shioura and Tamura's algorithm, procedure **ST** is split into two procedures which recursively call each other. One of these procedures corresponds closely to **ST**. The second procedure is used to maintain data structures which allow the determination of which edges can replace edge e_k (line 2 of **ST**).

Algorithm **ST** generates the spanning trees of G by replacing one edge at a time, but the algorithm does not guarantee that successive spanning trees differ by a single edge.

```
proc ST(k)
begin
  if k > 0 then
    for each edge g such that T ∪ g − e_k is a tree do
      T ← T ∪ g − e_k
      Print T
      ST(k − 1)
      T ← T ∪ e_k − g
    endfor
    ST(k − 1)
  endif
endproc
```

Figure 2.8: Shioura-Tamura (**ST**) Algorithm

2.9 Summary

The time and space requirements of the various algorithms are compared in Table 2.1. The only algorithms which have an optimal amortized complexity are due to Kapoor and Ramesh [14] and Shioura and Tamura [22]. These algorithms both execute in constant ($O(1)$) amortized time, and require $O(nm)$ space. The two Gray code algorithms both require a prohibitive amount of space, and are slower than the $O(n)$ amortized time algorithm of Gabow and Myers [11]. Chapter 3 presents a constant amortized time algorithm which generates spanning trees in Gray code order. The new Gray code algorithm, **BEST**, requires $O(n^2)$ space and $O(n^2 + \tau(G))$ time.

Algorithm	Year	Space	Amortized Time	Gray Code
Mayeda-Seshu	1965	$O(n+m)$	$O(nm)$	No
Minty	1965	$O(n+m)$	$O(m)$	No
Berger	1967	$O(n+m)$	$O(2^n)$	No
Kamae	1967	$\Omega(\tau(G))$	$\Omega(n)$	Yes
Kishi-Kajitani	1968	$\Omega(\tau(G))$	$\Omega(n)$	Yes
Gabow-Myers	1978	$O(n+m)$	$O(n)$	No
Kapoor-Ramesh	1992	$O(nm)$	$O(1)$	No
Shioura-Tamura	1993	$O(nm)$	$O(1)$	No
BEST	1995	$O(n^2)$	$O(1)$	Yes

Table 2.1: Comparison of Previous Generation Algorithms

Chapter 3

New Work

Most algorithms for generating spanning trees use an inclusion-exclusion principle; these algorithms generate all spanning trees which include a particular edge and then generate all spanning trees which exclude the edge. A natural method of implementing this type of inclusion-exclusion principle is to perform contractions and deletions on the graph. Graph algorithms based on contractions are often thought to be inefficient due to the cost of the contraction operations and the necessity of determining which connected component contains a particular vertex [10]. In this chapter, we present a series of improvements to a simple contraction-deletion based spanning tree generation algorithm which result in an algorithm which generates spanning trees in a Gray code order in constant amortized time.

3.1 Introduction

Perhaps the simplest algorithm for generating all spanning trees of a graph is the contraction-deletion algorithm. This algorithm recursively generates all spanning trees of G which include some edge $e \in G$, then recursively generates all spanning trees which do not include e. As the name suggests, the two graph operations used in this algorithm are edge contraction and edge deletion. In the contraction-deletion algorithm, the spanning trees which contain edge e correspond to the spanning trees of $G \bullet e$; the spanning trees which exclude edge e correspond to the spanning trees of $G - e$ (see Theorem 1.1).

An outline of the basic algorithm is presented in more detail in Figure 3.1. The function **Gen** returns **true** if and only if it generates at least one spanning tree. It is assumed that the current graph, G, and the current partial spanning tree (a spanning forest), T, are both global variables. **Gen** uses several routines which are specified in the following paragraphs. Since the number of vertices in the graph G changes as the algorithm performs contractions, let N be the number of vertices in the original graph. Similarly, let M be the number of edges in the original graph. The notations $n(G)$ and $m(G)$ are the number of vertices and edges, respectively, in the graph represented by the global variable G.

The variables *Selected* and TE are used in the following manner. TE is used to store the tree edges in the T, but it also contains edges from the previous tree. During the recursive call at a specific *level*, *Selected*[*level*] indicates which element of TE is to be used to store the edge under consideration, e. If edge e is contracted, then e is stored in $TE[\mathit{Selected}[\mathit{level}]]$ (see line 6). If $G - e$ contains spanning trees, then e is deleted, and $\mathit{Selected}[\mathit{level} + n(G) - 1] \leftarrow \mathit{Selected}[\mathit{level}]$ so that the final edge added during the generation of the first spanning tree of $G - e$ is stored in TE at position *Selected*[*level*]. Prior to the initial call to **Gen**, *Selected* should be initialized to $\{1, 2, \ldots, N-1\}$.

The first routine used by **Gen** is the **RejectGraph** function. In order for **Gen** to operate correctly, the **RejectGraph** function must satisfy two requirements. The

```
func Gen(level) : boolean
begin
  if RejectGraph() then
    Gen ← false
  else if n(G) = 2 then
    for e ← each of the remaining non-loop edges do
      T ← T + e
      TE[Selected[level]] ← e
      PrintSpanningTree(1)
      T ← T − e
    endfor
    Gen ← true
  else
    SelectEdge(e, level)
    G ← G •e,  T ← T + e
    PostContractionProcessing(e, level)
    success ← Gen(level + 1)
    UndoPostContractionProcessing(e, level)
    Gen ← success
    Restore G and T to state prior to contraction.
    if success then
      G ← G − e
      bridge ← PostDeletionProcessing(e)
      if not bridge then
        Selected[level + n(G) − 1] ← Selected[level]
        success ← Gen(level + 1)
      endif
      UndoPostDeletionProcessing(e)
      G ← G + e
    endif
  endif
endfunc
```

Figure 3.1: Function **Gen**

first requirement is that graphs with exactly two vertices are correctly classified; if $n(G) = 2$, then **RejectGraph** returns **true** if and only if G contains no spanning trees ($\tau(G) = 0$). The second requirement is that graphs which contain spanning trees are never rejected; **RejectGraph** returns **true** only if $\tau(G) = 0$.

A simple version of the **RejectGraph** function is given in Figure 3.2. In this simple version, **RejectGraph** only rejects graphs with more than two vertices (lines $1-4$) if $m < n-1$. In a more sophisticated version of **RejectGraph**, line 4 could be replaced with code which rejected some (or all) of the disconnected graphs. If line 4 returns **false** for a disconnected graph, then **Gen** attempts to generate spanning trees of G. Since G is disconnected, G does not contain any spanning trees, and **Gen** returns the value **false**. The unsuccessful attempt to generate spanning trees of G does not cause **Gen** to produce incorrect results, but does make **Gen** less efficient. The problem of performing recursive calls which do not generate any spanning trees is addressed in Section 3.7.

```
     func RejectGraph : boolean
     begin
1      if m(G) < n(G) − 1 then
2         RejectGraph ← true
3      else if n(G) > 2 then
4         RejectGraph ← false
5      else if G is disconnected then
6         RejectGraph ← true
       else
7         RejectGraph ← false
       endif
     endfunc
```

Figure 3.2: A simple version of **RejectGraph**

The **PrintSpanningTree** procedure prints the current spanning tree, T. In more efficient versions of **Gen**, the call to the **PrintSpanningTree** procedure is replaced with a call to a revised **PrintSpanningTree** procedure which may print several different spanning trees as the result of a single call from **Gen**. The recursive **PrintSpanningTree** versions of Figures 3.8 and 3.17 each print more than one spanning tree per call from **Gen**. These versions, **PrintSpanningTree2** and **PrintSpanningTree3**, use a parameter which is equal to the depth of recursion. Since the depth of recursion is always 1 immediately after the call from **Gen**, the **PrintSpanningTree** procedures is called with the parameter value 1.

The **SelectEdge** procedure selects a non-loop edge to contract and delete. A simple version of **SelectEdge** is given in Figure 3.3. An edge selection rule which results in a Gray code ordering of the spanning trees is presented in Section 3.3.

proc **SelectEdge**($e, level$)
begin
   Let $e \in G$ be the lexicographically smallest non-loop edge in G
endproc

Figure 3.3: A simple version of **SelectEdge**

The **PostContractionProcessing** and **PostDeletionProcessing** procedures allow more sophisticated algorithms to be represented within the framework of Figure 3.1. The **PostContractionProcessing** procedure operates on the graph after the edge contraction; the **PostDeletionProcessing** function operates on the graph after the edge deletion. In the basic algorithm, the **PostContractionProcessing** procedure does not do anything. In more efficient algorithms, **PostContractionProcessing** performs simple operations on the graph in an attempt to improve the efficiency of the algorithm (see Figures 3.6 and 3.12). The **PostDeletionProcessing** function returns **true** if the deletion of e made either of its incident vertices isolated (see Figure 3.4). This allows **Gen** to avoid some unnecessary recursive calls.

In more efficient algorithms, the **PostDeletionProcessing** procedure may be more sophisticated (see Figure 3.14). The details of these procedures will be discussed in later sections. The procedures **UndoPostContractionProcessing** and **UndoPostDeletionProcessing** restore various data structures to their states prior to the respective calls to **PostContractionProcessing** and **PostDeletionProcessing**.

```
func PostDeletionProcessing(e) : boolean
begin
  Let u and v be the vertices that were incident with e.
  if u or v is an isolated vertex then
    PostDeletionProcessing ← true
  else
    PostDeletionProcessing ← false
  endif
endfunc
```

Figure 3.4: A simple version of **PostDeletionProcessing**

The support procedures are discussed in greater detail in later sections, while the operation of **Gen** itself is described in the following paragraphs. The **Gen** procedure has been organized to allow different versions of the support routines to be used. Revised versions of the support routines will contain a revision number at the end of their name. For example, the second and third versions of the **PrintSpanningTree** routine will be referred to as **PrintSpanningTree2** and **PrintSpanningTree3**.

When **Gen** is called, it first checks to see if the graph can be rejected. If **RejectGraph** returns a **true** value, then the graph does not contain any spanning trees, and **Gen** can immediately return with the value **false**. If G contains only two vertices, then each of the remaining non-loop edges is, one at a time, included in the partial spanning tree T, and each resulting tree is printed (lines $4-8$). Since $n(G)=2$, the

false value returned by **RejectGraph** ensures that $\tau(G) > 0$, so **Gen** returns the value **true** (line 9).

If $n(G) > 2$, then **Gen** calls **SelectEdge** in order to choose an edge, e, for contraction and possible deletion (line 10). After the edge e is contracted, it is added to the partial spanning tree T (line 11). In this basic algorithm, the **PostContractionProcessing** procedure does not have any effect, so it can be ignored (line 12). All spanning trees of graph G which contain edge e are generated by the recursive call at line 13.

Since the graph G contains at least one spanning tree if and only if $G \bullet e$ contains at least one spanning tree, the value received from the recursive call at line 13 is the value which must be returned by **Gen** (line 15). After generating the spanning trees of $G \bullet e$, the graph is restored to its state prior to the contraction, and edge e is removed from the partial spanning tree T (line 16).

If $G \bullet e$ contains at least one spanning tree, then G is connected. If G is connected, then $G - e$ is disconnected if and only if e is a bridge. If the removal of e creates an isolated vertex, then e is a bridge, $\tau(G - e) = 0$, **PostDeletionProcessing** returns **true**, and the call to **Gen** is not necessary. If **PostDeletionProcessing** returns **false**, then it is not known if e is a bridge, and $G - e$ may or may not be connected. The spanning trees of $G - e$, if they exist, are generated by line 22. Line 21 is required so that the edge which replaces e overwrites e in the array TE. After the recursive call, the graph is restored to its state prior to the call to **PostDeletionProcessing** (line 23). If **PostDeletionProcessing** did not modify G, then **UndoPostDeletionProcessing** has no effect. Finally, edge e is restored to the graph (line 24).

Figure 3.5 represents a computation tree of the basic **Gen** framework when the **SelectEdge** procedure selects the lexicographically smallest edge at each step. This simple selection rule does not result in the spanning trees being generated in a Gray code order.

Gen stores the graph as an array of adjacency list records. Each adjacency list record contains pointers to the first and last edge records in the double-linked adjacency list. After the contraction of an edge, (u, v), the adjacency lists of u and v must be combined. The head and tail pointers for the adjacency lists allow efficient (constant time) concatenation of adjacency lists. Each edge (u, v) in the undirected graph G is represented by the pair of edge records (u, v) and (v, u). Each edge record, (u, v), also contains a pointer to its *mate*, (v, u). If an edge (u, v) is contracted or deleted, the mate pointer allows the edge (v, u) to be found in constant time.

3.2 Removing Loops in $G \bullet e$

Although the basic **Gen** algorithm of Figure 3.1 correctly generates all of the spanning trees of a graph, a large number of repetitive operations are performed. Prior to the contraction of an edge (u, v), it is possible that both u and v are adjacent to some common vertex w. After (u, v) is contracted, the edges corresponding to (u, w) and (v, w) are parallel. The repetitive operations arise when these parallel edges are selected for contraction and deletion.

Let e and f be two parallel edges of G, and assume that the algorithm, as described in Figure 3.1, selects edge e for contraction. When e is contracted, f becomes a loop of $G \bullet e$ and is effectively ignored within the first recursive call (which uses graph $G \bullet e$). After the recursive call is completed, e is uncontracted, and the graph $G - e$ is formed. Edge f is not a loop in $G - e$, and so f may be chosen for contraction during the second recursive call. If f is contracted immediately after the deletion of e, then the graph $(G - e) \bullet f$ differs from the graph $G \bullet e$ by only the loop f. Since loops

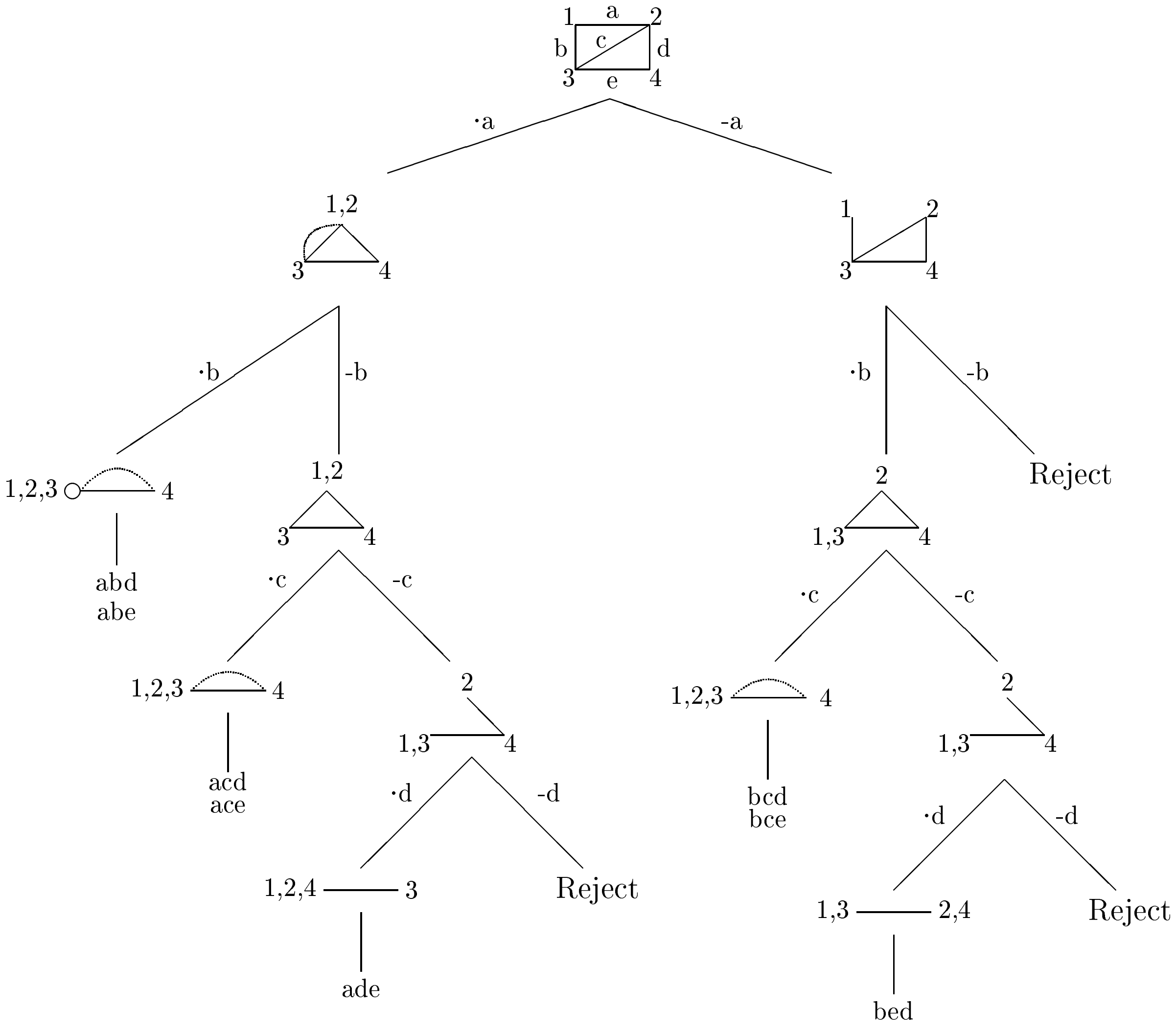

Figure 3.5: Computation tree for **Gen**

are never selected as edges to add to the partial spanning tree, $G \bullet e$ and $(G-e) \bullet f$ both contain the same number of spanning trees. Each spanning tree of $(G-e) \bullet f$ is a spanning tree of $G \bullet e$. Let T_e represent any spanning tree of G which includes edge e and is therefore generated by the recursive call which operates on graph $G \bullet e$. The spanning tree $T_f = (T_e - e) + f$ is a spanning tree which contains edge f and is generated by the recursive call which operates on graph $(G-e) \bullet f$. If the recursive call on $(G-e) \bullet f$ is not performed, the spanning trees which include f (but not e) can be obtained from the spanning trees which contain e by replacing edge e with edge f. This eliminates a substantial number of recursive calls to **Gen** and improves the efficiency of the algorithm.

Figure 3.5 illustrates the redundancy performed by the basic algorithm using the graph of Figure 1.1. After the contraction of edge a, edges b and c are parallel. The sequences of operations $(\bullet a, \bullet b)$ and $(\bullet a, -b, \bullet c)$ produce graphs which differ by only a single loop. The set spanning trees generated by the sequence of operations $(\bullet a, \bullet b)$ and those generated by the sequence $(\bullet a, -b, \bullet c)$ are identical except that each of the spanning trees in the first set contains edge b, while the trees of the second set contain edge c. The trees can be more efficiently listed by generating the trees which contain a and b, and for each of these trees, generating a second tree by replacing edge b with the parallel edge c.

For a general graph, G, with parallel edges e and f, an efficient method of generating the spanning trees of both $G \bullet e$ and $(G-e) \bullet f$ is to generate the spanning trees for both graphs simultaneously. For each spanning tree T_e of $G \bullet e$ which is printed, the spanning tree $(T_e - e) + f$ is also printed. Since the spanning trees of $G \bullet f$ are generated by the same recursive call which generates the spanning trees of $G \bullet e$, there does not need to be any explicit contraction/deletion of edge f. It can simply be removed from G when e is contracted or deleted.

This strategy is efficiently implemented by having the **PostContractionProcessing2** procedure store a list consisting of the contracted edge along with all loops which resulted from the contraction. The loops can then be removed from the graph. When a spanning tree is printed, **PrintSpanningTree2** can use these lists and replace the contracted edges with each of the edges with which they were parallel. The **PostContractionProcessing2** and **PrintSpanningTree2** procedures are described in Figures 3.6 and 3.8 respectively. The **PrintSpanningTree2** procedure traverses these lists in such a fashion that a single call of **PrintSpanningTree2** from **Gen** results in the printed spanning trees being listed in a Gray code order (see Section 3.3 for details).

```
    proc PostContractionProcessing2(e, level)
    begin
1     Let v be the supervertex resulting from the contraction of e.
2     Let L be the list of loops incident with v.
3     if L ≠ φ then
4       ParallelCount ← ParallelCount + 1
5       G ← G − L
6       PEL[ParallelCount] ← e + L
7       PLevel[ParallelCount] ← level
8       ParallelContraction[level] ← true
      else
9       ParallelContraction[level] ← false
      endif
    endproc
```

Figure 3.6: Procedure **PostContractionProcessing2**

The **PostContractionProcessing2** procedure constructs a list, L, of the loops which result from the contraction of edge e (line 2). If L is not empty, then the number of parallel edge lists that are stored is incremented (line 4), the parallel edges (loops) are removed from G (line 5), and the list L and the edge e are stored in the PEL array (line 6). The $PLevel$ array records which recursive call to **Gen** resulted in the formation of the parallel edges. The boolean array element $ParallelContraction[level]$ records whether any loops are created by the contraction of e (line 8 or 9).

proc **UndoPostContractionProcessing2**$(e, level)$
begin
1 if $ParallelContraction[level]$ then
2 $G \leftarrow G \bigcup (PEL[ParallelCount] - e)$
3 $ParallelCount \leftarrow ParallelCount - 1$
endif
endproc

Figure 3.7: Procedure **UndoPostContractionProcessing2**

The **UndoPostContractionProcessing2** procedure restores the graph G and counter $ParallelCount$ to their states prior to the call to **PostContractionProcessing2**. This is accomplished by restoring the deleted loops (line 2), and implicitly removing the list of parallel edges from PEL (line 3). If no loops were formed by the contraction of e, then $ParallelContraction[level]$ is **false**, **PostContractionProcessing2** did not remove any loops, and the **UndoPostContractionProcessing2** simply exits. Edge e is not added to G in line 2 since e is not included in the graph $G \bullet e$.

The lists of parallel edges which are stored by the various calls to **PostContractionProcessing2** are used by the **PrintSpanningTree2** procedure to generate multiple trees. The first line of **PrintSpanningTree2** ensures that there is another list of parallel edges to be dealt with. If there are no more parallel edge lists, then the spanning tree currently stored in T is printed (line 2). If there is a list of parallel edges, then the unique edge in $PEL[lev] \bigcap T$ must be either the first edge in the list or the last edge. If the first (last) edge in $PEL[lev]$ is in tree T, then lines $5-8$ $(10-13)$ generate all of the spanning trees which contain each of the edges

proc **PrintSpanningTree2**(lev)
begin
  if $lev > ParallelCount$ then
    Print spanning tree T
  else
    Let $e_1, e_2, \dots e_k$ represent list $PEL[lev]$
    if $e_1 \in T$ then
      $e_0 \leftarrow e_1$
      for $i \leftarrow 1$ to k do
        $T \leftarrow (T - e_{i-1}) + e_i$
        PrintSpanningTree2($lev + 1$)
      endfor
      $Selected[PLevel[lev]] \leftarrow e_k$
    else
      $e_{k+1} \leftarrow e_k$
      for $i \leftarrow k$ downto 1 do
        $T \leftarrow (T - e_{i+1}) + e_i$
        PrintSpanningTree2($lev + 1$)
      endfor
      $Selected[PLevel[lev]] \leftarrow e_1$
    endif
  endif
endproc

Figure 3.8: Procedure **PrintSpanningTree2**

in list $PEL[lev]$. When **PrintSpanningTree2** returns, either the first or last edge of $PEL[lev]$ will be stored in T, which allows the next call at level lev to perform correctly. For compatibility with the revision of Section 3.3, **PrintSpanningTree2** prints spanning trees in a Gray code order. The *Selected* array is modified (line 9 or 14) to ensure that it correctly represents the edges which are stored within tree T. The *Selected* array is used by the **SelectEdge** procedure of Section 3.3.

If the work performed within the loops of the **PrintSpanningTree2** procedure is charged to the recursive calls, then each call of **PrintSpanningTree2** either prints a spanning tree, or performs only a constant amount of work. Since there are always at least two edges in each PEL list, the **PrintSpanningTree2** computation tree contains more leaf nodes than interior nodes. Each leaf node of the computation tree corresponds to a new spanning tree being printed. If each interior node is paired with a distinct leaf node, then each of these pairs of nodes prints a spanning tree but performs only a constant amount of work (excluding the actual printing of the spanning tree). **PrintSpanningTree2** therefore executes in constant amortized time, and does not need to be considered during the analysis of the generation algorithm. Notice that **PrintSpanningTree2** could be modified so that after printing the first spanning tree, only the edges e_{i-1} and e_i which changed were printed. This would allow the actual printing to also execute in constant amortized time.

Figure 3.9 represents a computation tree of the improved algorithm. Notice that after the contraction of a, both the contraction and deletion of b also result in the removal of edge c. Since edges b and c are parallel after the contraction of edge a, **Gen** treats them as equivalent edges, and the call to **PrintSpanningTree2** lists the spanning trees which include edges a and c as well as the spanning trees which include edges a and b. Although the reduction in the number of calls to **Gen** is quite small in this example, when this strategy is applied to larger graphs, a greater number of

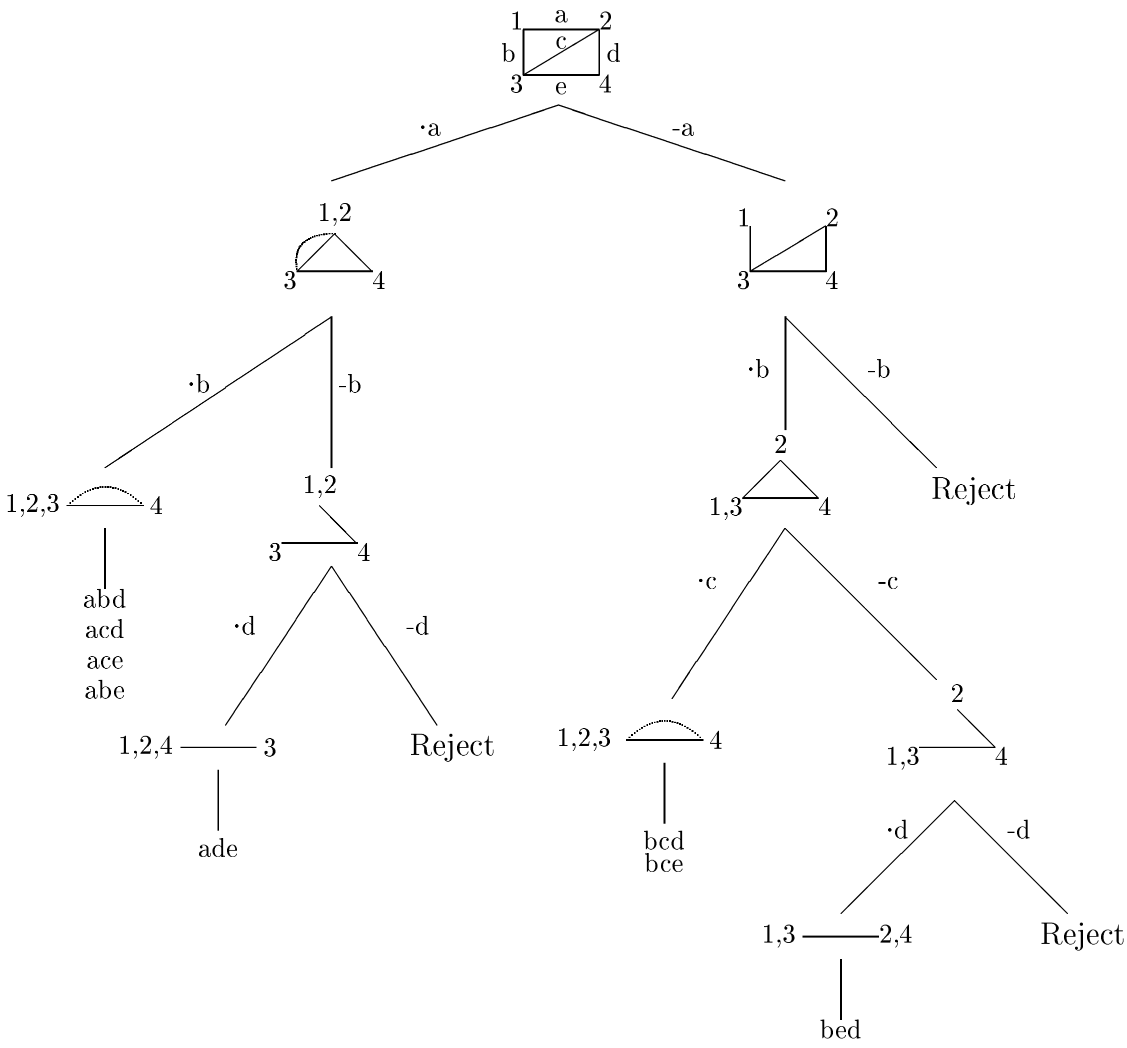

Figure 3.9: Computation tree for **Gen** (with loop removal)

parallel edges are formed, and a greater number of recursive calls to **Gen** are avoided.

3.3 A Gray Code Algorithm

Holzmann and Harary [12] have proven that if the tree graph of a graph contains at least two cycles, then the tree graph is uniformly Hamiltonian. Their proof is based on the partitioning of the tree graph into spanning trees which include a particular edge and spanning trees which do not include the edge. Since their proof is based on an inclusion-exclusion principle, this implies the existence of a contraction-deletion based algorithm for generating spanning spanning trees in Gray code order.

A simple Gray code algorithm can be constructed by using the basic algorithm of Figure 3.1 along with an appropriate **SelectEdge** procedure. This Gray code algorithm is not an implementation of the Holzmann-Harary proof; the algorithm generates a Hamilton path within the tree graph, but the first and last spanning trees do not necessarily differ by a single edge.

Figure 3.10 describes the **SelectEdge2** procedure which results in the generation of spanning trees in a Gray code order. Prior to calling **Gen**, it is necessary to initialize the first $n-2$ elements of the TE array to $n-2$ distinct edges of a reference spanning tree. During the generation of the first spanning tree, the $n-2$ calls to **SelectEdge2** ensure that the $n-2$ edges in the TE array are chosen. The final edge in the reference spanning tree is one of the edges selected by the for loop (lines $4-8$) of **Gen**.

Theorem 3.1 **Gen**, *when used with* **PrintSpanningTree2** *and* **SelectEdge2**, *generates spanning trees in a Gray code order.*

proc **SelectEdge2**($e, level$)
begin
  $e \leftarrow TE[Selected[level]]$
endproc

Figure 3.10: Procedure **SelectEdge2**: (Gray Code version)

Proof: To prove that **Gen**, when used with **PrintSpanningTree2** and **SelectEdge2**, generates spanning trees in a Gray code order, it is sufficient to prove that any successive pair of spanning trees differs by a single edge. There are three possible ways for two consecutive spanning trees to be printed by **Gen**. Both trees could be printed by the same call to **PrintSpanningTree2**, they could be printed by consecutive calls to **PrintSpanningTree2** from within the loop (lines $4-8$) in **Gen**, or the first tree could be printed during the contraction of an edge e and the second tree could be printed during the deletion of the edge e.

If both of the spanning trees are printed by the same original call to **PrintSpanningTree2**, the two trees differ by the single edge which was changed on the recursive call. When **PrintSpanningTree2** is called twice from within the loop (lines $4-8$) of **Gen**, **Gen** has placed a new edge into the spanning tree, and the second call to **PrintSpanningTree2** simply determines which edge in each list was in the previous tree. This is efficiently accomplished by the alternating forwards and backwards traversals of the PEL lists in the **PrintSpanningTree2** procedure.

If the two spanning trees result from two recursive calls, **Gen**$(G \bullet e)$ and **Gen**$(G - e)$, then let T_e be the last spanning tree generated by the **Gen**$(G \bullet e)$ and let $T_{\bar{e}}$ be the first spanning tree generated by the recursive call **Gen**$(G - e)$. In order for the spanning trees to be generated in a Gray code order, T_e and $T_{\bar{e}}$ must differ by an elementary tree transformation. Let $e_1, e_2, \ldots, e_{n-1}$ represent the edges in T_e. Let

k be chosen so that $e = e_k$. The edges $e_1, e_2, \ldots, e_{k-1}$ are common to both T_e and $T_{\bar{e}}$ since these are included in T before edge e is considered. During the generation of $T_{\bar{e}}$, **SelectEdge2** chooses all of the remaining edges of T_e except $e = e_k$. The edge e in T and TE is replaced by the first edge processed by the for loop of **Gen**. During the call to **Gen**(G), after the deletion of e, the location of e within TE ($Selected[level]$ is stored in $Selected[level + n(G) - 1]$ so that the final edge added to $T_{\bar{e}}$ overwrites e in TE. In the last call to **Gen** within the generation of $T_{\bar{e}}$, the for loop of **Gen**, overwrites the edge e in TE so that a later call to **SelectEdge2** can select this new edge. The two spanning trees T_e and $T_{\bar{e}}$ both contain all of the edges $\{e_1, e_2, \ldots, e_{k-1}, e_{k+1}, \ldots, e_{n-1}\}$, and therefore differ by a single edge.

Since any two successive spanning trees differ by an elementary tree transformation, **Gen**, when used with **PrintSpanningTree2** and **SelectEdge2**, generates spanning trees in a Gray code order. □

If e is a bridge, then $G - e$ does not contain any spanning trees, and **Gen** contracts edges from $T_e - e$ until the graph is rejected. When **Gen** fails to generate new spanning trees after the removal of a bridge, the algorithm backtracks and again tries to contract edges from the previous spanning tree until it either generates a new spanning tree or again fails. Failing to generate a new tree leads to further backtracking, and the process is repeated until the algorithm generates a new tree, or there are no more trees to be generated. During the backtracking, selecting edges which are included in the most recently generated spanning tree ensure that the Gray code property holds, but there may be many calls to **Gen** which do not generate any spanning trees. This inefficiency is addressed in Section 3.7.

In order to develop a more efficient Gray code algorithm, it is important to include the loop removal discussed in Section 3.2. **PrintSpanningTree2** always traverses the PEL lists in either forwards or reverse order; when a recursive call is completed,

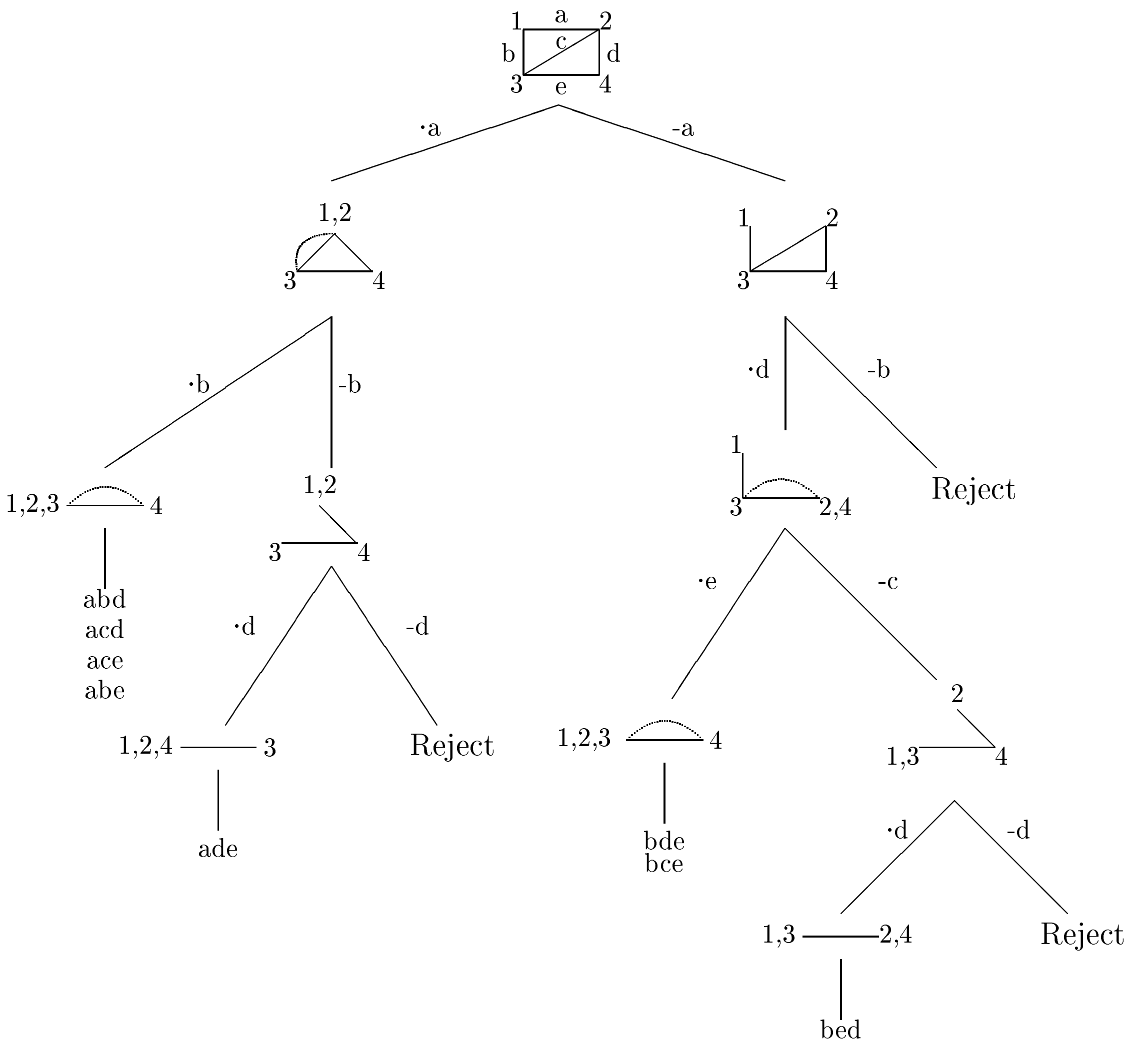

Figure 3.11: Computation tree for **Gen** (Gray code version)

T contains either the first or last edge from each PEL list. It is important that **PrintSpanningTree2** only needs to test whether the first or last edge in a PEL list is included in the tree, since these two tests can be performed in constant time.

3.4 Replacing Parallel Edges

In Section 3.2, the post-contraction processing performed by procedure **PostContractionProcessing2** results in the removal of all edges which are parallel to the recently contracted edge. This results in a more efficient generation algorithm since many of the calls to **Gen** can be eliminated.

Another benefit of removing the loops after a contraction is the reduction in the amount of work required for one call to **Gen**. Since the edges in the graph are stored in adjacency lists, it is advantageous to keep the adjacency lists as short as possible. When an edge (u, v) is contracted, it is necessary to traverse the adjacency list of either u or v to determine which edges become loops. Traversing the shorter of the two lists requires examination of $O(min(d(u), d(v)))$ edges, where $d(x)$ is the degree of vertex x.

Theorem 3.2 *Using the loop removal scheme of Section 3.2, the maximum size of an adjacency list is $O(M)$.*

Proof: Since the total number of edges in the graph is M, no adjacency list can contain more than M edges. To see that an adjacency list can contain $O(M)$ edges, consider the complete graph, K_N. Assume that during the generation of the first spanning tree, an edge is always chosen which is incident with vertex 1. The number of edges in the adjacency list for vertex 1 is originally $N-1$. After the first contraction, there are $(N-1)+(N-1)-2 = 2N-4$ edges in the adjacency list. A second contraction results in an adjacency list containing $(2N-4)+(N-1)-4=3N-9$ edges.

Assume that after $k-1$ contractions there are $kN-k^2$ edges in the adjacency list for vertex 1. Then the next contraction adds $N-1$ new edges, but there are k parallel edges connecting the new vertex to the component containing vertex 1. The number of edges in the adjacency list after contraction k is therefore $(kN-k^2)+(N-1)-2k$, which can be rewritten as $(k+1)N-(k^2-2k-1)$, or simply $(k+1)N-(k+1)^2$. The size of the adjacency list reaches a maximum of approximately $N^2/4$ after $\lfloor N/2 \rfloor$ contractions. Since $N^2/4 \approx m(K_N)/2$, the length of a single adjacency list may become $O(M)$. $\square$

In order to keep the adjacency lists as short as possible, pairs of parallel edges are replaced by a single *parallel edge* record. Whenever a pair of parallel edges is formed as a result of a contraction, this pair of parallel edges is replaced with a single edge which contains a list of the actual edges it represents. The construction of a parallel edge record results in one less edge record being stored in the adjacency lists which represent the graph. Since the construction of a parallel edge record results in a single edge record replacing two edge records, the number of parallel edge records cannot exceed $m(G)$. Since each record only requires a constant amount of space, the use of parallel edge records requires no more than $O(M)$ space.

During the contraction of an edge (u,v), the adjacency lists for u and v are merged. Two edges, (u,x) and (v,y) are parallel if vertices x and y are connected in the partial spanning tree T. Using the standard union/find approach to merging connected components, determining if x and y are in the same connected component requires $O(\lg n)$ time [1] [7]. In order to avoid this $O(\lg n)$ lookup time for each edge, each edge will contain a pair of vertex stacks which contain the label of the connected component (supervertex) on the top. After edge (u,v) is contracted, if u is the remaining supervertex, then all edges incident with vertex v will have u pushed onto the appropriate vertex stack.

In order to efficiently detect parallel edges, the adjacency lists store edges in sorted order. The merging of two adjacency lists will take time $O(d(u)+d(v))$, during which time the vertex stacks are updated and any parallel edges are detected. Each pair of parallel edges is replaced with a parallel edge record at this stage.

If parallel edges are always represented by a single parallel edge record, then the length of an adjacency list can never exceed the number of vertices in the graph. As the algorithm performs contractions, the number of vertices in the graph, and therefore the maximum length of adjacency lists, is reduced. Each time **Gen** chooses a non-bridge edge for contraction and deletion, two recursive calls are performed. This branching of the computation tree results in most recursive calls using much smaller graphs than the original. Since the graph contains one less vertex after each contraction, most calls to **Gen** contain much shorter adjacency lists than the original graph. The use of parallel edge records also ensures that the number of edge records in a graph with n vertices never exceeds $\binom{n}{2}$.

Parallel edges can only occur as the result of a contraction, so it is only necessary to modify the **PostContractionProcessing** and **UndoPostContractionProcessing** procedures to ensure that parallel edges are always stored in parallel edges records. In the **PostContractionProcessing3** procedure (see Figure 3.12), the formation of the parallel edge records (line 2) is performed during the merging of the adjacency lists. This requires keeping the adjacency lists sorted and correctly merging the lists during a contraction.

Since there are $O(m)$ edges, and a vertex stack could contain n vertices, it appears that $O(mn)$ space is required by the vertex stacks. However, there are never more than $n-1$ active contractions at any moment, and the length of the adjacency lists

```
proc PostContractionProcessing3(e, level)
begin
  Let u and v be the vertices which were incident with e
     (where v is the supervertex remaining after the contraction)
  Sort adjacency list v by merging the old adjacency list v
     with the adjacency list u, forming parallel edge records
     as they are discovered, and pushing v onto the vertex stack
     of each non-parallel edge from adjacency list u
  if e is a parallel edge then
     ParallelCount ← ParallelCount + 1
     PEL[ParallelCount] ← e
     PLevel[ParallelCount] ← level
     ParallelContraction[level] ← true
  else
     ParallelContraction[level] ← false
  endif
endproc
```

Figure 3.12: Procedure **PostContractionProcessing3**

at each contraction are bounded by $n-1$. Each contraction therefore results in at most $n-1$ vertices being pushed onto the vertex stacks, and so the total size of the vertex stacks is bounded by $O(n^2)$.

```
   proc UndoPostContractionProcessing3(e, level)
   begin
1    Let u and v be the vertices incident with e
       (where v is the vertex remaining after the contraction)
2    Replace parallel edge records with parallel edge pairs.
3    For each non-parallel edge originally in the adjacency list for u,
       pop the v vertex from of the vertex stack
4    Restore adjacency lists u and v to state prior to
       call to PostContractionProcessing3
5    if ParallelContraction[level] then
6      ParallelCount ← ParallelCount − 1
     endif
   endproc
```

Figure 3.13: Procedure **UndoPostContractionProcessing3**

In the **UndoPostContractionProcessing3** procedure (see Figure 3.13), the adjacency list which is formed by the contraction is scanned, and any parallel edges which are created by the contraction of edge e are replaced with the equivalent pair of parallel edges (line 2). Any non-parallel edges which were in adjacency list u have the v supervertex removed from their vertex stack (line 3). The adjacency list is then restored to its state prior to the call to **PostContractionProcessing3** (line 4). This restoration consists of separating the two adjacency lists which were merged during the **PostContractionProcessing3** procedure. If the contracted edge is a parallel edge, then the edge is implicitly removed from the PEL array by decrementing $ParallelCount$ (lines $5-6$).

3.5 Replacing Series Edges

The replacement of parallel edges with single parallel edge records is an effective method for controlling the size of adjacency lists. The method is effective because each of the edges within a parallel edge record can be treated identically. If the parallel edge record is contracted, the edge is later expanded by the **PrintSpanningTree2** procedure.

Another case where a pair of edges can be treated as a single edge is when they are both incident with the same degree two vertex. Let $P = v_0, e_1, v_1, \ldots, e_k, v_k$ be a path in G. If $v_1, \ldots, v_{k-1}$ are each degree two vertices, then every spanning tree of G will either include all of the edges $e_1, \ldots, e_k$, or will exclude exactly one of them. To see that only a single edge may be excluded, consider two edges, e_i and e_j, where $i < j$. If both e_1 and e_j are excluded from a spanning tree T, then the vertices $v_i, \ldots, v_{j-1}$ are not connected to the rest of T, so T is not a spanning tree of G.

Consider the graph, C_N, a cycle on N vertices. The removal of any one of the N edges results in a spanning tree, so $\tau(C_N) = N$. Since no loops or parallel edges are formed by the contraction of edges in C_N, the basic algorithm of Section 3.1 performs as well as the later variations.

If we assume that the **SelectEdge** procedure selects an edge incident with a degree one vertex, if such an edge exists, then **Gen** would operate as follows. Once a single deletion has occurred, the algorithm makes $n(G)-1$ calls to **Gen** in order to generate the single spanning tree. Since an edge which is incident with a degree one vertex must be a bridge, and the algorithm can determine this in constant time, these edges are not deleted. Let C_N represent the graph consisting of the cycle $v_1, e_1, v_2, e_2, \ldots, v_N, e_N, v_1$. Let $\bar{C}_{N-1}$ represent the graph $C_N \bullet e_1$, and define $\bar{C}_{N-i} = \bar{C}_{N-i+1} \bullet e_i$. Edge deletions only occur on the subgraphs $C_N, \bar{C}_{N-1}, \ldots, \bar{C}_3$. Since $\bar{C}_2$ contains only two vertices, the edge deletion does not occur. Let $W(G)$ represent the total number of calls to **Gen** during the generation of all spanning trees of G. For the cycle C_N, we have

$$W(C_N) = \sum_{j=3}^{N}(j-1) = \frac{N(N-1)}{2} - 1$$

Since the number of spanning trees is $\tau(C_N) = N$, and the number of calls to **Gen** is approximately $N^2/2$, the amortized cost in terms of calls per spanning tree is approximately $N/2$. This poor performance is due to the $n(G)-1$ calls which are required to generate the single spanning tree of a path.

To improve the performance of the generation algorithm on graphs containing long paths (sequences of degree two vertices), these paths can be replaced with a single series edge record. If a single series edge record, S, is constructed which represents $n-1$ of the edges of C_N, then the resulting graph contains only 2 vertices, and two parallel edges. During the construction of the series edge, all but one of the edges of S are stored in the spanning tree. In addition to containing a list of all of the edges in the path, the series edge record also keeps track of which edge is excluded from the current spanning tree. Each edge record which corresponds to a single edge in the original graph records its location within T (if it is included in T), so that it can be replaced without requiring a search of T. If S is contracted, then the excluded edge is added to T.

Assume that S represents k edges in series, and that edge e_1 is not included in the current partial spanning tree, but that $e_2, \ldots, e_k$ are included. If S is deleted, then each of the k edges in S would, one at a time, be the excluded edge. This is efficiently performed by replacing e_2 with e_1, then replacing e_3 with e_2, and continuing the process until each of the edges in S has been excluded. This process is performed by the **PrintSpanningTree3** procedure (see Figure 3.17).

The creation of a series edge record from the two edges incident with a degree two vertex effectively merges the two edges, so this process is called *vertex contraction*. Each series edge record is the result of a vertex contraction, so each series edge record corresponds to at least two original edges. A series edge record corresponds to more than two original edges when one of the merged edges was a series or parallel edge record.

```
func PostDeletionProcessing2(e) : boolean
begin
   Let u and v be the vertices that are incident with e.
   if u or v is an isolated vertex then
      PostDeletionProcessing ← true
   else
      PostDeletionProcessing ← false
      ReplaceSeriesEdges(u, v)
   endif
endfunc
```

Figure 3.14: Procedure **PostDeletionProcessing2**

The **PrintSpanningTree3** procedure uses an array, Q, of parallel/series edge records. When **PrintSpanningTree3** adds parallel or series edge records to Q (lines 5, 8, 13 and 16), they are added to the end of Q. Lines $6 - 12$ selects each of the parallel edges of L, one at a time, and includes it in the tree. In order for the spanning trees to be generated in Gray code order, the first edge selected by the loop must be the unique edge in L which was included in the most recently printed spanning tree. Line 9 adds the edge e to tree T; if e is a series edge, then the single excluded edge in e is added to T, otherwise e is an original edge and it is added to T. Similarly, lines $14 - 21$ select which of the series edges of L are to be excluded in the spanning tree. As in the parallel edge case, the first edge excluded by the loop must be the unique

proc **ReplaceSeriesEdges**(u, v)
begin
  if $d(u) = 2$ then
    Let (u, x_1) and (u, x_2) be the edges incident with u.
    Replace (u, x_1) and (u, x_2) with series edge record (x_1, x_2).
    $T \leftarrow T + (u, x_1)$
    Remove u from G.
    ReplaceParallelEdges$((x_1, x_2))$
  endif
  if $d(v) = 2$ then
    Let (v, y_1) and (v, y_2) be the edges incident with v.
    Replace (v, y_1) and (v, y_2) with series edge record (y_1, y_2).
    $T \leftarrow T + (v, y_1)$
    Remove v from G.
    ReplaceParallelEdges$((y_1, y_2))$
  endif
endproc

Figure 3.15: Procedure **ReplaceSeriesEdges**

proc **ReplaceParallelEdges**$((u, v))$
begin
  if (u, v) is parallel to another edge $e \in G$ then
    Replace (u, v) and e with a parallel edge record.
    ReplaceSeriesEdges(u, v)
  endif
endproc

Figure 3.16: Procedure **ReplaceParallelEdges**

```
     proc PrintSpanningTree3(lev)
     begin
1      if lev > QNext then
2        Print spanning tree T
       else
3        L ← Q[lev]
4        if L is a parallel list then
5          Put all series edges within L into Q
6          for each e ∈ L do
7            if e is a series edge then
8              Remove e from Q, and put all parallel edges within e into Q
             endif
9            T ← T + e
10           PrintSpanningTree3(lev + 1)
11           if e is a series edge then
12             Restore Q to state prior to line 8
             endif
           endfor
         else (L is a series list)
13         Put all parallel edges within L into Q
14         for each e ∈ L do
15           if e is a parallel edge then
16             Remove e from Q, and put all series edges within e into Q
             endif
17           Let f ∈ L be the unique edge which is not in T
18           T ← (T + f) − e
19           PrintSpanningTree3(lev + 1)
20           if e is a parallel edge then
21             Restore Q to state prior to line 16
             endif
           endfor
         endif
22       Restore Q to state at entry
       endproc
```

Figure 3.17: Procedure **PrintSpanningTree3**

edge in L which was excluded by the most recently printed spanning tree. Line 18 removes the edge e from tree T; if e is a parallel edge, then the single included edge in e is removed from T, otherwise e is an original edge and it is removed from T.

Since parallel edge records are recorded during edge contractions, and series edge records are recorded during edge deletions, both types of edge records are stored in the array Q. The **PostContractionProcessing3** and **UndoPostContraction-Processing3** routines are revised so that they use the Q array rather than the PEL array.

3.6 Preprocessing

One of the most effective tools in the design of efficient algorithms is the principle of divide and conquer. If a problem can be divided into two or more smaller problems, and if the solutions to these smaller problems can be combined to form the solution to the original problem, then this may lead to a more efficient algorithm.

Theorem 3.3 *If a graph G consists of two connected components, G_1 and G_2, joined by a bridge, b, then $\tau(G) = \tau(G_1)\tau(G_2)$.*

Proof: If T_1 and T_2 are spanning trees of G_1 and G_2, respectively, and $b = (x, y)$ is the bridge connecting the two connected components, then $T = (T_1 \bigcup T_2) + b$ is a spanning tree of G. To see that T is a spanning tree of G, consider any pair of vertices $u, v \in G$. If u and v are both contained in the same connected component, then u and v are (by definition) connected. Assume, without loss of generality, that $u, x \in G_1$ and $v, y \in G_2$. Since T_1 is a spanning tree of G_1, there is a path in T_1 connecting u and x. Similarly, there is a path in T_2 connecting v and y. These two paths, plus the bridge (x, y) constitute a path in T which connects u and v, so T is a connected spanning subgraph of G. Since T_1 is a tree, and T_2 is a tree, T cannot contain a cycle because there is only one edge connecting T_1 and T_2. Therefore, T is a spanning tree of G. For each of the $\tau(G_1)$ spanning trees of G_1, any of the $\tau(G_2)$ spanning trees of G_2 may be chosen to form a spanning tree of T. The total number of spanning trees of T is therefore $\tau(G) = \tau(G_1)\tau(G_2)$. □

The first step of preprocessing consists of finding all bridges of G, and dividing the graph into connected components after removing the bridges from G and storing them in T. This preprocessing step can be performed in $O(M)$ time (see for example [23]).

The second preprocessing step is the formation of parallel and series edge records. The formation of these edge records reduces the effective size of the graph and thereby reduces the amount of computation required to generate the spanning trees for the graph. The formation of parallel edge records may result in new degree two vertices; the formation of series edge records may result in new parallel edges. This preprocessing will reduce G to as small a graph as possible by performing vertex contractions and replacing parallel edges with parallel edge records. The effects of this preprocessing can be very spectacular. For example, both the ladder graph on n vertices, L_n, and the cycle graph on n vertices, C_n, are reduced to graphs which contain only two vertices and two parallel edges. This preprocessing step is applied to each of the connected components created in the first preprocessing step, and is described in Figure 3.18.

In procedure **ReduceGraph**, the construction of the adjacency matrix (line 1) requires $O(n^2)$ time. Each entry, $M[i, j]$, in the adjacency matrix will contain either a pointer to the edge (i, j), or the nil pointer if G does not contain edge (i, j). Counting the degree of each vertex can be performed in $O(m)$ time using the adjacency list of each vertex (lines $2 - 3$). The creation of each series record decreases the number

```
    proc ReduceGraph(G)
    begin
1     Create an adjacency matrix M of G
2     for each v_i ∈ V do
3       degree[i] ← degree of v_i
      endfor
4     while there are degree two vertices do
5       Let v ∈ G be one of the degree two vertices
6       Let x and y be the vertices adjacent to v
7       Contract vertex v
8       ReplaceParallelEdges((x, y))
      endwhile
    endproc
```

Figure 3.18: Procedure **ReduceGraph**

of vertices in the graph representation by one, so a maximum of $n - 1$ series edge records can be formed. Similarly, the creation of each parallel edge record decreases the number of edges in the graph representation by one, so a maximum of $m - 1$ parallel edge records can be formed. Since the formation of any series or parallel edge record requires only a constant amount of work, and at most $m - 1$ series or parallel edge records can be formed, lines $4 - 8$ require $O(m)$ time. The total amount of time required by **ReduceGraph** is therefore $O(n^2)$.

The use of these two steps of preprocessing allows the actual generation algorithm to perform local processing only. When an edge (u, v) is contracted, only the adjacency lists of u and v need to be examined for the formation of parallel edges. Similarly, when an edge (u, v) is deleted, only the incident vertices, u and v, need to be examined to determine if a vertex contraction can be performed. The contraction of u or v may result in the formation of parallel edges, and the formation of these parallel edges may in turn lead to the formation of degree two vertices. The algorithm alternately forms parallel edge records and series edge records until either the graph contains only two vertices, or no more parallel/series edge records can be formed.

Another benefit of this preprocessing stage is that each vertex of the reduced graph G will have degree at least three. Empirical evidence suggests that **Gen** generates spanning trees at a faster rate when the graph is dense.

3.7 A BEST Modification

Ruskey [21] has defined a backtracking algorithm to have the *BEST* property if no unsuccessful recursive calls are made. In a BEST algorithm, **B**acktracking **E**nsures **S**uccess at **T**erminal leaves of the computation tree. **Gen** does not have the BEST property since it is possible that $G-e$ does not contain any spanning trees. In order to ensure that the algorithm always succeeds at terminal leaves, it is necessary to avoid making a recursive call after removing a bridge. If the **PostDeletionProcessing2** routine is revised so that it returns **true** if and only if the deleted edge is a bridge of G, then **Gen** would have the BEST property. Determining if e is a bridge of G is equivalent to testing if the graph $G - e$ is disconnected, which can be performed in $O(m)$ time by a simple depth first search of $G - e$.

If e is a bridge, it may be more efficient to break the graph into two connected components G_1 and G_2 such that $G = (G_1 \bigcup G_2)+e$. The algorithm can then generate the spanning trees of G_1 and G_2 independently. Since this algorithm differs significantly from the **Gen** algorithm, this new algorithm is called the **BEST** algorithm. Let G_1 and G_2 be labelled so that $m(G_1) \leq m(G_2)$. If the test that e is a bridge is performed immediately after the edge e is selected, then the depth first search can provide a list of the vertices and edges within each of the components G_1 and G_2. If e is a bridge, then **BEST** performs a recursive call using G_1 as the graph G, and performs a recursive call with G_2 as the graph G in place of the call to **PrintSpan-**

```
     proc BEST(level)
     begin
1      if n(G) = 2 then
2        if e is a parallel edge then
3          Add e to Q.
         else
4          T ← T + e
         endif
5        if T does not span the original graph then
6          Pop(n(G), m(G), level).
7          BEST(level + 1)
8          Push(n(G), m(G), level).
         else
9          PrintSpanningTree3(1)
         endif
10       Restore Q or T (undo line 3 or 4)
       else
11       SelectEdge2(e, level)
12       if IsBridge(e) then
13         T ← T + e
14         G ← G − e
15         Rearrange edges in Selected
16         Push (n(G_2), m(G_2), level + m(G_1))
17         G ← G − G_2
18         BEST(level + 1)
19         Restore component stack to state prior to line 15
20         G ← G ⋃ G_2
21         G ← G + e
22         T ← T − e
         else
23         Recursively generate trees of G •e and G − e (as in Gen).
         endif
       endif
     endproc
```

Figure 3.19: Algorithm **BEST**

ningTree3. The recursive call to **BEST** which processes G_2 itself either divides the graph G_2 into two connected components (if a bridge is selected) or directly calls **PrintSpanningTree3** when only two vertices remain.

In algorithm **BEST**, there may be more than one component to process. If **BEST** is processing the last component, it calls **PrintSpanningTree3** (line 9), otherwise it continues with the next component (lines $6-8$). In line 6, the size of the next component is popped off the component stack, along with the correct *level* to ensure that the edges chosen during the call to **BEST** (line 7) are from the connected component G_2. If G contains more than two vertices, and e is not a bridge, then the spanning trees are generated for $G \bullet e$ and $G-e$ in a fashion similar to **Gen** (line 23). If e is a bridge, then e is added to T (line 13) and removed from G (line 14). The edges in the *Selected* array are then rearranged to ensure that the next $n(G_1)-1$ edges selected are from the component G_1, and the edges in the *Selected* array are moved down the array so that the first edge of G_2 is located at position $level + m(G_1)$ (line 15). The size of the larger component of G is then pushed onto the stack, along with an offset into the *Selected* array which allows the edges and the smaller component is processed (line 16). After the recursive call, the graph G, the tree T and the stack are all restored to their state at the start of the call (lines $18-22$).

The removal of component G_2 from G (line 17) is performed implicitly by adjusting the size of $n(G)$ and $m(G)$. Since the edges selected for contraction/deletion during the processing of G_1 are always chosen from G_1 (due to the rearrangement of *Selected* in line 15), G_2 is not explicitly removed from G. Each adjacency list of G_1 is identical to the corresponding adjacency list in $G_1 \bigcup G_2$, so only $n(G)$ and $m(G)$ are modified to ensure that $G = G_1$. In line 20, the graph G is restored to $G = G_1 \bigcup G_2$ by restoring $n(G)$ and $m(G)$ to their values prior to line 17.

Chapter 4

Analysis

In this chapter, the time complexity of algorithm **BEST** is determined. The other algorithms of Chapter 3 are not analyzed since they perform calls to **Gen** which do not generate any spanning trees.

The time spent in the routine **PrintSpanningTree3** is not considered in the analysis since the procedure requires $O(\tau(G))$ if the trees are not explicitly printed. If the actual printing time is ignored, the **PrintSpanningTree3** procedure executes in constant amortized time since the amount of work performed by any call is proportional to either the number of recursive calls or the number of spanning trees printed.

4.1 A Lower Bound on $\tau(G)$

In order to analyze the running time of **BEST**, it is necessary to obtain a lower bound on the number of spanning trees of a simple graph. The following bound is quite loose, but proves to be sufficient for the analysis.

Theorem 4.1 *If G is a simple connected graph with m edges and n vertices, then G contains at least $\tau(G) \geq 3 + 7\alpha/3 + 2\alpha^2/3 + \alpha^3/3$ spanning trees, where $\alpha = m - n$.*

Proof: Let T represent an arbitrary spanning tree of G, and let $S = \{e \mid e \in G, e \notin T\}$ represent the set of edges not included in T. Let $\alpha = m - n = |S| - 1$.

Let τ_k represent the number of spanning trees of G which contain exactly k edges from S. If none of the edges in S are included, then T is the only spanning tree, so $\tau_0 = 1$.

Consider the number of spanning trees which contain exactly one of the edges in S. It is well known that each edge $e \in S$ can be added to T to form a graph with a unique cycle (see for example [3], page 29, Theorem 2.5). Since there are no parallel edges in G, the length of the shortest possible cycle is three. The removal of any edge, f, in the cycle results in a spanning tree. If $f \neq e$, then the resulting spanning tree contains exactly one edge from S. Since there are at least two choices for f, the number of spanning trees of G which contain exactly one edge from S is $\tau_1 \geq 2(m - n + 1) = 2(\alpha + 1)$.

Now consider the number of spanning trees which contain exactly two of the edges in S. Let e_1 and e_2 represent the two edges from S which will be included in the new spanning tree. Then there exist edges $f_1, f_2 \in T$ such that $(T - f_1) + e_1$ and $(T - f_1 - f_2) + e_1 + e_2$ are spanning trees of G. There are at least two choices for the edge f_1, but there may be only a single choice for edge f_2 since the cycle in $(T - f_1) + e_1 + e_2$ may contain only three edges, including both e_1 and e_2. The number of spanning trees of G which contain exactly two edges in S is therefore

$$\tau_2 \geq 2\binom{m-n+1}{2} = (m-n+1)(m-n) = \alpha^2 + \alpha$$

The number of spanning trees which contain exactly three edges in S can be determined by counting the number of edge replacements which make a tree containing two edges from S into a tree containing three edges from S. Consider any of the τ_2 spanning trees which contain exactly two edges $e_1, e_2 \in S$. Let e_3 be the edge which will be added to form the new tree. If $\{e_1, e_2, e_3\}$ form a cycle, then e_3 cannot be used to form any new spanning trees; if $\{e_1, e_2, e_3\}$ do not form a cycle, then at least one new tree is formed. Since there is at most one edge e_3 which forms a cycle with e_1 and e_2, there are $m-n-2$ edges which do not form a cycle. Consider an arbitrary spanning tree which includes exactly three edges $x, y, z \in S$. This spanning tree is counted once with x in the e_3 position, a second time with y in the e_3 position and a third time with z in the e_3 position. The number of spanning trees of G which contain exactly three edges in S is therefore

$$\tau_3 \geq \tau_2(m-n-2)/3 = (\alpha^2+\alpha)(\alpha-2)/3.$$

Since a spanning tree of G may contain up to $n-1$ edges from S, the total number of spanning trees of G is

$$\tau(G) = \tau_0 + \tau_1 + \tau_2 + \ldots + \tau_{n-1}.$$

If this sum is truncated after the τ_3 term, then a lower bound on $\tau(G)$ is obtained.

$$\tau(G) \geq 1 + 2(\alpha+1) + (\alpha^2+\alpha) + (\alpha^2+\alpha)(\alpha-2)/3$$

After simplifying this sum, the following bound on $\tau(G)$ is obtained

$$\tau(G) \geq 3 + 7\alpha/3 + 2\alpha^2/3 + \alpha^3/3. \qquad \square$$

Lemma 4.1 *If G is a connected graph with $\delta(G) \geq 3$, then $\tau(G) > 3+7n/6+nm/9+nm^2/54$.*

Proof: From the previous theorem (Theorem 4.1), after replacing α with $m-n$, the bound on $\tau(G)$ becomes

$$\tau(G) \geq 3 + 7(m-n)/3 + 2(m-n)^2/3 + (m-n)^3/3.$$

By the assumption that $\delta(G) \geq 3$, $m-n$ has the following lower bounds: $m-n \geq n/2$ and $m-n \geq m/3$. Replacing one $m-n$ factor with $n/2$ in each term, and any remaining $m-n$ factors with $m/3$, the bound on $\tau(G)$ becomes

$$\tau(G) > 3 + 7n/6 + nm/9 + nm^2/54.$$

□

4.2 An Upper Bound on Leaves

Theorem 4.2 *Let $L(G)$ be the number of terminal calls to* **BEST** *performed, during generation of all spanning trees of G by* **BEST**. *If G is a connected graph with minimum degree three, then $L(G) \leq \frac{1}{2}\tau(G)$.*

Proof: We will prove that every leaf of the computation tree of **BEST** represents at least two spanning trees, and therefore $\tau(G) \geq 2L(G)$.

Let ℓ be any leaf in the computation tree, and let $\mathcal{P}$ be the parent of node ℓ. Similarly, let $\mathcal{Q}$ be the parent of node $\mathcal{P}$. Let G_ℓ, $G_\mathcal{P}$ and $G_\mathcal{Q}$ represent the graph G within the calls corresponding to ℓ, $\mathcal{P}$ and $\mathcal{Q}$.

The graph $G_\mathcal{P}$ has minimum degree $\delta(G_\mathcal{P}) \geq 3$. If $G_\mathcal{P}$ had a degree one or degree two vertex, then the graph would have been collapsed by the **PostContraction-Processing3** routine called from $\mathcal{Q}$. The **PostContractionProcessing3** routine removes degree one and degree two vertices until either the graph contains only two vertices or the graph has minimum degree three. Since $\mathcal{P}$ is not a leaf of the computation tree, $G_{\mathcal{P}}$ must have more than two vertices and therefore has minimum degree three.

Since $\delta(G_{\mathcal{P}}) \geq 3$, we know that $n(G_{\mathcal{P}}) \geq 4$ and $m(G_{\mathcal{P}}) \geq \frac{3}{2}n(G_{\mathcal{P}}) \geq n(G_{\mathcal{P}}) + 2$. Since $\mathcal{P}$ is the parent of leaf ℓ, we know that the leaf must generate all of the spanning trees corresponding to either $G_{\mathcal{P}} \bullet e$ or $G_{\mathcal{P}} - e$. From Theorem 4.1, we have the simplified lower bound $\tau(G) \geq 3 + 2(m - n)$. From the previous bounds on $n(G_{\mathcal{P}})$ and $m(G_{\mathcal{P}})$, and the simplified bound on $\tau(G)$, we have the following lower bounds:

$$\tau(G_{\mathcal{P}} \bullet e) \geq 3 + 2(m(G_{\mathcal{P}} \bullet e) - n(G_{\mathcal{P}} \bullet e)) \geq 3 + 2 * 2 = 7$$

$$\tau(G_{\mathcal{P}} - e) \geq 3 + 2(m(G_{\mathcal{P}} - e) - n(G_{\mathcal{P}} - e)) \geq 3 + 2 * 1 = 5$$

Since the leaf ℓ must produce at least five spanning trees, we have $\tau(G) \geq 5L(G)$ and therefore $L(G) \leq \frac{1}{2}\tau(G)$. □

4.3 Analysis of BEST

The following analysis of algorithm **BEST** assumes that the following operations are performed:

1. Preprocessing in time $O(n^2)$ to ensure G is connected and $\delta(G) \geq 3$.
2. Parallel edge records are formed as soon as possible. This ensures that $m < \binom{n}{2}$.
3. Series edge records are formed as soon as possible, to ensure that $\delta(G) \geq 3$ remains true.

4. When a bridge is selected, the component with the fewest edges is processed first.

In order to show that **BEST** performs in constant amortized time, it is necessary to determine the amount of work performed in both the case that the selected edge is a bridge and also when the selected edge is not a bridge.

In the more simple case, when the selected edge is not a bridge, the amount of work performed by **BEST** can easily seen to be bounded by

$$W(G) \leq W(G \bullet e) + W(G - e) + C^* m$$

where C^* is a positive constant.

When the selected edge is a bridge, the amount of work performed by **BEST** can easily seen to be bounded by

$$W(G) \leq W(G_1) + L(G_1)W(G_2) + A' L(G_1) + C' m$$

where A' and C' are positive constants.

Theorem 4.3 *Given a connected graph G with $\delta(G) \geq 3$, the amount of time required by* **BEST** *to generate the $\tau(G)$ spanning trees of G, $W(G)$, is bounded by*

$$W(G) < (A + 2C)\left(\tau(G) - \frac{1}{9}mn\right),$$

for some constants $A \geq A'$ and $C \geq \max(C^*, C')$.

Proof (by induction on n):

Basis: Consider all connected graphs with $\delta(G) \geq 3$, and $n < 12$. Using Lemma 4.1, we know that $\tau(G) - mn/9 > 0$ for each of these graphs, so it is possible to choose the constants A and C such that the theorem holds.

Induction: Assume that G is a connected graph with $\delta(G) \geq 3$ and $n(G) \geq 12$. The amount of work performed by **BEST** is dependent upon the type of edge which is processed at each step.

Case 1: Let us first consider the case where the selected edge, e, is not a bridge. We can use the previous bound on $W(G)$.

$$W(G) \leq W(G \bullet e) + W(G - e) + C^* m$$

If we apply the inductive hypothesis, and use the fact that $C \geq C^*$, we have

$$\begin{aligned} W(G) &< (A+2C)\left(\tau(G \bullet e) - \frac{1}{9} n(G \bullet e) m(G \bullet e)\right) \\ &\quad + (A+2C)\left(\tau(G-e) - \frac{1}{9} n(G-e) m(G-e)\right) + Cm. \end{aligned}$$

Adding together the $\tau(G \bullet e)$ and $\tau(G-e)$ terms to form $\tau(G)$, the bound becomes

$$\begin{aligned} W(G) &< (A+2C)\tau(G) + Cm \\ &\quad - (A+2C)\left(\frac{1}{9} n(G \bullet e) m(G \bullet e)\right) \\ &\quad - (A+2C)\left(\frac{1}{9} n(G-e) m(G-e)\right). \end{aligned}$$

Since $n(G \bullet e) = n - 1$ and $m(G \bullet e) = m - 1$,

$$\begin{aligned} W(G) &< (A+2C)\tau(G) + Cm \\ &\quad - (A+2C)\frac{1}{9}\left((n-1)(m-1) + n(G-e)m(G-e)\right). \end{aligned}$$

Since $n(G-e) \geq n-2$ and $m(G-e) \geq m-3$,

$$\begin{aligned}
W(G) &< (A+2C)\tau(G) + Cm \\
&\quad -(A+2C)\frac{1}{9}\left((n-1)(m-1) + (n-2)(m-3)\right) \\
&= (A+2C)\left(\tau(G) - \frac{1}{9}(2nm - 3m - 4n + 7)\right) + Cm \\
&= (A+2C)\left(\tau(G) - \frac{1}{9}nm\right) \\
&\quad -\frac{1}{9}(A+2C)(nm - 3m - 4n + 7) + Cm \\
&< (A+2C)\left(\tau(G) - \frac{1}{9}nm\right) \\
&\quad -\frac{1}{9}C(2nm - 15m - 8n + 14)
\end{aligned}$$

Since $n \geq 12$ and $m \geq \frac{3}{2}n \geq 18$, we know that $2nm - 15m - 8n + 14 > 0$, and so

$$W(G) < (A+2C)\left(\tau(G) - \frac{1}{9}nm\right).$$

Case 2: Let us now consider the case where the selected edge e is a bridge. If e is a bridge, then G is divided into two graphs, G_1 and G_2 such that $G = (G_1 \bigcup G_2) + e$. In the following, we use the notations n_i, m_i and τ_i as a convenient shorthand for $n(G_i)$, $m(G_i)$ and $\tau(G_i)$ respectively. Recall the previous bound on $W(G)$,

$$W(G) \leq W(G_1) + L(G_1)W(G_2) + A'L(G_1) + C'm.$$

Applying the inductive hypothesis, and using the fact that $A \geq A'$ and $C \geq C'$, we have

$$W(G) < (A+2C)\left(\tau_1 - \frac{1}{9}n_1m_1\right)$$

$$
\begin{aligned}
&+L(G_1)(A+2C)\left(\tau_2 - \frac{1}{9}n_2 m_2\right) \\
&+L(G_1)A + Cm
\end{aligned}
$$

Since $L(G_1) < \frac{1}{2}\tau(G_1) = \frac{1}{2}\tau_1$, we have

$$
\begin{aligned}
W(G) &< (A+2C)\left(\tau_1 - \frac{1}{9}n_1 m_1\right) + \frac{1}{2}\tau_1 A + Cm \\
&\quad + \frac{1}{2}\tau_1 (A+2C)\left(\tau_2 - \frac{1}{9}n_2 m_2\right) \\
&= \frac{1}{2}\tau_1\tau_2(A+2C) + \frac{1}{2}\tau_1 A + Cm \\
&\quad +(A+2C)\left(\tau_1 - \frac{1}{9}n_1 m_1 - \frac{1}{18}\tau_1 n_2 m_2\right) \\
&= (A+2C)\left(\frac{1}{2}\tau + \tau_1 - \frac{1}{9}n_1 m_1 - \frac{1}{18}\tau_1 n_2 m_2\right) \\
&\quad + \frac{1}{2}\tau_1 A + Cm \\
&< (A+2C)\left(\frac{1}{2}\tau + \frac{3}{2}\tau_1 + \frac{1}{2}m - \frac{1}{9}n_1 m_1 - \frac{1}{18}\tau_1 n_2 m_2\right) \\
&< (A+2C)\left(\frac{1}{2}\tau + \frac{3}{2}\tau_1 + \frac{1}{2}m\right)
\end{aligned}
$$

Since $\tau_1 \leq \frac{\tau}{16}$

$$
\begin{aligned}
W(G) &< (A+2C)\left(\frac{1}{2}\tau + \frac{3}{32}\tau + \frac{1}{2}m\right) \\
&= (A+2C)\left(\tau - \frac{13}{32}\tau + \frac{1}{2}m\right)
\end{aligned}
$$

From Lemma 4.1, we know that $\tau(G) \geq \frac{1}{9}nm + \frac{1}{54}nm^2$, so the bound becomes

$$
\begin{aligned}
W(G) &< (A+2C)\left(\tau - \frac{13}{32}\left(\frac{1}{9}nm + \frac{1}{54}nm^2\right) + \frac{1}{2}m\right) \\
&= (A+2C)\left(\tau - \frac{13}{288}nm - \frac{13}{32}\left(\frac{m}{54}\right)nm + \frac{1}{2}m\right).
\end{aligned}
$$

Using the lower bounds of $n \geq 12$, and $m \geq 18$, the inequality becomes

$$\begin{aligned} W(G) &< (A+2C)\left(\tau - \frac{13*12}{288}m - \frac{13}{32}\left(\frac{m}{54}\right)nm + \frac{1}{2}m\right) \\ &= (A+2C)\left(\tau - \frac{52}{96}m - \frac{13}{96}nm + \frac{1}{2}m\right) \\ &< (A+2C)\left(\tau - \frac{13}{96}nm\right) \\ &< (A+2C)\left(\tau - \frac{1}{9}nm\right) \square \end{aligned}$$

Theorem 4.4 *Given a graph G, then algorithm* **BEST**, *when coupled with preprocessing, and performing the operations specified in Theorem 4.3 executes in $O(n^2 + \tau(G))$ time.*

Proof: Let G' be the representation of the graph G after preprocessing. The preprocessing ensures that either $\delta(G') \geq 3$ or G' includes fewer edges than K_4. The preprocessing of G requires $O(n^2)$ time since each vertex must be examined. If a vertex has degree two, then the vertex will be contracted and the adjacency lists of the incident vertices will be merged in $O(n)$ time. During the merging, parallel edge records will replace any pairs of parallel edges. If the vertex has degree greater than two, then the adjacency list of the vertex will be scanned for parallel edges.

Theorem 4.3 ensures that $W(G')$ is $O(\tau(G'))$, but $\tau(G') = \tau(G)$, so the amount of time required by **BEST** is $O(\tau(G))$. The total time required by both preprocessing and the generation procedure **BEST** is therefore $O(n^2 + \tau(G))$. $\square$

Chapter 5

Conclusion

A new Gray code algorithm, **BEST**, for generating all spanning trees of an undirected graph has been developed. The algorithm executes in constant amortized time and requires only $O(n^2)$ space and $O(n^2)$ preprocessing time. Previous Gray code algorithms required space proportional to the number of spanning trees, while the previous constant amortized time algorithms required $O(nm)$ space and did not produce a Gray code listing of the spanning trees.

It is believed, due to empirical evidence, that algorithm **Gen** of Section 3.1 executes in constant amortized time if preprocessing is performed, along with the formation of parallel and series edge records. The analysis of **Gen** requires a tighter lower bound on the number of spanning trees than was developed in the previous chapter. It is believed that a connected simple graph with $\delta(G) \geq 3$ must contain at least $2^{n/2}$ spanning trees. Valdes [25] has determined the minimum number of spanning trees of 2-connected cubic graphs. After simplifying and reducing the bound by Valdes, we have a lower bound of at least $8^{n/4}$ spanning trees for any 2-connected cubic graph.

Two open problems are whether the space requirement can be reduced to $O(n+m)$, and whether a loop-free algorithm can be developed to generate spanning trees in Gray code order.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Don Mills, 1974.

[2] I. Berger. The enumeration of trees without duplication. *IEEE Trans. on Circuit Theory*, CT-14:417–418, 1967.

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

[4] R.L. Brooks, C.A.B. Smith, A.H. Stone, and W.T. Tutte. Determinants and current flows in electric networks. *Discrete Mathematics*, 100:291–301, 1992.

[5] L. Caccetta and K. Vijayan. Applications of graph theory. *Ars Combinatoria*, 23B:21–77, 1987.

[6] A. Cayley. A theorem on trees. *Quart. J. Math.*, 23:376–378, 1889.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* McGraw-Hill, Toronto, 1990.

[8] R.L. Cummins. Hamilton circuits in tree graphs. *IEEE Trans. on Circuit Theory*, CT-13:82–90, 1966.

[9] H.N. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Computing*, 6:139–150, 1977.

[10] H.N. Gabow, Z. Galil, and T.H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of the ACM*, 36:540–572, 1989.

[11] H.N. Gabow and E.W. Myers. Finding all spanning trees of directed and undirected graphs. *SIAM J. Computing*, 7:280–287, 1978.

[12] C. A. Holzmann and F. Harary. On the tree graph of a matroid. *SIAM J. Appl. Math.*, 22:187–193, 1972.

[13] T. Kamae. The existence of a hamilton circuit in a tree graph. *IEEE Trans. on Circuit Theory*, CT-14:279–283, 1967.

[14] S. Kapoor and H. Ramesh. Algorithms for generating all spanning trees of undirected, directed and weighted graphs. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Lecture Notes in Computer Science*, pages 461–472. Springer-Verlag, 1992.

[15] G. Kishi and Y. Kajitani. On hamilton circuits in tree graphs. *IEEE Trans. on Circuit Theory*, CT-15:42–50, 1968.

[16] G. Kishi and Y. Kajitani. On the realization of tree graphs. *IEEE Trans. on Circuit Theory*, CT-15:271–273, 1968.

[17] R.B. Mallion. On the number of spanning trees in a molecular graph. *Chem. Phys. Lett.*, 36:170–174, 1975.

[18] W. Mayeda and S. Seshu. Generation of trees without duplications. *IEEE Trans. on Circuit Theory*, CT-12:181–185, 1965.

[19] G. J. Minty. A simple algorithm for listing all the trees of a graph. *IEEE Trans. on Circuit Theory*, CT-12:120, 1965.

[20] R.C. Read and R.E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5:237–252, 1975.

[21] F. Ruskey. Combinatorial generation. Draft copy of book, 1994.

[22] A. Shioura and A. Tamura. Efficiently scanning all spanning trees of an undirected graph. Technical Report B-270, Department of Information Sciences, Tokyo Institute of Technology, June 1993.

[23] R. Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2:160–161, 1974.

[24] R. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6:306–318, 1985.

[25] L. Valdes. Extremal properties of spanning trees in cubic graphs. *Congressus Numerantium*, 85:143–160, 1991.

[26] R.J. Wilson. *Introduction to Graph Theory*. Wiley, New York, 1985.

VITA

Surname: Smith Given Names: Malcolm James

Place of Birth: Prince Rupert, B.C. Date of Birth: October 23, 1962

Educational Institutions Attended:

University of Victoria	1980 to 1995

Degrees Awarded:

B.Sc.	University of Victoria	1985

Honours and Awards:

Publications:

PARTIAL COPYRIGHT LICENSE

Title of Thesis:

Generating Spanning Trees

Author: ______________________________
Malcolm James Smith

____
Date