Univ. u Novom Sadu
Zb. Rad. Prirod.–Mat. Fak.
Ser. Mat. 21, 1 (1991), 39-56

Review of Research
Faculty of Science
Mathematics Series

A SPACE OPTIMAL ALGORITHM FOR ENUMERATING SPANNING TREES OF A CONNECTED GRAPH

Ladislav Novak, Žarko Karadžić
Faculty of Technical Sciences, University of Novi Sad
Trg Dositeja Obradovića 6, 21000 Novi Sad, Yugoslavia

and

Dragan M. Acketa
Institute of Mathematics, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia

Abstract

A simple space optimal algorithm is introduced for generating all the spanning trees of a connected graph G. The algorithm is based on a backtracking on the family F, which contains all the stars of G, with one single exception, sorted by non-decreasing cardinality. The efficiency of the algorithm is due to a local test, which gives the answer to the question: "Will the addition of a new edge x to a cycle-free subset S of edges of G make the set $S \cup x$ contain a cycle of G?"

AMS Mathematics Subject Classification (1980): Primary: 05C05 Secondary: 05C30, 68E10
Key words and phrases: Spanning trees, enumeration, backtracking

1. Introduction

We assume familiarity with some very basic notions of graph theory.

Let there be given a connected unoriented graph G by its standard representation, that is, by a collection of (un)ordered pairs of its vertices. We shall give an algorithm for an efficient enumeration of all the spanning subtrees of G.

Let the vertices of G be denoted by $1, 2, ..., n$. A *star* associated to a vertex i (denotation: $Star[i]$) is a set of edges of G, which are incident to i. The vertex-set of G and the edge-set of G will be denoted by $V(G)$ and $E(G)$ respectively.

A *spanning tree* T of G is a cycle-free subgraph T of G (= a tree), which is incident with all the vertices of G. Thus, if $|V(G)| = n$, then $|E(T)| = n - 1$.

This paper introduces a simple space optimal algorithm for generating all the spanning trees of a connected graph G. The algorithm is illustrated by an example, which includes the trace. Some experimental results concerning the speed of the algorithm on some classes of graphs, as well as the influence of sorting to this speed, are also discussed.

2. Algorithm

2.1 Structure of the algorithm

We are going to primarily identify the stars associated to the vertices of the input graph G. The family of $V(G) - 1$ stars of G may be implemented as an array of linked lists, the elements of which are associated to edges.

It turns out (this will be explained later) that the sorting of the identified stars w.r.t. their non-decreasing cardinalities is a useful operation in the stage of preprocessing.

The algorithm uses a standard backtracking on the stars, by keeping a current candidate for tree (a cycle-free set of edges), which is expanded whenever the addition of a new edge to the candidate would not produce a cycle within the enlarged candidate and reduced whenever there are no new possibilities for expansion:

The main shell BACKTRACK of the backtracking process seems as follows:

```
PROCEDURE BACKTRACK;
    BEGIN
        (* Let C denote a (current) tree candidate
            (a subgraph of G which has no cycles), and
            let x denote a current edge of G,
            which we will try to add to C. *)

        C := the empty set ; (* Initialization *)
        x := the first edge from the first star ;

        REPEAT
            IF the set C ∪ x does not contain a cycle of G THEN
                IF | C ∪ x| = |V(G)| - 1 THEN BEGIN
                    output ( C ∪ x ); (* new tree is generated *)
                    BACKWARDS
                END
                ELSE
                    FORWARDS
            ELSE
                BACKWARDS
        UNTIL stop
        (* the TRUE value of the global boolean variable "stop"
            should be set during the last call of BACKWARDS *)
    END; (* BACKTRACK *)
```

Thus the backtracking is directed backwards in the cases when the addition of new edge x closes a cycle, as well as in the cases of a final success, that is, in the cases when a new spanning tree is produced by addition of x. In both cases the edge x is deleted from the current candidate and a new attempt should be made; that is, a new edge x should be found.

```
PROCEDURE BACKWARDS;
    BEGIN
        Give up from the addition of the new edge and choose the following
        edge x, for which a new addition atempt will be made
    END;

PROCEDURE FORWARDS;
    BEGIN
```

```
        Add the new edge to the current candidate
    END;

                                    (* Main program *)
BEGIN
    Identify the stars of the input graph G;
    (* Generate their associated linked lists *)

    Sort the identified stars by their non-decreasing cardinalities
    and delete one star with the maximal cardinality;

    BACKTRACK
END.
```

2.2 Local test and conrol connectivity structure

The key idea of the algorithm is to introduce a local test, which decides whether the addition of a new edge to a cycle-free set of edges will close a cycle or not (answer to the question posed in the first IF-statement within the main loop of the backtracking).

For this purpose, we would by no means construct the whole family of cycles of G. That family would be an unbearably large database with large memory requirements, which would be time-consuming even in the construction stage, but much more time-consuming in the stage of its use at each step of the backtracking.

Instead, we shall use a control structure, which would give exact answers to the very question that we ask, but which would not recognize the cycles of G themselves (such a recognition would be beyond our requirements). This structure will be adjusted at each step of the backtracking and will consist of a vector indexed by vertices of G, the components of which are the labels of connected components, which correspond to the current candidate.

The crucial idea of the local test is the following:

An edge $x = \{u, v\}$ is allowed to be added to a candidate C (and a step forwards should be made) if and only if the vertices u and v belong to different connected components of the graph $G(C)$, which is defined in

the following way:

$$V(G(C)) = V(G) \quad \text{and} \quad E(G(C)) = V(C) .$$

Namely, if the vertices u and v belong to the same connected component of $G(C)$, then there already exists a path P connecting them. The addition of the edge $\{u,v\}$ would close the cycle $P + \{u,v\}$, which is a forbidden situation.

On the other hand, if the vertices u and v belong to different connected components , then the addition of $\{u,v\}$ would join two connected components (trees) into one (larger tree); $n-1$ consecutive joins of this type are sufficient to produce a spanning tree.

Initially, each vertex i ($1 \leq i \leq |V(G)|$) is a separate connected component labelled with the integer i. Connected components can be implemented as linked lists, the elements of which are associated to the vertices of G. Each one of these lists can be reached by two pointers, which are attached to the head and to the tail of the list, respectively.

It is essential to note that a new control vector is produced whenever the procedure FORWARDS is called and that the control vectors associated to lower levels of backtracking are kept in memory, in order to make possible their restoration in cases of back flow during the backtracking process. We shall put these comments more precisely:

We need to have exactly $n-2$ control vectors. Each control vector $v[i]$ is associated to Star[i] ($1 \leq i \leq n-2$), the star ordering after sorting being assumed.

Suppose that the candidate C has j edges ($1 \leq i \leq n-2$) and that the addition of a new edge $\{u,v\}$ is considered, where the vertices u and v belong to different connected components $Comp_1$ and $Comp_2$ of the graph $G(C)$. Let

- $head_1$ and $tail_1$ be the pointers showing to $Comp_1$;
- $head_2$ and $tail_2$ be the pointers showing to $Comp_2$.

The control vector $v[j+1]$ is generated (this is an additional task performed by the procedure FORWARDS) from the vector $v[j]$ by application of the following two commands:

(1) $v[j+1] := v[j]$; (* copy the previous control vector *)

(2) Replace in $v[j+1]$ the labels (associated to $Comp_2$) of all the vertices from $Comp_2$ by the label associated to $Comp_1$. (* This last label will become the common label of the joined tree *).

Remark. Note that the labels themselves are generated at random; in fact, they are highly dependent on the order of input data. However, their role will be successfully performed regardless of their values; all that the algorithm requires to know at some moment is whether the labels of the corresponding two vertices are equal or not.

Step 2) (the label replacement) is performed during the passage through the linked list associated to $Comp_2$. The new linked list, which corresponds to the joined tree, is generated by the following three operations:

- $newhead := head_1$;
- $newtail := tail_2$;
- $tail_1\hat{}.next := head_2$.

The procedure BACKWARDS requires restoration of the previous control vector solely in cases when the current star is exhausted. Then the last control vector, say $v[j+1]$, is replaced by the already existing control vector $v[j]$. Such a replacement will be impossible solely in the case when $j = 0$. However, the last condition denotes that the whole backtracking is completed (the value of the boolean variable stop in the above shell should be set to TRUE at that moment).

Remark. Note that the control vector $v[n-1]$ is not necessary. Those calls of the procedure FORWARDS, which complete a spanning tree, are necessarily followed by the deletion of the last added edge, and by coming back to the previous state, which is controlled by the vector $v[n-2]$.

2.3 Review of the algorithm

We shall proceed with a complete and detailed review of the above described algorithm:

ALGORITHM
for enumeration of all the spanning trees of a connected graph

```
ALGORITHM List-all-spanning-trees;

Input: Connected unoriented graph G represented by its edges
Output: All the spanning trees of G, without repetitions

PROCEDURE Choice-of-the-following edge;
    BEGIN (* Choice-of-the-following edge *)

        IF there exists the next edge in the j-th list THEN choose it
        ELSE WHILE (end of the j−th list) AND (j > 0) DO BEGIN

            Make the current edge of the j−th list be equal
            to the first edge of that list;
            (* this is in accordance with the lexicographic principle;
                later visits to the j−th list should start from the
                beginning, because some move to the right
                has already been made on a lower level *)
            j := j − 1;

            Restore the previous state of the control structure
            (* this state has been kept since the previous visit *)
        END
    END; (* Choice-of-the-following edge *)
```

Remark. Global variable j is used within the main loop by means of a "side effect" (the same holds for the most of the variables used).

```
BEGIN (* List-all-spanning-trees *)

    Identify each one of the n stars of G;
    Determine the degrees of these stars;
    Sort the degrees into a non-decreasing sequence;
    Delete the star with a maximal degree;
    j := 1; (* the serial number of the current star *)
```

```
    REPEAT (* main loop of the backtracking *)

        IF the vertices of the current edge of the j-th
            list belong to two different connected components THEN

            IF  j = |V(G)| - 1  THEN BEGIN
                (* the last edge ; there is no need to
                adjust the control structure *)
            Output tree;
            Choice-of-the-following edge
            END
            ELSE BEGIN
                Adjust the control structure;
                j := j + 1
            END

        ELSE
            Choice-of-the-following edge

    UNTIL  j = 0

END ; (* List-all-spanning-trees *)
```

3. Proof of validity of the algorithm

In this section we shall prove that the algorithm given in Section 2. is correct. In addition, we shall show that the algorithm is space optimal.

We are primarily going to prove the following lemma:

Lemma 1. *Let G be a connected graph on n vertices, let S be a subset of $V(G)$ consisting of $n-1$ vertices, and let T be a spanning tree of G. Then there exists a bijection $f : E(T) \to S$, such that each edge x of T belongs to the star determined by the vertex $f(x)$.*

Proof. We label the vertices of T in the following way: the unique vertex z from $V(G) - S$ is labelled by 0, while the other vertices are labelled by the distance (within T) from z. Each edge x of T connects two vertices

with different labels $k-1$ and k and the required bijection f can be defined with $f(x)=k$. □

Now we are able to prove the validity of the algorithm :

Theorem 1. *The algorithm given in Section 2. is correct, that is, it produces exactly all the spanning trees of the input connected graph G without duplications.*

Proof. We shall separately prove three facts concerning the presented algorithm. Two independent proofs will be given concerning the third of them:

(1) All the spanning trees of G are generated by our algorithm; that is, no spanning tree of G can be missed.

(2) The only sets which are output by our algorithm are the edge-sets of spanning trees of G .

(3) Each spanning tree of G is generated only once by our algorithm; that is, the algorithm does not produce duplicates.

Proof of (1): The above proved lemma implies that the set of edges of each spanning tree T is a transversal (= a system of distinct representatives) of the family F of stars indexed by vertices of S. More precisely, each edge x of T should be represented as a member of $Star(f(x))$. Such a representation guarantees that an algorithm which passes through all the transversals of the family F will not miss any spanning tree of T.

Proof of (2): Each output set has cardinality $n-1$ and has no cycles by construction, that is, it IS a spanning tree of G.

The first proof of (3): Suppose that there exists a spanning tree T, which has two distinct associated transversals $a_1,...,a_{n-1}$ and $b_1,...,b_{n-1}$. Thus both sets $\{a_1,...,a_{n-1}\}$ and $\{b_1,...,b_{n-1}\}$ are equal to the set $E(G)$ and there exists an index i, such that $a_i \neq b_i$ for some i from the set $\{1,2,...,n-1\}$.

It is easy to conclude that there exists a sequence $i=i_1,...,i_k$ of indices such that

$$a_{i1}=b_{i2},\ a_{i2}=b_{i3}\ ,\ \ldots\ ,\ a_{ik}=b_{i1}\ .$$

Note that, e.g., $a_{i1} = b_{i2}$ implies that $Star(i1)$ and $Star(i2)$ have a common edge of G. It follows that the edges $\{i1, i2\}, \{i2, i3\}, ..., \{ik, i1\}$ of T constitute a cycle, which contradicts the assumption that T is a tree.

The second proof of (3): It suffices to show that the bijection f introduced in the above lemma is unique. In other words, we should show that the transversal representation of a tree T w.r.t. the stars associated to the vertices of S is unique.

Let z denote the only vertex in the difference of the sets $V(G)$ and S. Any edge of T, which is of the form $\{z, y\}$, must be considered as the member of $Star(y)$, because $Star(z)$ is not included in our representation of spanning trees. In a similar way, by expanding the process started in the "root" z, we can uniquely determine the star to which each of the remaining edges of T should be attached. □

There remains to show that our algorithm is space optimal :

Theorem 2. *The given algorithm is space optimal; more precisely, its worst-case space complexity is equal to the size of the input; that is to* $O(n^2)$, *where* n *is the number of vertices of* G.

Proof. The structures that consume most of space in our algorithm are arrays of linked lists. Their particular lists are associated to vertices of G; that is, to their stars. However, the space necessary to store them is obviously of an $O(n^2)$ size. □

4. An illustrative example

We are going to give the trace of the above described algorithm , which is applied to a small-size example :

Let the connected graph G be given on 4 vertices (numerated 1 thru 4) and 6 edges (denoted $e1, ..., e6$) as follows:
$e1 = (1,2)$; $e2 = (2,3)$; $e3 = (2,3)$;
$e4 = (3,4)$; $e5 = (4,1)$; $e6 = (4,2)$;

The linked lists which corresponding to particular connected stars seem

after sorting as follows :

$Star[1]: \rightarrow e5 \rightarrow e1$
$Star[3]: \rightarrow e4 \rightarrow e3 \rightarrow e2$
$Star[4]: \rightarrow e6 \rightarrow e5 \rightarrow e4$
$Star[2]: \rightarrow e6 \rightarrow e3 \rightarrow e2 \rightarrow e1$

We shall proceed with a trace of the backtracking in this example:

Applications of the procedure FORWARDS are denoted by "ADD", while the applications of the procedure BACKWARDS are denoted by "TRY" or "DELETE". "TRY" denotes an unsuccessful attempt of attaching new edge on the same level , while "DELETE" reduces the candidate after all the possibilities for augmentation on the previous level are exhausted. The vector of incidencies of the vertices 1,2,3,4,5,6 (in order) to the labelled connected components. The labels of which are the elements of the vector are provided at each step of the algorithm.

Note that the current candidate determines uniquely the corresponding incidence vector. However, both of them are not sufficient to determine uniquely the position within the backtracking . Therefore, an additional piece of information is also added (in brackets) with each position: the next edge from the considered star, the addition of which should be attempted at the next step; if such an edge does not exist, then the algorithm deletes the last edge and comes back to the previous star in search of a new edge to be added. Such a situation is denoted by "—-" in the column (next).

TRANSFORMATION	CURRENT CANDIDATE			(next)	INCIDENCE VECTOR
			– – – – –	(*e*5)	1 2 3 4
ADD	*e*5		*e*5	(*e*4)	4 2 3 4
ADD	*e*4		*e*5 *e*4	(*e*6)	3 2 3 3
ADD	*e*6	*TREE*	*e*5 *e*4 *e*6	– – – – –	– – – – –
DELETE	*e*6		*e*5 *e*4	(*e*5)	3 2 3 3
TRY	*e*5		*e*5 *e*4	(*e*4)	3 2 3 3
TRY	*e*4		*e*5 *e*4	– – – – –	3 2 3 3
DELETE	*e*4		*e*5	(*e*3)	4 2 3 4
ADD	*e*3		*e*5 *e*3	(*e*6)	4 2 2 4
ADD	*e*6	*TREE*	*e*5 *e*3 *e*6	– – – – –	– – – – –
DELETE	*e*6		*e*5 *e*3	(*e*5)	4 2 2 4
TRY	*e*5		*e*5 *e*3	(*e*4)	4 2 2 4

TRANSFORMATION	CURRENT CANDIDATE		(next)	INCIDENCE VECTOR
ADD	*e*4	*TREE*	*e*5 *e*3 *e*4	– – – –	– – – – – –
DELETE	*e*4		*e*5 *e*3	– – – –	4 2 2 4
DELETE	*e*3		*e*5	(*e*2)	4 2 3 4
ADD	*e*2		*e*5 *e*2	(*e*6)	4 2 2 4
ADD	*e*6	*TREE*	*e*5 *e*2 *e*6	– – – –	– – – – – –
DELETE	*e*6		*e*5 *e*2	(*e*5)	4 2 2 4
TRY	*e*5		*e*5 *e*2	(*e*4)	4 2 2 4
ADD	*e*4	*TREE*	*e*5 *e*2 *e*4	– – – –	– – – – – –
DELETE	*e*4		*e*5 *e*2	– – – –	4 2 2 4
DELETE	*e*2		*e*5	– – – –	4 2 3 4
DELETE	*e*5		– – – – –	(*e*1)	1 2 3 4
ADD	*e*1		*e*1	(*e*4)	1 1 3 4
ADD	*e*4		*e*1 *e*4	(*e*6)	1 1 3 3
ADD	*e*6	*TREE*	*e*1 *e*4 *e*6	– – – –	– – – – – –
DELETE	*e*6		*e*1 *e*4	(*e*5)	1 1 3 3
ADD	*e*5	*TREE*	*e*1 *e*4 *e*5	– – – –	– – – – – –
DELETE	*e*5		*e*1 *e*4	(*e*4)	1 1 3 3
TRY	*e*4		*e*1 *e*4	– – – –	1 1 3 3
DELETE	*e*4		*e*1	(*e*3)	1 1 3 4
ADD	*e*3		*e*1 *e*3	(*e*6)	1 1 1 4
ADD	*e*6	*TREE*	*e*1 *e*3 *e*6	– – – –	– – – – – –
DELETE	*e*6		*e*1 *e*3	(*e*5)	1 1 1 4
ADD	*e*5	*TREE*	*e*1 *e*3 *e*5	– – – –	– – – – –
DELETE	*e*5		*e*1 *e*3	(*e*4)	1 1 1 4
ADD	*e*4	*TREE*	*e*1 *e*3 *e*6	– – – –	– – – – –
DELETE	*e*4		*e*1 *e*3	– – – –	1 1 1 4
DELETE	*e*3		*e*1	(*e*2)	1 1 3 4
ADD	*e*2		*e*1 *e*2	(*e*6)	1 1 1 4
ADD	*e*6	*TREE*	*e*1 *e*2 *e*6	– – – –	– – – – –
DELETE	*e*6		*e*1 *e*2	(*e*5)	1 1 1 4
ADD	*e*5	*TREE*	*e*1 *e*2 *e*5	– – – –	– – – – –
DELETE	*e*5		*e*1 *e*2	(*e*4)	1 1 1 4
ADD	*e*4	*TREE*	*e*1 *e*2 *e*6	– – – –	– – – – –
DELETE	*e*4		*e*1 *e*2	– – – –	1 1 1 4
DELETE	*e*2		*e*1	– – – –	1 1 3 4
DELETE	*e*1		– – – – –	– – – –	

Total number of trees : 13

Remark. The steps backwards in the above trace are always preceded by either producing a tree or by an attempt to add a new edge to a candidate which already contains that edge (the last situation is equal to the occurrence of a cycle of size 2 within the candidate). In the general case the attempts will also be made to close some cycles of a larger size. These attempts will be prevented in principally the same manner by use of the control vector.

5. Experimental results on some graph classes

In this section we are going to give a number of examples, with which the tests on the action of the above described algorithms were performed. The obtained experimental results include the total number of trees with graphs of a specified type, the total time (in seconds), which was used by a certain PC AT 386 (in all cases the same one) to enumerate these trees, as well as the average time (in 10^{-6} seconds) of the calculation per tree.

Example 1. Let K_n denote the complete graph on n vertices. It is well-known that the total number of spanning trees of Kn is n^{n-2}.

Experimental results for the class K_n:

n	5	6	7	8	9	10
num.of trees	125	1296	16807	262144	4782969	100000000
time(sec)	0.06	0.11	1.10	16.15	281.27	5664.85
10^{-6} s p.t.	480	84.9	65.4	61.6	58.8	56.6

Conclusion: It seems that the average speed per tree does note an increase with n. In fact, it slowly decreases (perhaps towards a limit ?), with the exception of the first few values, where there is a great decrease. Such a behaviour might be explained by some constant time (probably spent with the input-output business), which does not depend on the size of the problem.

This example suggests that the complexity of the algorithm is equal to the size of the output (that is, that a constant time is needed for each particular generated tree).

Example 2. Let A_n denote the graph on $2 \cdot n$ vertices, numerated by

$1,2,...,2\cdot n$, and the following $3\cdot n-2$ edges:

$$[i,i+1] \quad \text{for } i=1,2,...,2\cdot n-1$$
$$[i,2\cdot n+1-i] \quad \text{for } i=1,2,...,n-1$$

Experimental results for the class A_n:

n	1	2	3	4	5	6	7	8	9
num. of trees	1	4	15	56	209	780	2911	10864	40545
time(sec)	–	–	–	–	0.11	0.50	3.57	14.94	45.37
10^{-6} s p.t	–	–	–	–	526.3	641.0	1226.4	1375.2	1119.0

n	10	11	12	13
num. of trees	151316	564719	2107560	7865521
time(sec)	322.02	1248.34	3528.20	25076.45
10^{-6}s p/t	2128.1	2210.6	1674.1	3188.1

Example 3. Let B_n denote the graph on $4\cdot n$ vertices, numerated by $1,2,...,4\cdot n$, and the following $8\cdot n-2$ edges:

$$[i,i+1] \quad \text{for } i=1,2,...,4\cdot n-1$$
$$[i,4\cdot n+1-i] \quad \text{for } i=1,2,...,2\cdot n-1$$
$$[2\cdot i-1,2\cdot i] \quad \text{for } i=1,2,...,n$$
$$[2\cdot i,4\cdot n+2-2\cdot i] \quad \text{for } i=1,2,...,n\,.$$

Experimental results for the class B_n:

n	1	2	3	4
num. of trees	13	533	21801	891709
time(sec)	–	0.22	12.79	904.57
10^{-6} s p/t	–	412.8	586.7	279.5

Remark. Note that the graphs B_n are obtained by inserting some edges in the corresponding graphs A_n. In such a case the average time per tree decreases. It seems as if the graphs with a "higher edge density" have smaller average time per tree and conversely.

Example 4. Let P_n denote the path consisting of n edges. The time (in hds/s = sec/100) necessary to output the only spanning subtree of P_n by our algorithm is given below for sime values of n:

n	40	50	60	70	80	90	100	110	120	130	140	150
	5	11	16	22	27	39	50	55	66	77	88	104

n	160	170	180	190	200	210	220	230	238
	115	132	148	165	187	208	225	247	263

This time has an almost linear increase, which is quite an expectable behaviour, since a linear growth of the structure is also present.

6. On the influence of star sorting

We have made an extensive study of the dependence between the ordering of stars (by their cardinalities) and the speed of our algorithm for generating all the spanning trees in the following example:

Let the connected graph G be given on 7 vertices (numerated 1 thru 7) and 10 edges (denoted e1,...,e10) as follows:

$$e1 = (1,2); \quad e2 = (1,3); \quad e3 = (2,3); \quad e4 = (2,5);$$
$$e5 = (3,4); \quad e6 = (3,5); \quad e7 = (3,6); \quad e8 = (4,5);$$
$$e9 = (5,6); \quad e10 = (1,7);$$

We observe that the degrees of the vertices 1,2,3,4,5,6,7 are 3,3,5,2,4,2,1 respectively. Of course, the same holds for the cardinalities of the corresponding stars. On the other hand, 7 is the only vertex of degree 1, 4 and 6 are the vertices of degree 2, 5 is the only vertex of degree 4 and 3 is the only vertex of degree 5.

We have compared 480 runnings of our algorithm on the graph G (which has 52 spanning trees), which differed from each other solely in the ordering of the stars (equivalently, the corresponding vertices). These orderings were chosen as follows:

For each one of the 120 permutations of the degree set 1,2,3,4,5, we have introduced four corresponding orderings. We have adjoined to the degrees (in order) either the unique corresponding vertex or (in the case of degrees 2 and 3) two corresponding vertices (there are four possible arrangements in this last case).

For example, given the permutation 21435, we can primarily adjoin the auxiliary sequence 2214335 to it, which further leads to the following four orderings:
4,6,7,5,1,2,3; 4,6,7,5,2,1,3; 6,4,7,5,1,2,3; 6,4,7,5,2,1,3.

With each one of the 480 orderings, we have counted the number of steps down (= the number of steps up) in the backtracking across the list of stars.

Observations:

1) There is a considerable difference between the number of counted steps, as well as with the runtime, with distinct orderings, with this particular graph G. It seems that the same is generally valid for the graphs with sufficiently diverse vertex degrees.

2) There is a strong correlation between the number of counted steps, although there is also some alteration w.r.t. the expected strict direct proportion.

   Example. The first and the second ordering associated to the permutation 41235, that is, 5746123 and 5746213 respectively, have the respective numbers of steps 51 and 59, but the respective runtimes are 11 hds/s and 6 hds/s.

   Example. As two antipodes, we might consider the ordering 7461253, obtained lexicographically with the first permutation 12345, which has 35 steps and 5 hds/s, and the ordering 3512647, obtained lexicographically with the last permutation 54321, which has 286 steps and 44 hds/s. All the four figures (35, 5, 286 and 44) are the absolute extremes within this test.

3) Given a quadruple of orderings, note that one ordering can be transformed to another one according to the following scheme:

   first → third , by transposition (46)
   second → fourth , by transposition (46)
   first → second , by transposition (12)
   third → fourth , by transposition (12) .

   With all the 120 quadruples of orderings, the number of steps with the first and the third (respectively, with the second and the fourth) ordering have the same numbers of steps (although in many cases all the four numbers are equal). This seemingly interesting feature is valid regardless of whether 2 is in front of 3, or conversely, in the initial permutation.

   The question is: Why does the permutation (12) , of vertices of degree 3, have stronger consequences (on the number of steps) than the permutation (46), of vertices of degree 2 ?

4) In many cases we have coinciding runtimes with the first and third, respectively with the second and the fourth ordering associated to a permutation. However, in many other cases, the coincidence relates to the first and the fourth, respectively the second and the third ordering. At last, there are many cases with which there is one outstanding (mostly greater) runtime with respect to the other three. Adding the cases with four coincidences, these are the four by far most common behaviours of the runtimes.

   Remark. Observe that some other elementary steps should be counted as well. Besides up and down, it would be interesting to count steps to the right, within the particular stars.

5) It would be of extreme interest to study the relationships between the initial permutation and the (average) number of steps and runtime associated to the corresponding orderings. It seems that the last values are proportional to a specific function defined on permutations p_1, p_2, p_3, p_4, p_5:

$$\sum_{i=1}^{5} |i - p_i| \ ,$$

   although some refinement is also necessary. (Look to the permutations with the same value of this function!)

6) It seems reasonable to study the effect of sorting with different types of examples. Thus after the sorting is completed, it is important which of the following monotonously increasing behaviours is present:
   a) concave **b)** linear **c)** convex **d)** constant

Paper [1] contains another algorithm for the enumeration of all the spanning trees of a connected graph. The worst-case time complexity of the algorithm is $O(n + m + n \cdot t)$, where n and m denote respectively the number of vertices and the number of edges, while t denotes the number of spanning trees. This paper also contains a detailed comparative study of the existing algorithms for spanning tree enumeration, as well as an extensive list of references on the subject.

References

[1] Winter, P.: An algorithm for the enumeration of spanning trees, BIT 26(1986), 44-62.

REZIME

JEDAN MEMORIJSKI OPTIMALAN ALGORITAM ZA NABRAJANJE PREPOKRIVAJUĆIH STABALA POVEZANOG GRAFA

Opisan je jedan jednostavan, i u pogledu memorijskih zahteva optimalan, algoritam za generisanje svih prepokrivajućih stabala datog povezanog grafa G. Izbor grana se vrši bektrekingom po familiji koja sadrži sve zvezde grafa, sa izuzetkom jedne (proizvoljne). Efikasnost algoritma se zasniva na *lokalnosti* testa, kojim se ispituje da li će dodavanje jedne nove grane x skupu grana S, koji ne sadrži konturu grafa G, – dovesti do toga da $S \cup x$ sadrži takvu konturu.

Received by the editors March 12, 1990.