A NEW SPANNING TREE ALGORITHM

BY

W. Dana Wasson
Robert McIssaac

TR74-002, October 1974

A NEW SPANNING TREE ALGORITHM

BY

W. DANA WASSON
Professor of Computer Science
and Director
School of Computer Science
University of New Brunswick
Fredericton, N.B.

ROBERT McISSAAC
IBM Canada Ltd.
Saint John, N.B.

TR74-002, October 1974

A NEW SPANNING TREE ALGORITHM

W. Dana Wasson
School of Computer Science
University of New Brunswick
Fredericton, N.B.

Robert McIssaac
IBM Canada Ltd.
Saint John, N.B.

Summary

The algorithm outlined in this paper finds all the spanning trees of a non-directed graph G. The graph is represented by a binary incidence matrix using compact bit storage. Each row of the matrix corresponds to a branch of the graph and is represented by one or more computer words. Each column of the matrix corresponds to a graph node and is represented by a single bit within each computer word or multiple word. It is shown, in addition to saving computer storage, that this representation greatly speeds up the tree generation process by allowing many bit-parallel operations to occur during the execution of a single computer instruction cycle.

Introduction

A spanning tree of a n-node, connected, non-directed, graph G has the following properties;

(P1) It is connected
(P2) It contains n-nodes (vertices)
(P3) It contains no loops (circuits)
(P4) It contains n-1 branches (edges)

It is easily shown that any three of the above properties of a spanning tree implies the fourth.

Any combination of b-branches taken n-1 at a time is a possible tree candidate. There are bCn-1 candidates and since each tree candidate adheres to property P4, only two more properties need be checked to determine if the candidate is a valid tree. The algorithms presented in this paper chooses properties P1 and P3 for this purpose (i.e. are the n-1 branches connected and loopless). These properties are easily checked if the graph G is represented by the binary incidence matrix as defined in the next section. It can be shown that Minty's algorithm [1] for listing all the trees of a graph is in effect checking each tree candidate for properties P1, P2 and P4 rather than P1, P3 and P4. A later section of this paper compares Minty's and the new algorithm in this regard and also experimentally.

A recent report [2] (October, 1970) gives an "Analysis of Algorithms for Finding All Spanning Trees of a Graph" that attempts to be machine independent in the comparison of the algorithms. Both empirical and analytical methods were used. The empirical methods used a Graph Algorithm Software Package GASP which counts the number of GASP operations in an attempt to obtain machine independence. GASP is an extension of PL/1. This extension is obtained by the use of a PL/1 Preprocessor. GASP statements consist of PL/1 statements, GASP Procedure calls and Type-functions. In addition GASP has sets and graphs as additional data types.

The counting of GASP operations only give limited machine independence to the comparison of algorithms operating on different models of a computer series with the same architecture (eg. the IBM System 360/370 series) since the ratio of instruction speeds among the various models is not constant. The implementation of algorithems on the same machine using different versions of the same programming language can yield different conclusions as to what is the "best" algorithm. For example, a recent International Mathematical and Statical Libraries (IMSL) newsletter [3] states that

> "Parlett and Wang [4] indicate how the desire for optimum execution time efficiency can conflict with portable program development, even between compilers on a single computer type. Their points are developed through comparing the cost of triangular factorization for several computer-compiler combinations and several "reasonable" implementations of different factorization algorithms.
>
> ..... For example, with regard to the question on explicit versus implicit pivoting, they state:
>
> > "On the IBM 360/50, the 13% advantage of the implicit technique on the G compiler turns into a 40% handicap on the H compiler (for 50 x 50 matrices).....
> >
> > There is a similar phenomenon in comparing the CDC RUN compilers with the FTN type compilers. Here a 19% advantage becomes a 35% to 40% disadvantage for the implicit technique." "

This sort of result is likely to occur in situations where the number of hidden bookkeeping operations approaches the number of required operations placed in evidence from a mathematical statement of that algorithm.

The algorithm presented in this paper is new in its representation and implementation which gives close attention to the computer's architecture. It attempts to make maximum use of bit-parallel operations and at the same time to minimize the total number of machine instructions required.

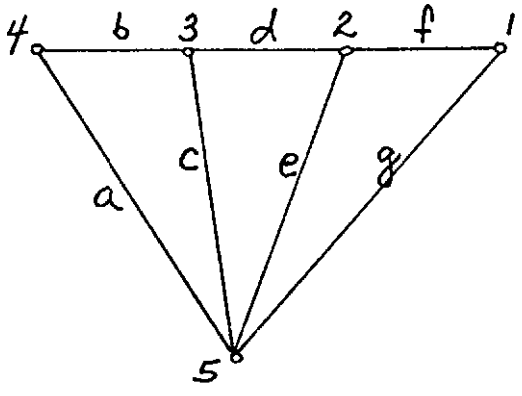
Graph Representation

A binary incidence matrix was used to represent the graph since this greatly facilitates the checking of properties P1 and P3 for each tree candidate.

The elements of the incidence matrix A are defined to be;

$$a_{p,q} = \begin{cases} 1 & \text{if branch p is incident of node q} \\ 0 & \text{otherwise} \end{cases}$$

where $a_{p,q}$ is the (p,q) entry of matrix A. An example of a graph and its incidence matrix is given in Figure 1.



A =

	node q 1	2	3	4	5
a	0	0	0	1	1
b	0	0	1	1	0
c	0	0	1	0	1
d	0	1	1	0	0
e	0	1	0	0	1
f	1	1	0	0	0
g	1	0	0	0	1

Branch p

Figure 1: A Graph and Its Incidence Matrix

The matrix A can be stored very compactly if each bit of a binary computer word is associated with an A entry. In this way a whole row of A, corresponding to a branch, can be represented with one or more computer words. Each column of A corresponds to a node of the graph.

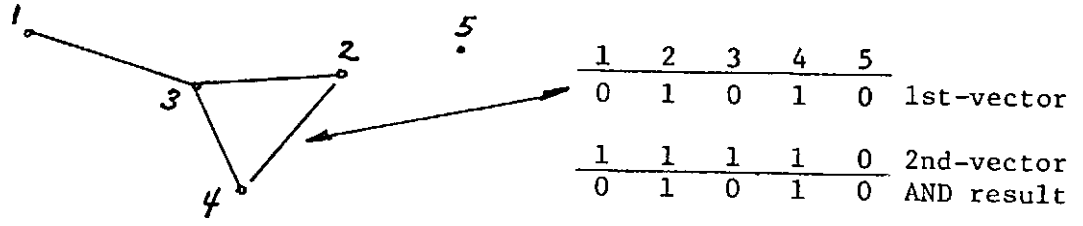
Checking of Properties

Each tree candidate of b branches is checked for properties P1 and P3 to determine if it is connected and loopless. The candidate is a valid tree if it passes these tests.

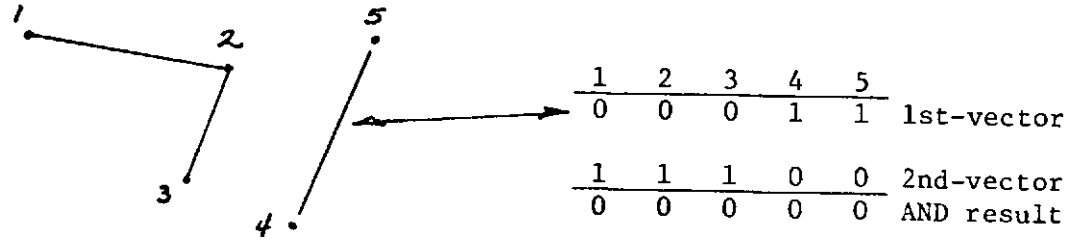
A loop is detected by performing a Boolean AND operation on the row vector of matrix A, representing one of the b branches of the tree candidate, and a second vector representing one or more of the remaining b branches which are connected. The second vector represents a partial or incomplete tree. If the result of the AND operation is equal to the first vector this implies that the nodes to

which the new branch is connected are already connected (contained) in the partial tree. The inclusion of the new branch with the partial tree would result in a loop. However, if the result of the AND operation is a vector of zeros this implies that the branch represented by the first vector is not connected to the set of branches represented by the second vector. In this case the AND operation is repeated with the remaining second vectors. If all results are zero the new branch becomes a new member of k, (k=1,2,...,≤n/2) second vectors.

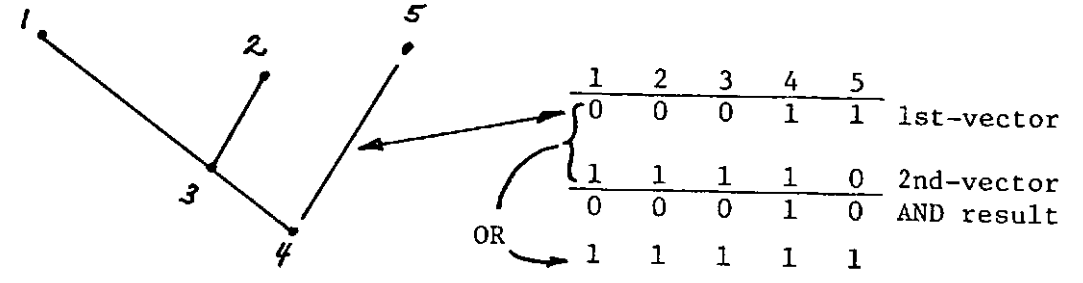
If the result of the AND operation is all zeros except for a single bit this implies that the new branch is connected to the partial tree at a single node. In this case, the second vector is replaced by the Boolean OR of the two vectors. Figure 2 represents the three possible outcomes of the AND operation.



(a) New Branch Forms Loop



(b) New Branch Not Connected To Partial Tree



(c) New Branch Connected to Partial Tree at Single Node

Figure 2: The Addition of a New Branch to a Partial Tree

Figure 2(a) shows a new branch connected between nodes 2 and 4 and is represented by the 1st-vector (a row of the A-matrix). The 2nd-vector represents the partial tree with branches connecting node 3 to nodes 1, 2 and 4. The 1's in columns 1, 2, 3 and 4 of the 2nd-vector indicate that the partial tree connects all these nodes.

The new branch added to the graph of Figure 2(b) is not connected to the partial tree. Thus, the ANDing of the 1st and 2nd-vectors yields a vector of zeros. The new branch added to the graph of Figure 2(c) is connected to the partial tree at a single node. This results in the AND operation yielding a single 1-bit in column 4, the node connecting the new branch to the partial tree. In this third case, the partial tree is updated with the new branch by ORing it with the partial tree. In this case the partial tree is a complete tree.

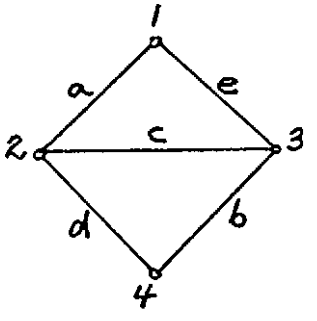
Note that it is possible to resolve these three cases by two tests on the result of the AND operation;

(i) Is the result zero? If yes, case (b) is detected. If no, proceed with test (ii).

(ii) Is the result equal to the 1st-vector? If yes, case (a) is detected. If no, case (c) is detected.

The Complete Algorithm

The complete algorithm will now be presented by considering a graph of four nodes and five branches as shown in Figure 3.



A =

	1	2	3	4
a	1	1	0	0
b	0	0	1	1
c	0	1	1	0
d	0	1	0	1
e	1	0	1	0

Figure 3: Graph and Its Incidence Matrix A.

The number of tree candidates is given by ${}_5C_3=10$. They are;
abc, abd, abe, acd, ace, ade, bcd, bce, bde and cde.
Figure 4(a) shows all the steps involved for testing of a valid tree candidate abc, while Figure 4(b) shows the testing of an invalid tree candidate ace.

Figure 4(a) shows the steps involved in testing tree candidate abc. The first branch a vector is stored in B_1. Then the second branch b is choosen as the new branch and ANDed with B_1 in step (3). Since the result is zero b is stored in B_2 and c is choosen as the new branch. Since B_1 AND c results in a single '1' bit this indicates that a and c are connected. Step (7) updates the partial tree B_1 to include both branches a and c. Since there are no branches left in the tree candidate to be processed it is necessary to connect the partial trees stored in the B_i's by ORing them together after testing that they only touch at a single node.

If they touch at more than one node then a loop is formed. The number of nodes at which the partial trees touch is equal to the number of '1' bits in the ANDing of thepartial tree vectors. Step (8) shows that the partial trees represented by B_1 and B_2 touch at a single node. Therefore step (9) B_1 OR B_2 gives a vector of ones which indicates that the combining of the two partial trees touches all nodes is a valid tree. If there are more than two B_i's then all possible combinations are combined as above until a loop is detected or a valid tree results. Step (6) of Figure 4(b) shows the detection of a loop in tree candidate ace.

1	2	3	4		Comments
1	1	0	0	(1)	a → B, (B_i's hold 2nd-vectors)
0	0	1	1	(2)	Choose new branch b
0	0	0	0	(3)	B_1 AND b = 0; (4) b → B_2
0	1	1	0	(5)	Choose new branch C
0	1	0	0	(6)	B_1 AND c ≠ 0, ≠ c
1	1	1	0	(7)	B_1 OR c → B_1 (new B_1)
0	0	1	0	(8)	B_1 AND B_2 one '1' implies step (9)
1	1	1	1	(9)	B_1 OR B_2 all 1's implies complete tree

Figure 4(a): Testing of Tree Candidate abc.

1	2	3	4		
1	1	0	0	(1)	a → B_1 (B_i's hold 2nd-vectors)
0	1	1	0	(2)	Choose new branch c
0	1	0	0	(3)	B_1 AND c ≠ 0, ≠ c
1	1	1	0	(4)	B_1 OR c → B_1 (new B_1)
1	0	1	0	(5)	Choose new branch e
1	0	1	0	(6)	B_1 AND e=e; loop detected (invalid tree)

Figure 4(b): Testing of Tree Canadidate ace.

The testing for the presence of a single bit in the result of the ANDing of the partial tree vectors is carried out as follows;

Case (1) Partial trees don't touch.

0	0	0	0	0	0	Result of AND operation
					-1	Subtract -1 (2's-complement)
1	1	1	1	1	1	Negative result, therefore partial trees don't touch

Case (2) Partial trees touch at single node.

	0	0	1	0	0	0	Result of AND operation
AND						-1	Subtract -1
	0	0	0	1	1	1	Result not negative
=	0	0	0	0	0	0	Zero result indicates partial trees touch at single node

Case (3) Partial trees touch at two or more nodes.

	0	0	1	0	1	0	Result of AND operation
AND						-1	Subtract -1
	0	0	1	0	0	1	Result not negative
=	0	0	1	0	0	0	Result not zero, therefore a loop is detected

Note that the -1, AND, sequence removes a "1" bit with each application.

Minty's Tree Generation Algorithm

This section introduces a tree generation algorithm, due to Minty [1], for comparison purposes.

Suppose b_1 is a branch of a network graph g. Minty has shown that the trees of g can be classified into those which contain the branch b_1 and those which do not. Let g_1 and g_2 be graphs that are obtained from g by first shorting and then deleting b_1. Then every tree of category 1 consists of trees of g_1 plus b_1, while every tree of category 2 consists of a tree of g_2. The network determinant Δg of graph g is then given by

$$\Delta g = b_1 \Delta g_1 + \Delta g_2 \tag{1}$$

It may be seen that graph g has been converted into two new graphs g_1 having one less branch and node, and g_2 having one less branch. This process is referred to as graph reduction and can be carried on to reduce graph g_1 and g_2. Suppose branch b_2 is selected then graphs g_3 and g_4 are formed from g_1, and graphs g_5 and g_6 are formed from g_2. Branch b_2 is included as a branch of the trees associated with the shorted graphs g_3 and g_5. Using equation (1) it is possible to express Δg as

$$\begin{aligned} \Delta g &= b_1(b_2 \Delta g_3 + \Delta g_4) + b_2 \Delta g_5 + \Delta g_6 \\ \Delta g &= b_1 b_2 \Delta g_3 + b_1 \Delta g_4 + b_2 \Delta g_5 + \Delta g_6 \end{aligned} \tag{2}$$

This reduction process can be continued until the Δg_i's are 1 or 0 (ie. until g_i is a one node graph, or has isolated nodes).

A tree generation algorithm based upon equation (2), and with the aid of Figure 5, is described by the following steps:

1. Draw a graph corresponding to the original network. Number each node and letter each branch. Number the graph 1.

2. Select the lowest lettered branch b_i of the graph with the highest assigned number (initially graph 1).

3. Form a "deleted" graph by removing branch b_i. Draw the deleted graph and assign it a number one higher than the last graph unless one or more nodes are isolated. In this latter case cancel the graph.

4. Form a "shorted" graph by coalescing (shorting) the two nodes connected by branch b_i. Draw the shorted graph with all self-loops removed and associate the name of the branch b_i with this graph. If only one node remains, the branch names associated with this graph now form a tree. Record this tree in a tree list. Number the graph with the next highest number only if more than one node remains. Cancel out the selected graph used to form the deleted and shorted graph.

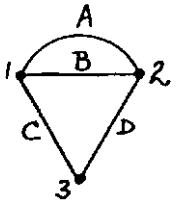
5. Go back to step 2 if any uncancelled graphs remain, being otherwise the tree list is complete.

To implement the above algorithm on a digital computer the graphs and sub-graphs are represented internally by binary incidence matrices. Figure 5(a) shows a graph g_1 and its incidence matrix. Each column of the matrix represents a branch of the network, while each row represents a node. Note that branch C is connected to nodes 1 and 3 but is not connected to node 2. This is indicated in column C of the incidence matrix with entries of 1 in rows 1 and 3 and with entry 0 in row 2. Note that this incidence matrix is the transpose of the previous one used by the new algorithm.

The operation of deleting a branch is shown in Figure 5(b) where branch A is deleted by entering zeros in all rows of column A. The shorting operation is shown in Figure 5(c), where the branch A is shorted by performing a modulo-two (exclusive or) addition of the rows of the incidence matrix which correspond to the two nodes of the branch being shorted. Note that this operation also removes any loops that might be formed at the newly formed node (ie. a loop of branch b in this case).

For a graph containing b branches and n nodes the maximum secondary storage requirement (on disc packs) is when b-1 graphs have been formed in one string. Therefore, the maximum number of computer locations L required at any one time is

$$L = [n(b-1) + (n-2)(b-2) + \ldots + \{b-(n-1)\}2]$$



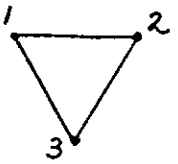
Graph g_1

Node Branch

	A	B	C	D
1	1	1	1	0
2	1	1	0	1
3	0	0	1	1

Select A

(a)

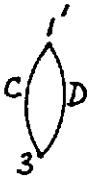


Graph g_2

	A	B	C	D
1	0	1	1	0
2	0	1	0	1
3	0	0	1	1

Delete A

(b)

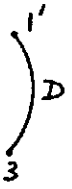


Graph g_3 Assoc. with A

	A	B	C	D
1'	0	0	1	1
3	0	0	1	1

Short A (ie. R_1 and R_2) and delete graph g_1

(c)



Graph g_4 Assoc. with A

	A	B	C	D
1'	0	0	0	1
3	0	0	0	1

Select and delete C

(d)

Graph Assoc. with A and C

	A	B	C	D
1''	0	0	0	0

Short C(ie. R_1' and R_3) and delete graph g_3

One-node, record AC in tree list

(e)

Graph

	A	B	C	D
1'	0	0	0	0
3	0	0	0	0

Delete D

Isolated node, cancel graph

(f)

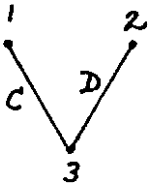
Graph Assoc. with A and D

	A	B	C	D
1''	0	0	0	0

Short D of graph g_4 and delete graph g_4

One-node, record AD in tree list

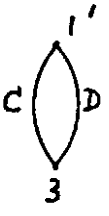
(g)



	A	B	C	D
1	0	0	1	0
2	0	0	0	1
3	0	0	1	1

Delete B of graph g_2

(h)



	A	B	C	D
1'	0	0	1	1
3	0	0	1	1

Short B of graph g_2

Graph g_4, Associated with B. Succeeding steps leads to trees BC and BD.

(i)

Selecting Graph g_3 of Figure 7(h) results in the last tree CD.

Tree list: AC,AD,BC,BD,CD

Δ = AC+AD+BC+BD+CD

(j)

Figure 5: Example of Minty's Graph Reduction Process to Generate Trees

The trees generated by this algorithm are ordered, with the lowest lettered (or numbered) branch appearing first. If trees which contain a specified branch or a number of specified branches are to be distinguished, this may be accomplished for one or two branches by numbering these branches low.

Testing and Comparison of the Algorithms

Programs for both the new algorithm and Minty's Algorithm were written for execution on an IBM System 370/155 Computer. The programs were written in both FORTRAN IVG and Assembler. The FORTRAN programs made use of assembler written functions for performing AND, OR and Exclusive OR operations on 32-bit words. As expected, the Assembler programs were up to five times faster than the FORTRAN programs because the Boolean operations on words are used by both algorithms in the inner loops. The assembler programs are faster because they can use the Boolean operations on words in-line without incurring the overhead of the FORTRAN callable assembler-cooled functions.

The tests indicated that Minty's algorithm was slower by a factor of 2.5 to 16 for graphs of 5-nodes and 6-branches (11 trees) to 15-nodes and 22-branches with 5,103 trees. The speed difference between the two algorithms increased with the size of the graph. Tree generation rates of 100 to 1000 trees per second were observed for the new algorithm.

Chase [2] has classed Minty's algorithm as belonging to a group of connected expansion algorithms and his analysis gives similar results for connected-expansion and for a group of method termed Circuit-Free Expansion. Chase shows that these groups of algorithms were the "best" for the case where the trees were explicitly generated. Factoring algorithms which implicidly generated the trees were shown to be faster, but the trees are left in a factored form.

In programming and running the two algorithms on the computer it was discovered that they generated trees in the same order. This enables the two algorithms to be closely compared. Chase termed Minty's algorithm a connected expansion algorithm where each selected branch is connected to its previously selected branches. An invalid tree is not detected until all subgraphs are expanded. The new algorithm can, in many cases, detect an invalid tree without processing all the branches of the tree candidate. In addition, the number of computer instruction involved in processing a valid tree is less in the new algorithm.

There appears to be a good possibility of speeding up the new algorithm by not generating tree candidates to be tested in the first place if they contain branches which correspond to previously detected loops.

References

1. Minty, G. "A Simple Algorithm for Listing All the Trees of a Graph". IEEE Transactions on Circuit Theory, Vol. CT-12, p. 120, March 1965.

2. Chase, M.C. "Analysis of Algorithms for Finding All Spanning Trees of a Graph". Report 401, Dept. of Computer Science, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., 61801, October 19, 1970.

3. IMSL Numerical Computations Newsletter, No. 8, Jan., 1975. Sixth Floor, GNB Bldg., 7500 Bellaire Blvd., Houston, Texas, 77036.

4. Parlett, B.N. and Wang, Y. "The Influence of the Compiler on the Cost of Triangular Factorization", (accepted for TOMS).