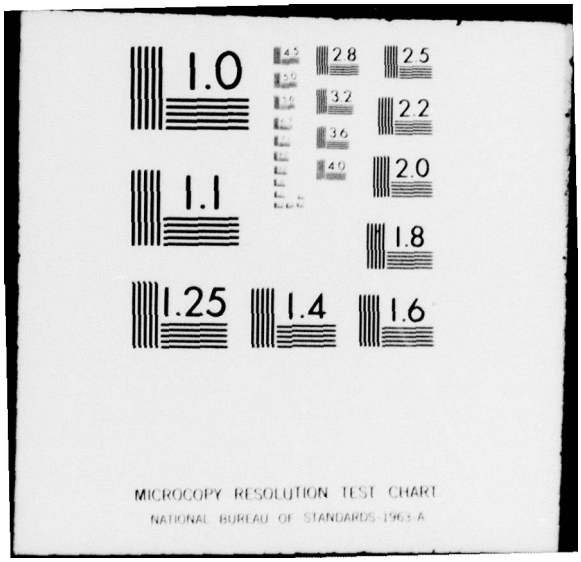
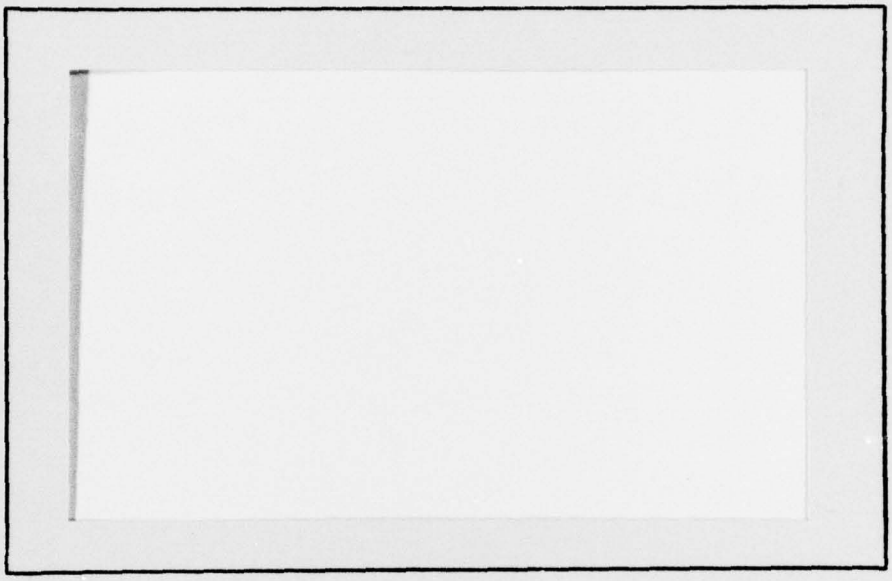
AD-A059 907 CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENC--ETC F/G 12/1
TREELESS SEARCHES.(U)
OCT 76 C E BLAIR, R G JEROSLOW
UNCLASSIFIED MSRR-396 N00014-75-C-0621
NL

END
DATE FILMED
12-78
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

LEVEL II

12

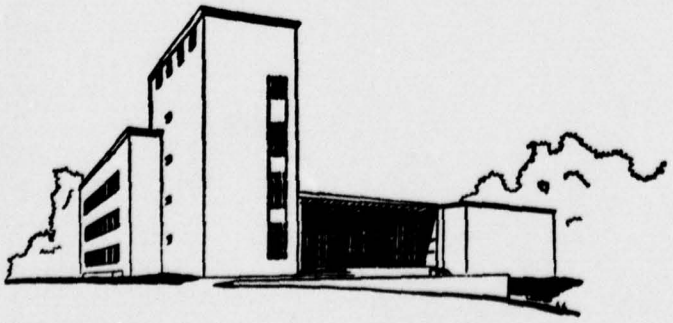


Carnegie-Mellon University

PITTSBURGH, PENNSYLVANIA 15213

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION

WILLIAM LARIMER MELLON, FOUNDER



(9) Management Sciences Research Report, No. 396

(14) MSRR-396

(6) TREELESS SEARCHES •

(10) C.E. Blair and R.G. Jeroslow

(11) October, 1976

(12) 23 p.

(15) N00014-75-C-0621,
NSF-GP37510

This report was prepared as part of the activities of the Management Sciences Research Group, Carnegie-Mellon University, under Grant #GP 37510 X1 of the National Science Foundation and Contract N00014-75-C-0621 NR047-048 with the U.S. Office of Naval Research. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

Management Science Research Group
Graduate School of Industrial Administration
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

403 426

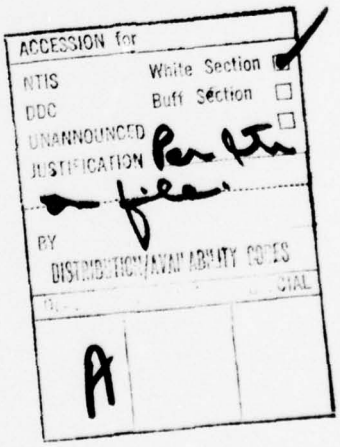
403 426

Abstract

We demonstrate that there are natural heuristics for partial enumeration, that are not based on tree structures for guiding the enumerative search, nor can these heuristics be implemented in any tree-search framework. We also provide means for "redrawing the tree," when the current state of a tree search makes it desirable to utilize a different tree representation of current information.

Key Words

1. Integer programming
2. Branch-and-bound
3. Implicit enumeration



TREELESS SEARCHES

by C.E. Blair and R.G. Jeroslow

0. Introduction

The most successful algorithms to date for the practical solution of integer programs are based on methods of partial enumeration, and often derive their fathoming power from linear programs and cutting-planes (see e.g., [1]). Here we examine some basic conceptions of how partial enumerations proceed, and identify "hidden assumptions" which can lead to overlooking devices that are likely to limit the size of the enumeration.

Specifically, we demonstrate that there are natural heuristics for partial enumeration, which entail enumerative searches that are not based on tree structures for guiding the next state of the search. In addition, we provide means for "redrawing the tree." Tree-redrawing involves, first, the consolidation or "merging" of alternative problems which can be combined into one without much loss of information; and second, the representation of the set of merged subproblems via a tree structure which does not require very many new nodes or new linear programs to be evaluated.

From a practical perspective, the techniques of this paper can be useful when a search has "mushroomed," and become excessively large in a certain way, specifically the same relatively few variables are branched on again and again at many different parts of the tree.

This kind of phenomenon has been observed in practice, usually in connection with integer programs which failed to run successfully by automatic methods, until they were restarted with a different set of initial branchings specified in advance.

1. Treeless searches

In a tree-guided partial enumeration for bivalent integer programs, at any given point of time the current state of the search can be represented by the set of bottommost nodes of a tree (the "leaves" of the tree) that denote partial solutions, both unfathomed and fathomed. The tree structure itself describes the history of the search.

In Figure 1, let A_j denote the condition $x_j = 1$ and $\bar{A}_j$ denote $x_j = 0$; also temporarily abbreviate the letters A_1, A_2, A_3 as A,B,C. Then Figure 1 constitutes the tree structure behind the search in which the first branching variable was x_1. From the tree, we see that x_2 was the branching variable on the branch $x_1 = 0$; thereafter x_3 was the branching variable on both branches $x_2 = 1$ and $x_2 = 0$. Similarly, x_3 was the branching variable on the branch $x_1 = 1$; and x_2 thereafter was the branching variable for both $x_3 = 1$ and $x_3 = 0$. In Figure 1, no bottommost partial solution has been fathomed.

The tree of Figure 1 tells us the entire history of how we obtained the eight bottommost nodes, just below which we have written the status of fixed variables in an obvious symbolism. However, it is really the leaves themselves which aid in specifying the current state of the search; the history is interesting only if it helps us to resolve the integer program.

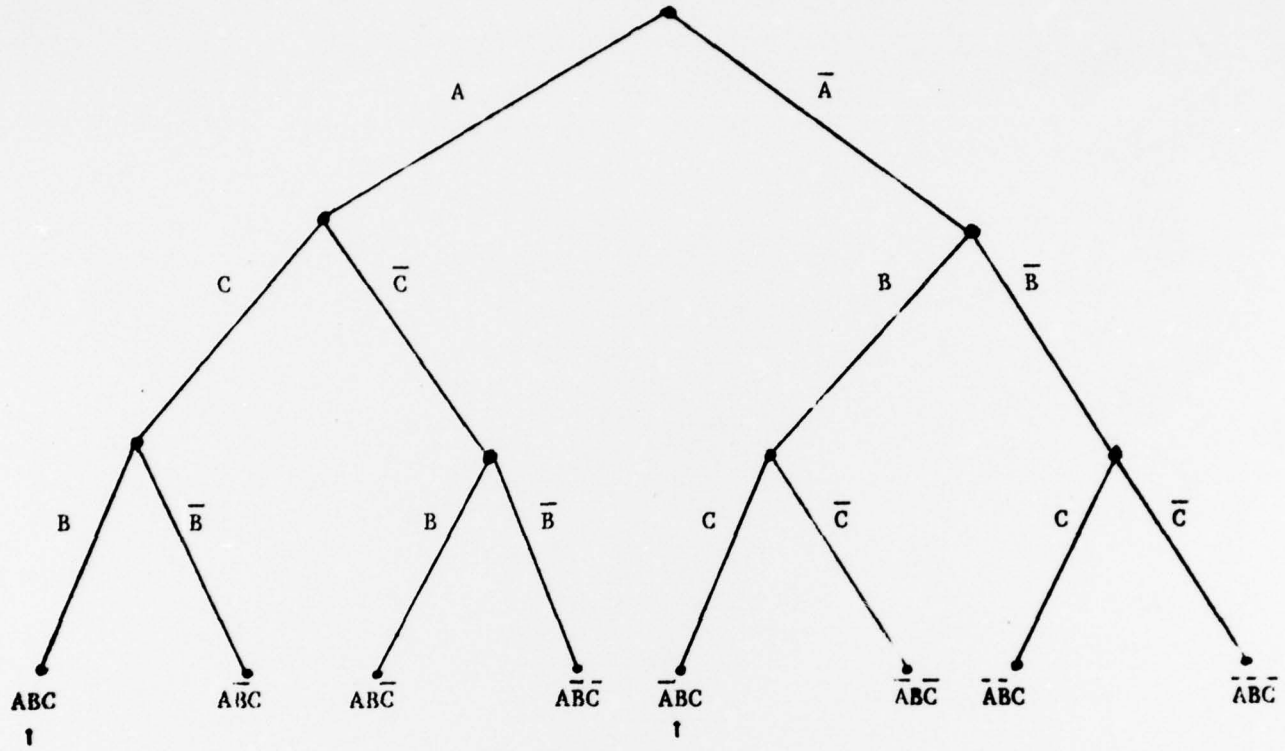

A
$\bar{A}$
C
$\bar{C}$
B
$\bar{B}$
B
$\bar{B}$
B
$\bar{B}$
C
$\bar{C}$
C
$\bar{C}$
ABC
A$\bar{B}$C
A$\bar{B}\bar{C}$
$\bar{A}$BC
$\bar{A}\bar{B}$C
$\bar{A}\bar{B}\bar{C}$

Figure 1

We may have current indications that at the node denoted ABC the setting $x_1 = 1$ may be of very little value (for one such indication, see Sec. 2. below). The criterion value of the associated linear program at ABC may be very little changed if the implicit constraint $x_1 = 1$ is replaced by $0 \leq x_1 \leq 1$. Possibly the setting $x_1 = 0$ is also of little value at $\bar{A}BC$. In such a case, instead of proceeding "forward" and chosing another branching variable at ABC, we may actually want to move "backward" and merge nodes ABC and $\bar{A}BC$ into one node.

The difficulty with merging the nodes ABC and $\bar{A}BC$ into a new node BC, is that these nodes occur at very different parts of the tree structure. Had the historical order of fixing variables first been B and then C, followed by A and $\bar{A}$, there would be no difficulty; we would simply decide that x_1 was not the proper branching variable at node BC, and we would chose another. However, BC is not the historical order, and now if we desire to keep the tree structure we simply can't do the merge that intuitively we ought now to do. Obviously, the tree structure has to go.

Indeed, the tree structure has to go, but not simply to save one linear program, by combining two nodes to one. The nodes ABC and $\bar{A}BC$ can be so similar, that virtually the same branching variables will be chosen in refining each node, so that whole sections of the search tree will become unnecessarily duplicated.

From Figure 1, when we chose x_1 as the first branching variable, it was because the dichotomy $x_1 = 1$ versus $x_1 = 0$ appeared then to be very significant, probably because one of the two alternatives was likely to be easily fathomed. However, given BC representing the settings $x_2 = 1$ and $x_3 = 1$, evidently the effect of the variable x_1 is now inconsiderable. This would be the case, for example, if the location of a warehouse at site 2($x_2 = 1$) or site 3($x_3 = 1$) is unlikely to be helpful, given the overall situation, and in view of this fact, the location of a warehouse at site 1 ($x_1 = 1$) or not ($x_1 = 0$) is clearly a crucial decision; but if both site 2 and site 3 are (unexpectedly) utilized, then a warehouse at site 1 might be somewhat useful, but only marginally so.

Originally, the state of our current search can be represented in terms of the bottommost nodes of the tree of Figure 1, in this way:

(1) $ABC \vee A\bar{B}C \vee AB\bar{C} \vee A\bar{B}\bar{C} \vee \bar{A}BC \vee \bar{A}B\bar{C} \vee \bar{A}\bar{B}C \vee \bar{A}\bar{B}\bar{C}$

Suppose that the following merges are found to be advantageous and are performed:

(2a) ABC and $\bar{A}BC$ to BC

(2b) $A\bar{B}C$ and $A\bar{B}\bar{C}$ to $A\bar{B}$

(2c) $\bar{A}B\bar{C}$ and $\bar{A}\bar{B}\bar{C}$ to $\bar{A}\bar{C}$

Then the state of the search will be represented by:

(3) $BC \vee A\bar{B} \vee AB\bar{C} \vee \bar{A}\bar{C} \vee \bar{A}\bar{B}C$

which has only five subproblems in place of eight. Upon reaching (3), with no further merges possible, the search can be resumed only by again moving "forward," i.e., by chosing some one subproblem and branching on a new variable.

Is there a tree structure with (3) as its leaves? To put the question another way, if we had in advance the knowledge we now have, about the relative affect of different branchings from different nodes, could we have figured out the "right" variable to branch on at the start and subsequently?

The answer to this question is "no," since any one branching variable at the top of the tree gives some letter P such that either P or $\overline{P}$ must occur in all leaves of the tree, but no such letter P exists for (3).

Our analysis just previous of course does not demonstrate that a good enumerative search must of necessity diverge from a tree structure during at least some parts of the search; possibly most such searches do have tree representations for their histories. We have seen, however, that guiding a search by reference to a tree structure requires a special effort in representing past events, and can get in the way of natural heuristics.

It is certainly true that treeless searches can require more "bookkeeping" than a LIFO ("depth-first") strategy, but the latter has been proven decidedly inferior to flexible backtracking [5] and

is not used in most commercial codes. Fast list-processing subroutines that save whole linear programs are often to be preferred to simplistic approaches.

2. Merging and "reverse penalties."

Merging is possible when two nodes have descriptions

(4) $$A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}A_{j(r)}$$

and $$A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}\bar{A}_{j(r)}$$

that are identical except for exactly one "opposition," in which $A_{j(r)}$ is opposite $\bar{A}_{j(r)}$. (In (4), the order in which the description is written is immaterial ($A_{j(1)}A_{j(2)}$ and $A_{j(2)}A_{j(1)}$ are viewed as the same logical condition), and certainly has nothing to do with the historical order of fixing variables). If a merging is performed, it will result in a node described as $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}$. Mergings may be iterated, with, e.g., ABC and $AB\bar{C}$ merged to AB, $A\bar{B}C$ and $A\bar{B}\bar{C}$ merged to $A\bar{B}$; and then AB and $A\bar{B}$ merged to A. Also A and $\bar{A}$ can be merged to $\emptyset$, signifying that no variable has been fixed at any value.

When would the merging of the two nodes described in (4) be advantageous?-this important question can be rephrased in a more useful way as follows: if we examined a node $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}$, when would we decide not to branch on variable $x_{j(r)}$, provided we had any reasonable alternative? Clearly, we would not branch on $x_{j(r)}$ if the less promising of the two resulting subproblems after

branching is not "significantly" less promising than $A_{j(1)}A_{j(2)}\cdots A_{j(r-1)}$ itself, with "significance" possibly measured by a "threshold difference" $\Delta \geq 0$.-Here Δ can be determined by a prior expectation modified with exponential smoothing by criterion value changes actually experienced.

The linear program associated with $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}$ has a value for each different right-hand-side, and as this right-hand-side is parametrically varied from $x_{j(r)} = 0$ to $x_{j(r)} = 1$, this value is given by a function $f(w)$ of the value w of $x_{j(r)} = w$. From theory, assuming a minimizing program, we know that $f(w)$ is a piecewise linear convex function of w in the interval $0 \leq w \leq 1$, and clearly the linear programming value at $A_{j(1)}A_{j(2)}\cdots A_{j(r-1)}$ is the minimum value v of $f(w)$ on the interval $0 \leq w \leq 1$, since the constraint $0 \leq x_{j(r)} \leq 1$ occurs among those of the linear program. The rightward slope S_D of $f(w)$ at $w = 0$ is easily obtained via parametric linear programming, or by use of suitable reduced costs (as done in [2], [3]); and similarly one easily computes the leftward slop S_U of $f(w)$ at $w = 1$.

Clearly, by the convexity of f, if $S_D \geq 0$ then v is also the value of the problem at $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}\bar{A}_{j(r)}$, and the optimum at this latter node (which already has $x_{j(r)}$ integral) is optimal for $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}$; so merging is recommended, because we would never have branched on $x_{j(r)}$ given the situation at the node $A_{j(1)}A_{j(2)}\cdots A_{j(r-1)}$. Again, if $S_U \geq 0$, the situation favors a merge.

Now assume $S_D < 0$ and $S_U < 0$, let v_D denote the value of the program at node $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}\bar{A}_{j(r)}$, and let v_U denote the

value at $A_{j(1)}A_{j(2)} \cdots A_{j(r-1)}A_{j(r)}$. From convexity, we have both $f(w) \geq S_D w + v_D$ and $f(w) \geq S_U(w-1) + v_U$. It follows at once that

$$(5) \qquad v \geq \min \left\{ \max\{ S_D w + v_D, S_U(w-1) + v_U \} \mid 0 \leq w \leq 1 \right\}$$
$$= \frac{S_D v_U - S_U v_D - S_D S_U}{S_D - S_U}$$

Consequently, merging is certainly recommended if the right-hand-side of (5) plus Δ exceeds $\max\{v_D, v_U\}$ (in a minimizing problem).

The right-hand-side in (5) leads to a "reverse penalty," and was derived by virtually reversing the analysis that provides ordinary up-down penalties [1]. Just as penalties provided a lower bound on criterion deterioration after branching, reverse penalties provide an upper bound on criterion improvement after merging. Reverse penalties are therefore conservative, and merging may be recommended even when the previous reverse penalty fails to signal this fact. As with penalties, one obtains better reverse penalties if several pivots are performed in parametric programming.

Clearly, after the merging of the two nodes described in (4), $x_{j(r)}$ should not be used as a branching variable, unless no very promising branching variable is available. This simple rule is probably sufficient to avoid cycling in many cases, but cycling can occur when e.g., $\emptyset$ is branched to $A \vee \bar{A}$ and then A and $\bar{A}$ are merged to $\emptyset$. To insure rigorously against cycling, one may store a representation of every node which resulted from a merge, and a list of variables for

the merges resulting in this node; one then forbids a merge which repeats a variable in the list.

In examining the current state of the search, and looking for all possible merges for a specific current node, the whole set of current nodes must be scanned. It is not difficult to write conditions for detecting possible "partners" for the given node, and to update these dynamically; however, this practice results in a list of current notes being associated with each current node. It is probably more efficient to simply note the <u>length</u> of each current node, to compare nodes of a specific length against the one or two nodes just created, and see if they have precisely one opposition: one then applies reverse penalties to that one opposition if they do.

3. <u>The systematic use of merging</u>.

To understand the theoretical limits of the branch-merge-and-bound methods we have outlined above, a certain degree of abstraction is necessary.

We shall find it helpful to say that a logical condition

$$B_1 \vee B_2 \vee \ldots \vee B_t \tag{6}$$

is a (disjunctive) <u>normal form tautology</u>, or n.f.t., if each B_k ($1 \leq k \leq t$) is the conjunction of atomic letters A_j and their negations $\bar{A}_j$ (with no B_k containing both an A_j and $\bar{A}_j$), and if (6) is always true whether each A_j is true or false. We shall call (6)

a <u>disjoint tautology</u> if each pair B_h, B_k exclude each other, i.e., some letter A_j occurs in both B_h and B_h but in opposition. By definition, $\emptyset$ is a disjoint tautology.

<u>Theorem</u>: The repeated use of branching and merging produces disjoint tautologies. Moreover, any disjoint tautology can be obtained in this manner.

<u>Proof</u>: Since $\emptyset$ is a disjoint tautology, to prove the first assertion we need only establish that branching and merging preserve the disjointness of a disjoint tautology (6). For branching, the result is immediate. For merging, suppose that B_h and B_k are merged into P, and let B_p be other than B_h or B_k. Then the truth value of $B_p \wedge P$ is that of $B_p \wedge (B_h \vee B_k)$, i.e., that of $(B_p \wedge B_k) \vee (B_p \wedge B_k)$, hence $B_p \wedge B$ is always false. Therefore B_p and B must have at least one letter in opposition.

For the second assertion, let $\{A_1, A_2, \ldots, A_t\}$ contain all the letters occurring in any B_k ($1 \leq k \leq t$). By repeated branching, one obtains a disjoint tautology of which the general term is $A_1^{\pm 1} A_2^{\pm 1} \ldots A_t^{\pm 1}$, using the abbreviation

$$A_j^i = \begin{cases} A_j & \text{if } i = +1 \\ \bar{A}_j & \text{if } i = -1 \end{cases} \tag{7}$$

Each such term is consistent with exactly one B_k in (6). By grouping together that set of terms consistent with B_k and repeatedly merging, exactly B_k is obtained.

Q.E.D.

3.1 Non-disjoint normal form tautologies.

A consideration favoring disjoint tautologies is that each subproblem is more tightly constrained than in normal form tautologies where there may be overlap among the leaves. This consideration could become minor if certain non-disjoint n.f.t.'s can be shown to provide more "information" than any disjoint tautology with the same number of leaves.

How much "information" a given node provides - in terms of criterion value, closeness to integrality, etc. - depends of course on the exact integer program to be resolved. In general, one node B_k can be guaranteed to provide as much information as another B_h only if it is uniformly more tightly constrained, i.e., if B_k contains all the fixed variables of B_h fixed at their values for B_h (and may contain more fixed variables). When this latter condition occurs, we say that B_k is a refinement of B_h. Also, one n.f.t. is a refinement of another, if each of its leaves is a refinement of at least one leaf of the other.

Here is a non-disjoint n.f.t., which possesses the property that its every disjoint refinement has more leaves:

$$AB \vee CD \vee \bar{A}\bar{C} \vee \bar{A}\bar{D} \vee \bar{B}\bar{C} \vee \bar{B}\bar{D} \tag{8}$$

The reader can easily check that (8) is a tautology; it is clearly not disjoint.

Suppose that the following disjoint tautology is a refinement of (8):

$$B_1' \vee \ldots \vee B_t' \ . \tag{6}'$$

We shall say that B'_k is <u>unique</u> to a leaf of (8)if it refines that leaf and only that leaf. To prove that t' exceeds six (the number of leaves of (8)), it clearly suffices to show: (i) Each leaf of (8) has a leaf of (6)' that is unique to it; (ii) At least one pair of leaves of (8) have at least three distinct B_h 's in (6)' that are refinements of at least one node of the pair and are not refinements of any leaf of (8) not in the pair.

To establish (i), note that there are sixteen truth valuations of the four letters A,B,C,D, and that each leaf of (8) has a valuation that makes it true and all other leaves of (8) false. This valuation makes at least one leaf B'_k of (6)' true, and clearly B'_k must be unique to the given leaf of (8).

To establish (ii), note that any refinement of AB or of CD cannot be a refinement of the other leaves $\bar{A}\bar{C}$, $\bar{A}\bar{D}$, $\bar{B}\bar{C}$, or $\bar{B}\bar{D}$. There cannot be only two refinements of both AB and CD in total, for then these two would be AB and CD themselves, and hence would not be disjoint. Hence there are at least three B_k's which are refinements of AB or CD.

This completes our proof that t' exceeds six.

There is a result analogous to the previous theorem for nondisjoint n.f.t.'s, in which the simple merging operation is replaced by a "copy and merge" operation. Specifically, one allows several "copies" of a given leaf to be made, each different one of which is merged with another leaf, to which it has an opposition at a different variable.

The proof of the previous Theorem is then applicable, virtually unchanged except for the fact that one copy of $A_1^{\pm 1} A_1^{\pm 1} \ldots A_1^{\pm 1}$ is made for each B_k in (6) with which it is consistent. The branch-copy-merge-and-bound approach can, in theory, produce any n.f.t., disjoint or not.

4. Redrawing the tree.

The concept of merging can also be used to suggest alternate tree structures for a tree-based search already underway.

We illustrate the technique by an example. Suppose the tree of Figure 1 has the advantageous merges (2a) to (2c), after which the state of the search would be given by (3). (In general, we allow the case in which no merges are advantageous, or in which one chooses to do no merges).

To represent the current state (3) at least partially in tree form, there would have to be a "first" branching variable with corresponding letter P, such that P or $\bar{P}$ occurs in each leaf. As noted in Section 1, no such letter exists for (3). But if e.g., we wished to branch on x_2, we can artificially have both B and $\bar{B}$ occur in each leaf, by viewing $\bar{A}\bar{C}$ (which is the only leaf not containing B) as $\bar{A}B\bar{C} \vee \bar{A}\bar{B}\bar{C}$, which causes an increase in the number of leaves. (We will see in Figure 2 that this "unmerging" need not lead us back to the original tree of Figure 1).

The general situation in tree re-drawing is similar to that of the example. The search to the present point of time is unsatisfactory; one has picked up information indicating that certain variables, branched on lower in the tree, may be more "significant" that the variables branched on by the automatic procedure toward the top of the

tree. One wants to perform merges and proceed with a treeless search; one first performs these merges, at least symbolically, but the necessity of using a branching code forces us to tree redrawing.

After one has selected the variable x_j that one wishes to make the first branching variable, every leaf B_k, in which neither A_j or $\bar{A}_j$ appears, must be doubled to become $B_k A_j \vee B_k \bar{A}_j$; then this technique is repeated inductively with the resulting branches for A_j and $\bar{A}_j$, and whatever variables are viewed as good choices for "second" branching variables. In principle <u>any</u> first branching variable x_j can be chosen.

Clearly, the consideration of avoiding very many "doublings" leads to chosing a branching variable x_j such that the number of leaves in which either A_j or $\bar{A}_j$ appear (i.e., the number of leaves which will not have to be doubled) is large, if not maximum. This approach to limiting the choice of branching variable is also in accordance with the observation, that one really does not have good information throughout the tree about a variable branched on only at a few places. In our specific example, all three of the letters A,B,C or their negations appear in all but one leaf, so the choice of first branching variable is left to other heuristics, quite possibly involving the observed criterion value deterioration with each branching on a given variable.

Suppose that x_2 is chosen as first branching variable in (3). Then we view $\bar{A}\bar{C}$ as two duplicates $\bar{A}B\bar{C} \vee \bar{A}\bar{B}\bar{C}$, and consequently, on the branch $x_2 = 1$ we have the problem $BC \vee \bar{A}B\bar{C} \vee AB\bar{C} = B\cdot(C \vee \bar{A}\bar{C} \vee A\bar{C})$;

the method is repeated on $C \vee \bar{A}\bar{C} \vee A\bar{C}$. This time, C and $\bar{C}$ occur in each leaf, so x_3 is the branching variable. For $x_3 = 0$, we have only C; the method terminates. For $x_3 = 1$, we must consider $\bar{A}\bar{C} \vee A\bar{C} = \bar{C} \cdot (\bar{A} \vee A)$, and we repeat with $\bar{A} \vee A$, which is a simple branching on x_1.

On the branch $x_2 = 0$, we have the problem $A\bar{B} \vee \bar{A}\bar{B}\bar{C} \vee \bar{A}\bar{B}C = \bar{B} \cdot (A \vee \bar{A}\bar{C} \vee \bar{A}C)$, hence the method applies to $A \vee \bar{A}\bar{C} \vee \bar{A}C$ and the branching variable is x_1; etc. The results are shown in Figure 2, where nodes which are not among the leaves of (3) are branched to on dotted lines.

In Figure 2, we have six nodes, in place of the five leaves of (3) or the eight leaves of Figure 1. Only the values of the two "artifical" nodes $\bar{A}B\bar{C}$ and $\bar{A}\bar{B}\bar{C}$ cannot be obtained by merging alone, and would necessitate solving a new linear program.-One may tentatively use the value of the merged problem $\bar{A}\bar{C}$ for fathoming at both these nodes, and one never needs to actually solve either of the two additional linear programs, unless a backtracking heuristic selects one of these for further examination.

For related tree redrawing techniques in more restrictive circumstances, see [2], [3] or [6].

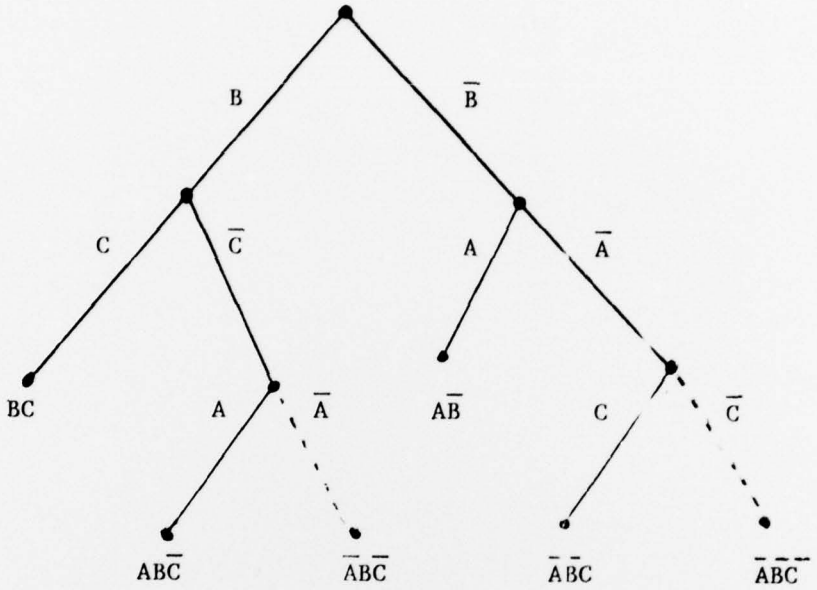

B
$\overline{B}$
C
$\overline{C}$
A
$\overline{A}$
BC
A
$\overline{A}$
$A\overline{B}$
C
$\overline{C}$
$AB\overline{C}$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$\overline{A}\overline{B}\overline{C}$

Figure 2

5. Refinements.

In our examples and discussion above, the individual atomic letters A_j resp. $\bar{A}_j$, which were conjoined to form leaves, represented $x_j = 1$ resp. $x_j = 0$. For more generality, the atomic letters can represent any system of linear inequalities and equalities, such that repeated branching exhausts all the logical possibilities.

For instance, in bivalent programming one may add atomic letters C_{jk} representing $x_j = x_h$ and $\bar{C}_{jk}$ representing $x_j = 1 - x_k$. This is the "cross-branching" that the second author introduced in [4]; it is useful when, for instance, there are prior expectations as to when two projects are likely to be done together if at all ($x_j = x_h$), or are basically opposed projects ($x_j = 1 - x_k$).

In the present framework, the cross-branching allows a novel merging in addition to the obvious merges like B_iC_{jk} and $B_i\bar{C}_{jk}$ to B_i, for a leaf B_i. Specifically, we have the "two oppositions merge" of $B_iA_jA_k$ and $B_i\bar{A}_j\bar{A}_h$ to B_iC_{jk}, as well as $B_iA_j\bar{A}_h$ and $B_i\bar{A}_iA_h$ to $B_i\bar{C}_{jk}$. Just as with ordinary branching, the cross-branching reduces dimension by one.

6. Acknowledgements.

The second author has been influenced by Fred Glover's emphasis on the need to re-examine the basic conceptions behind enumerative algorithms [2].

We particularly wish to thank Roger Heddin of UNIVAC for an interesting discussion of problems arising during tree-searches for solving integer programs from industry, in which he pointed out the frequency of tree structures with multiple occurrences of branching on the same set variables. The term 'tree redrawing" is his.

Carnegie-Mellon University
University of Illinois
October 8, 976

References

1. R.S. Garfinkel and G.L. Nemhauser, Integer Programming, John Wiley & Sons, New York, 1972. 390+ pp.

2. F. Glover and L. Tangedahl, "Dynamic Strategies for Branch and Bound," OMEGA 4 (1976), pp. 1-6. Given as an invited talk by F. Glover at the Workship in Integer Programming, University of Bonn, September 1975.

3. F. Glover and L. Tangedahl, "Dynamic Branch and Bound Strategies Using Tree Manipulations," talk at the Fifth Annual Meeting, American Institute of Decision Sciences, Western Regional Conference, March 1976. Available as a manuscript, 4pp.

4. R.G. Jeroslow, "Cross-branching in Bivalent Programming," MSRR no. 331, GSIA, Carnegie-Mellon University, March 1974.

5. C.J. Piper, Computational Studies in Optimizing and Postoptimizing Linear Programs in 0-1 Variables, Ph.D. dissertation, GSIA, Carnegie-Mellon University, 1975.

6. N. Ph. Tuan, "A Flexible Tree Search Method for Integer Programming Problems," Operations Research 19 (1971), pp. 115-119.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: M.S.R.R. 396	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Treeless Searches	5. TYPE OF REPORT & PERIOD COVERED: October, 1976
	6. PERFORMING ORG. REPORT NUMBER: M.S.R.R. 396
7. AUTHOR(s): C.E. Blair and R.G. Jeroslow	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0621
9. PERFORMING ORGANIZATION NAME AND ADDRESS: GSIA, Carnegie-Mellon University, Pittsburgh, PA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 047-048
11. CONTROLLING OFFICE NAME AND ADDRESS: Personnel and Training Research Programs, Office of Naval Research (Code 458), Arlington, Virginia 22217	12. REPORT DATE: October, 1976 13. NUMBER OF PAGES: 17
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Integer programming, Branch-and-bound, Implicit enumeration.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
We demonstrate that there are natural heuristics for partial enumeration, that are not based on tree structures for guiding the enumerative search, nor can these heuristics be implemented in any tree-search framework. We also provide means for "redrawing the tree," when the current state of a tree search makes it desirable to utilize a different tree representation of current information.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)