Research Report Number 70-1

Department of Computer Science

YALE UNIVERSITY

New Haven, Connecticut 06520

October, 1970

MULTIPLE-TRACK PROGRAMMING

by

Edgar T. Irons

#1
MULTIPLE-TRACK PROGRAMMING

Edgar T. Irons
Yale University, New Haven, Connecticut

Key words: Multiple-track, backtrack, games, pattern recognition, parsing.

Abstract: Multiple-track programming and backtrack programming are defined in terms of traversing a binary tree. The two methods are compared for trees arising from game playing, pattern recognition, and parsing.

1. Introduction

A backtracking or multiple-tracking organization is suitable for computer programs in which several different solutions to a problem must be carried along for more than one stage of the analysis. The following space flight scenario illustrates such a situation. A spaceship is proceeding on a mission using earth based computers for assistance. Instrumentation on the ship provides redundant information to compute results needed for future maneuvers. At a point in a mission, a disruption occurs which causes a significant contradiction between two instruments; both readings are a priori reasonable, but cannot at the same time be correct. At a future point in the mission, additional data can be available from manual observations which will determine which instrument is malfunctioning. Shortly after these observations, however, a maneuver must be made which depends critically on the information coming from the two contradictory instruments. The ground
based computer can be prepared to cope with this situation in two ways. The first is to record the state of the analysis at the point just before the instruments began to disagree, then to arbitrarily assume one of them is correct and to proceed with the analysis, but recording all incoming data. When additional data can be obtained to determine which instrument is working, the analysis proceeds normally if the right guess was made. If the guess was wrong, the computer must be reset to the point of disagreement, and the computation must be redone using data from the other instrument. This procedure is a backtracking algorithm. The corresponding multiple-tracking procedure treats the contradiction differently. Rather than picking one of the contradictory data streams and ignoring (but recording) the other until resolution is possible, the multiple-tracking algorithm pursues both paths at once. This requires making a second copy of the state of the analysis at the point of disagreement, and then updating each copy as more information comes in. The continuing analysis is now done twice at each stage, once for each copy of the state, one using data from one instrument and the other using data from the other. When resolution of the conflict takes place, the copy which has been updated from the correct instrument is kept and continued, the other is discarded.

Which of these procedures is most satisfactory depends on three characteristics of the system. 1. The amount of data which must be kept in either case may differ significantly. In the backtracking analysis, one copy of the state of the analysis and a copy of the input during the ambiguous situation must be kept. In the multiple-tracking system only the second state must be kept. 2. If special characteristics of the problem indicate that some paths of analysis are more likely to be correct than others, backtracking and selecting the most likely path first will save computation. Even if no special knowledge is available, the backtracking system would be expected to avoid the double calculation at least half the time. 3. Time constraints in some systems can force a choice of one or the other procedure. In the hypothetical space flight, it could be the case that not enough time is available between resolution of the conflict and the subsequent maneuver which depends on this resolution to carry out the updating of the alternate analysis under backtracking. On the other hand, in the absence of time critical factors such as this, backtracking may allow smaller computation
facilities to be used than one which can carry out two or more analyses at once. We will return to these points after a formal discussion of backtracking and multiple-tracking, and some further examples of their use.

2. Process Trees

We use process trees to describe the branching of analysis which is characteristic of the programs we shall discuss. Process trees are analogous to the state diagrams of automata theory or to the graphs used for critical path analyses. They represent the (actual or possible) progress of a process correlated with time or data streams. Each node of the tree represents a state of the process, and each branch of the tree represents a step from one state to another, and is usually associated with a program which derives one state from another. In the space scenario, each node of the process tree represents a state of the analysis. Each time a new data group is received, the ensuing calculations carry the process to the next state. As long as no disagreement in instrumentation occurs the tree has one branch entering and emanating from each node. When a disagreement arises, two branches emanate from the node where the disagreement occurred, one going to a state calculated according to one reading and the other to a state calculated from the contradictory reading. If a resolution of the conflict occurs, the state last calculated from erroneous data has no branches emanating from it. This indicates termination of calculation based on data now known to be wrong. A process tree corresponding to the space flight scenario is shown in Figure 1. Examples of more process trees are given in later sections giving applications.

3. Tree Traversals

A tree is usually stored in a computer as an equivalent binary tree to avoid having a variable number of branches emanating from a node. The binary tree equivalent of a tree is obtained by using the left (or as we have drawn it, downward) branch of the binary tree to link to the left most subtrees of the original, and the right branch to link to the right sibling tree as in Knuth [1]. Figure 2 illustrates a tree and its binary tree equivalent. Traversing a tree is applying a process to each node of the tree in an orderly way. There are a number of ways in which the nodes can be processed differing in the order in which the nodes are taken. Three frequently used orderings are derived from the tree path illustrated in Figure 3. As defined in
Knuth [1] these are preorder, postorder and endorder. The path drawn around the tree of Figure 3 passes each node at three distinct times: once to the lower left, once to the lower right and once to the upper right. If each node is processed as the path passes to its lower left, the processing is in preorder (1,2,5,10,11,3,6,4,7,8,12,9). If each node is processed as the path passes to its lower right, the processing is in postorder (10,11,5,2,6,3,7,12,8,9,4,1). If each node is processed as the path passes to its upper right, the processing is in endorder, (11,10,5,6,12,9,8,7,4,3,2,1). Programs 1 and 2 process a binary tree in any of these orders; Program 1 is recursively formulated. Program 2 uses an explicit stack. A different order of processing the nodes is illustrated in Figure 4. Here, nodes are processed in an order determined by their distance from the root of the tree, where distance is defined in terms of the original tree as number of branches from the root. In the tree shown in Figures 2,3 and 4, node 1 is the root (distance 0 from the root) nodes 2,3 and 4 are distance 1 from the root, 5,6,7,8,9 distance 2 and 10,11,12 distance 3. In this traversal, all nodes at distance 1 are processed before those at distance 1 + 1 so the processing is in lexical order or (1,2,3,4,5,6,7,8,9,10,11,12) for the trees shown. Program 3 processes the binary tree in this order.

4. Chess and Kriegspiel

Process trees can be used to characterize the progress of a number of games. The process is the playing of the game, and each node of the process tree represents an actual or potential state of the game. In cards, the state of the game will be the location of the cards of the deck in hands or various collections associated with the game. In games involving a board, the positions of pieces on the board may characterize the state of the game. In chess, the state of the game is entirely characterized by the composition of the board, each such state giving the position (of 65 positions including absence) of each of 32 pieces. Moves of the game correspond to branches of the tree. Thus, the distance of a node from the tree root is the number of moves or plays (by all parties) which would have taken place to produce the state at the node. Manipulation of process trees for games is of interest in computer game playing for testing out possible moves to evaluate their relative merit. In games where the state of the game is not completely known
to all players, a process tree can be used to represent potential states of the game for the computer as a player. Bridge and kriegspiel are games of this kind.

In playing out the game for a number of moves ahead, multiple-tracking and backtracking traversals of the process tree offer different advantages. Consider this test playing activity in a computer program for playing chess. Most chess playing programs contain a process for evaluating the board and assigning a figure of merit to it relative to other boards. As a means of deciding which move to make next, typically a program will make all possible moves, evaluate the resulting board from each move and select the actual move which produces the best board. It is usually desirable to also examine possible responses to these moves, and perhaps subsequent moves and responses. In chess, as in most games, the number of possible moves and responses is large enough that carrying this process beyond three or four stages is not practical without apriori elimination of some paths.

Suppose it is decided in the chess program to play out the game completely for two moves ahead. There are roughly \(2^6\) moves possible from each state, so we are contemplating evaluation of \(2^{24}\) boards (a large number for playing in real time). The process tree has an average of \(2^6\) branches from each node. If we choose a multiple-tracking traversal, the storage required for intermediate boards (each requiring about 180 bits) will be exorbitant, for we will need to store \(64\) boards resulting from the possible moves, \(2^{12}\) boards resulting from the responses, and \(2^{18}\) boards from the second move. On the other hand, backtracking traversal requires only storage of three boards and moves. This dramatic difference is a direct consequence of the shape of the process tree. The chess process tree is very fat (many branches from each node and short (a short distance from the root to every node). We see a principle emerging: short, fat process trees suggest backtracking for storage conservation. In Section 6 on parsing we shall note the complementary phenomenon: tall, thin process trees suggest multiple-tracking for storage conservation.

If we prune the tree a bit, however, the situation changes. Since there are so many moves possible at each stage, and since many of them will be silly (a blunder by the computer) or improbably (a blunder by the opponent)
we may choose to eliminate many moves at an early stage, and play out promising ones further. This makes the process tree thinner (by eliminating some branching) and taller (playing more moves). Furthermore, a reasonable strategy might require selecting moves to retain or discard based on the ranking of the resulting boards. Multiple-tracking is now a more attractive procedure. We begin by making all moves from the current board. Then we evaluate all of these boards and select perhaps the 100 most promising, if there are that many. Then we make all possible responses and select the 100 most likely ones (by measuring improvement for the opponent). Subsequent moves and responses with pruning at each stage corresponds to a multiple-tracking traversal of the (pruned) process tree. Backtracking traversal rules out the possibility of comparing boards resulting from alternate moves on the same board.

The use of process trees in Kriegspiel (Goren [2]) is of a somewhat different nature. Kriegspiel is a variant of chess in which each player has a board which his opponent cannot see. The moves of the game are the same as for chess, except that the players can only guess the opponent's moves. A referee watches both boards and provides enough guidance to insure no illegal moves are made, and in some variants some additional information. Thus, the referees reaction to a move may be, "your move is illegal" or "you have taken a rook" or "check". These comments, of course, aid a player in reconstructing his opponent's moves.

For a computer program to play one side of this game, it must be prepared to make guesses about the opponent's moves, and either to store up possible boards or to back up and make alternate guesses in the event contradiction is revealed by the referee. The requirements of this game and the dimensions of the process tree suggest a combination of multiple-tracking and backtracking! At first glance, multiple-tracking seems a natural approach to dealing with uncertainty about the state of the board. The computer makes all possible moves for his opponent resulting in one board for each move, then computes his move as for normal chess, but using a weighted figure of merit for the move on each of the possible moves. As in playing out the game, however, the number of boards is likely to become unmanageable with perhaps $2^6$ possibilities for each unknown move.
A number of these boards can be eliminated by some tricks of play designed to produce illegal moves (see Goren [2]), but the number of remaining states will still be very large, and since we are now actually playing the game rather than testing, we are obliged to continue for many moves. The process tree is now both tall and fat, (although breadth still clearly dominates). Backtracking is not entirely a satisfactory solution, however, as long as it is desired to make decisions based on a number of probable boards. A strategy growing out of the second test move strategy above is to limit the breadth of the tree by selecting the most probable boards and building on those, leaving the others in reserve. We cannot discard the less probable branches, however, for the actual board may indeed be among them. Assuming 64 possible moves at each stage and limiting those being considered to 10, each move of the opponent gives rise to 640 boards, all of which must be remembered, but only 10 of which will be used. When boards are discarded because the referee calls a contradiction to a move, then more boards can be added by returning to earlier boards held in abeyance, and playing them up to the current move. Thus, a backtracking traversal of sorts is imposed on the basically multiple-tracking approach.

A modification of the multiple-tracking system which can save considerable storage is to eliminate duplicate states at the same distance from the root. In doing so we recognize that we may arrive at the same board by several different routes (by, among other ways, making the same moves in a different order) and that only the current state of the board is interesting, not its history. We turn the process tree into a process graph by merging nodes representing identical states at the same level of the tree. This procedure is used in a parsing program to detect ambiguous sentences. (Section 6).

5. Pattern Recognition

Multiple-tracking has an application in providing a suitable interface between cascaded recognition processes, where some uncertainty exists about the results from each stage. Rather than requiring each stage of the analysis to decide completely which is the correct set of the results to be passed to the next stage, it is sometimes advantageous to pass along several sets of results, together with an indication of the confidence in each set.
Subsequent stages of analysis can then be applied to each set, possibly causing some to be discarded as unreasonable.

Some processes for interpreting handwritten text illustrate these points. In Figure 5, a handwritten phrase is subjected to five processes which attempt to extract the meaning of the phrase as between cap and gown. The process tree has as its root the phrase in its raw form. After subjecting the writing to a character recognition process, 16 nodes result, each denoting a different string of characters which could be extracted from the writing. These are indicated in Figure 5 by listing alternatives for characters or words vertically and enclosed in brackets: thus the 16 nodes result from a choice of 1. a space or no space in between. 2. an i or an e as the fifth character of between. 3. an a or o as the second character of cap and 4. ru pair or w in gown. Each of these nodes is subjected to a word recognition process, resulting in 16 nodes, one for each of the strings, but containing word groups rather than characters. Each of these nodes is subjected to word identification. At this stage, several branches of the tree end; namely, those containing words not in the (Webster's Collegiate) dictionary. Parsing to extract the syntactic structure will probably eliminate between as a possibility and post parse processing might be expected to eliminate cap on the basis of the idiomatic association of cap and gown.

For another twist on this theme, consider the problem of a one-eyed robot trying to find his way out of a room. The problem is for the robot to identify a door and then move to it. Following the plan of Minsky [3] a number of processes will be useful to the robot. The first of these is to take a picture of the room. With this raw material Process 2 might be to identify edges, 3 to identify potential objects, shapes on the walls, and perhaps shadows. Process 4 can discard some possibilities by considering relationships which usually exist between some types of objects. Each of these processes will probably produce a number of tentative results for the next process and so the scheme outlined above for phrase processing will be appropriate. The result of the first such analysis may still contain several possibilities since we have only taken one two dimensional picture. Assuming the room really has a door, the robot's
most reasonable action should be to pick one of the interpretations of the scene which contains the most door-like thing and move a bit toward it. He can then take a new picture, and not only analyze this picture but relate the finally selected objects to those he had in mind before moving. Some ambiguities may be removed, and some new ones discovered as new objects come in sight. A process tree for the robot's motion can be constructed so that each node of the tree represents a potential state of the room (one of the robot's guesses about his surroundings) and each branch represents one step the robot takes. As the creature moves about, some hypotheses are dropped for inconsistency (nodes with no branches emanating) and some nodes split (ambiguities arise from new objects sighted) but ultimately it would be expected that strong evidence will be collected in favor of one interpretation, which hopefully includes a door, and the robot has solved his problem.

The processing of this motion tree is multiple-tracking. Backtracking traversal of these process trees seems unnatural, although it is not out of the question; what is required is for processes at later stages to be willing to reject the input they get as not useful and return to earlier stages saying "try again". The disadvantage of this treatment is the loss of ability to compare the alternatives at a given stage of the process.

6. Parsing

Backtracking or multiple-tracking inevitably are required to implement parsers for context-free grammars (Chomsky [4]) which treat the sentence being analyzed in an orderly left to right sequence. This requirement arises from the ability of the grammar to describe sentences which are unambiguous as a whole, but which contain local ambiguities. These local ambiguities prevent assigning a unique structure to a part of a sentence without considering words which may be arbitrarily distant (Floyd [5], Irons [7]). Thus, in a recognizer which reads a sentence from left to right, it is not always possible to select one of several potential structures for a sentence without considering a word arbitrarily further to the right. A simple illustration of this phenomenon is given by the grammar
\[ S: = \quad dAb | dBc \]
\[ A: = \quad a \mid Aa \]
\[ B: = \quad a \mid aB \]

which designates strings of a \( d \) and then an arbitrary number of \( a \)'s followed by either \( a b \) or \( a c \). The structure covering the \( a \)'s can never be determined without knowing the last character, which can be an arbitrary distance to the right of any selected \( a \). The two different types of structures are:

```
    d a a a b
     \_\_\_\_\_\_\_\_\_
        A
          A
            A
              A
                A
                  S
```

and

```
    d a a a c
     \_\_\_\_\_\_\_\_\_
        B
          B
            B
              B
                S
```

Every parsing program which can ultimately assign the correct structure to such strings must be prepared to entertain several structures until the words determining which of them is correct occur.

A backtracking program does this by arbitrarily selecting one of several potentially correct structures and proceeding as though it were the only possibility. Each time such a choice is made, however, the alternatives which were not selected are recorded so that they may be tried later if necessary. If the end of the sentence is encountered, every arbitrary choice made has been correct and the parse is complete. If a point is reached where the next word cannot be fitted into the current structure, then either the
sentence is not grammatical, or an incorrect choice has been made at
an earlier stage of the process. In this case the program must return
to the last point where an arbitrary choice was made and try another
alternative.

A multiple-tracking program follows all alternatives in parallel
so that when several structures are possible with the next word in a
sentence, each of them is selected. The program carries along a number
of parses simultaneously by reserving separate areas of computer memory
for each. Every new word is tried in each parse structure being carried;
the word may fit in one of these in one way, several ways, or it may
not fit. If the word does not fit a structure, the structure is discarded.
If it fits in only one way, the structure is extended as indicated. If
it fits in several ways, then each of these ways gives rise to a separate
structure to be used with the following word.

The multiple-tracking program will discover ungrammatical sentences
when none of the structures being carried along can fit with the next
word. It will discover ambiguous sentences (if the grammar allows these)
by reaching the end of the sentence with more than one structure still
valid. The backtracking program will discover ungrammatical programs
when it has exhausted all untried alternatives and still cannot fit the
next word into its chosen structure. Ambiguous sentences will be assigned
only one structure (the first complete one discovered) unless the program
continues backtracking after it has reached the end of the sentence to
see if it can reach the end again in a different way.

The process tree for parsing is different from tree structures
which are sometimes used to show the parse structure. For this reason,
we have used bracketing diagrams to depict parses and trees to depict
process trees in the accompanying illustrations. In the process tree,
each node represents a partially completed parse. Branches emanating
from nodes lead to nodes representing parses derived from the parse of
the parent node by adding a new word from the sentence being treated.
Thus all the possible partial parses for the first n words are given by
the nodes at distance n from the root of the process tree. Figure 6
gives the process tree for parsing the sentence \( d \ a \ a \ b \) according to the small grammar above.

The process tree in Figure 6 appears to branch infinitely from the root. Actually, if we restrict ourselves to finite sentences (which we shall do), the tree is finite, but nonetheless the degree of branching is large enough to be troublesome. The branching is due to the left-recursive production

\[ A ::= Aa \]

which specifies an indefinite number of \( a \)'s and a left-gathering structure on them. Having seen only one \( a \), we cannot tell how many \( A \) brackets to use under it without knowing how many \( a \)'s follow. We must therefore have a separate node for each possibility, giving us nodes with \( 1,2,3,\ldots \) \( A \) brackets as shown. If we know how long the sentence is (or place an arbitrary upper limit on its length) we need only \( n \) such nodes if there are \( n \) words left in the sentence, for each \( A \) bracket will ultimately require an \( a \) word. As a practical matter, this limit really doesn't do much good, since \( n \) will almost always be so large that it is impractical to branch \( n \) ways whenever a left-recursive production is involved in the parsing.

The history of parsing algorithms for context-free grammars is rife with techniques for escaping or living with this indefinite branching problem. We shall discuss three of these techniques, and one of them in detail because they provide substantial examples of the advantages of backtracking and multiple-tracking in different situations. The three solutions are 1. backtracking in an ordered process tree, 2. combining all the nodes in the indefinitely branching group into one, and 3. altering the parse representation to eliminate indefinite branching. The first of these solutions takes advantage of a property of backtracking to avoid the problem, namely, that the whole process tree need not be traversed. This solution was used in early algorithms by [Brooker and Morris [11] and Cheatham [12] and is conceptually simple, but not particularly efficient in treating the branching problem. The multiple-
tracking analogue for these algorithms will make sense only if the
tree is made finite by limiting the length of strings, but even then
will not be very good, for the process tree will be very fat! The second
solution was the basis of another early algorithm by the author [8],
which has come to be called bottom-up or away from the symbols parsing.
In this algorithm, the brackets of the parse diagram are added on starting
with those closest to the string, and only the brackets whose final pre-
sence in the diagram is certain are kept. In this case, the nodes in
the indefinitely branching group all look the same and can therefore be
stored as one node (Figure 7). The disadvantage of this method is that
without the information about all the partially completed brackets it is
difficult to keep these programs from building up false structures which
ultimately are discarded. Both backtracking and multiple-tracking are
practical with this solution. The third group of solutions have been
used only more recently, and have typically been associated with multiple-
tracking algorithms rather than backtracking ones, although both are
appropriate. These algorithms by Irons [9], Kuno [11], Earley [14],
among others, use grammars from which the left-recursion has been removed
or turned into an explicit iteration so that indefinitely branching process
trees do not occur. The parse diagrams can be modified as suggested
in Figure 8 to collapse adjacent left gathering brackets into one bracket
with several names on it. This can be done in a unique way so that
given either bracket picture, the other can be constructed. The advantage
of the new notation is that the picture in each of the indefinitely
branching nodes is again the same, and so only one node need be kept.
This is illustrated in Figure 9 for comparison with Figures 6 and 7.
We can show that the only way indefinite branching can now occur is if the
grammar has a production like $A::=A$ and we readily rule out such
productions or groups of them ($A::=B$ together with $B::=A$) as do
most authors. Details of this argument can be found in the companion paper by Irons [10].
The programs we shall use to illustrate these ideas have been constructed to separate the control organization of the algorithm from their proper work. We present three different algorithms for parsing, each in a backtracking organization and a multiple-tracking organization. The three algorithms are a top-down algorithm, a bottom-up algorithm and an operator precedence algorithm, which taken together in their two guises embrace most parsing algorithms published. The first two algorithms work from syntax graphs which are derived in a simple mechanical way from the productions of a context-free grammar, but which express left-recursive productions as cycles in the graph to avoid the looping problem for the top-down scheme. The third algorithm, operator precedence, is that of Floyd [6] but generalized to a backtracking and multiple-tracking form to allow conflicts to be present in the precedence matrix. This generalization is not capable of use on all context-free grammars, and is not even efficient for all grammars it does work for, but is included to illustrate the remarkable similarity of form between three popular algorithms. This form is depicted in Figure 10. The center part of the diagram in this figure (inside the box) contains the parsing part of the program and is different for each of the three algorithms but the same for either a backtracking or multiple-tracking control. This control is in the boxes so marked, which are therefore different for different control organizations, but the same for the corresponding version of each algorithm. In addition, all three algorithms are organized into three sections, as indicated: one to open a new parse bracket, one to close a bracket, and one to extend an existing bracket over another symbol.

The syntax graphs used by the first two algorithms are derived from the syntax graph described and analyzed in the companion paper [10]. They are obtained from the productions of a grammar in the following way:

1. For each non-terminal in the grammar, write down its name, circled.
2. For each terminal symbol, write down its name.
3. For each production \( a ::= b \ldots s \) draw an arrow from symbol \( b \) to the symbol \( a \) and write down the rest of the production \( (c\ldots rs) \) along the line.
In addition it is sometimes useful to write the production number for the arrow by the head of the arrow. This process has been carried out for our small grammar in Figure 11. The left-recursive production results in a loop in the graph. This graph is used directly by the bottom-up parsing. Top-down parsing requires further modifications of the graph. The graph is first split into one graph for each non-terminal symbol in the following way: all arrows into the node for the non-terminal are followed backwards until a circled node is encountered or a dead end is reached. All these circled nodes and dead ends are then bunched together as the entry point for the graph for the non-terminal, and the rest of the graph is thrown away. If there is a loop, so that the arrow following could lead back to the starting node, the entire loop (including any circled nodes) is retained. This process returns the graph to a straight forward top-down syntax tree, with loops added to describe left-recursive productions. The result of these manipulations for our small grammar is shown in Figure 12. A more complete description of these graph processes and comparative analysis of the parsing programs is given in the author's paper [10]. The operator precedence algorithm is a straight forward generalization of that in Floyd [6]. The more popular algorithm by Wirth and Weber [15] clearly can be treated in the same way.

We shall not comment here on the relative merits of the three algorithms, since this analysis is presented in Irons [10]. We shall instead restrict ourselves to discussing the relative merits of backtracking and multiple-tracking embellishments of the algorithms. For this discussion some remarks about typical process trees are relevant. There is a different process tree for each string considered, but we might hope it would be substantially the same for different algorithms. Actually, we have seen from our earlier discussion that this is not really the case; the trees for different algorithms can be somewhat different as in Figures 6, 7 and 9. More substantial differences arise from the phenomenon occurring in many algorithms that a number of blind alleys are followed: that is, the algorithm extends a parse to the right when it is theoretically possible to determine from all the symbols to the left that the extension is futile. Unadorned top-down and bottom-up algorithms have suffered
considerable from this malady. (Our operator precedence extended algorithm does too!) We shall define a minimum process tree as one in the style of Figure 9, (with multiply labeled brackets) which has no excess foliage of this kind. The top-down multiple-tracking algorithm of Irons [9] uses such a minimum tree.

The shape of typical minimum process trees for strings in languages is particularly interesting because they determine a lower bound on processing time required to process sentences. The processing time is clearly related to the number of nodes in the tree. Algorithms which restrict the language to guarantee no branching (bounded context with small bound, operator precedence etc.) are fast because the process tree has the minimum number of nodes possible, namely, one for each word parsed.

We can, in fact, make a strong statement about minimum process trees for sentences produced by or designed to be used by people, which certainly includes every programming language, as well as natural languages. The statement is that the process trees will be sharply limited in width for sentences which are easily understood by a human reader. Fat process trees indicate a high degree of local ambiguity which confronts the reader in very much the same way it confronts a computer program. Psychological measurements showing us that people can follow only a very limited number of concepts simultaneously suggest that process trees for sentences oriented toward, or produced by, people will be narrow indeed and in fact allow a fixed bound to be confidently placed on such trees. The result of these observations is that we can expect process trees to be perhaps two or three nodes wide at one point, but usually returning frequently to a single node in width (thus resolving a local ambiguity) and never exceeding 6 or 7 nodes in width. It is interesting, however, that there is quite frequently some branching in process trees, as some ambiguity is everyday fare for the human being, in fact the basis of such literary devices as puns. These remarks account for the remarkable almost-but-not-quite success of single-tracking algorithms for parsing which, until recently at least, have offered significant economies of performance
over their context-free cousins. The "not quite" qualification arises because invariably languages constructed by people allow at least a few local ambiguities to occur, especially where the symbol set is restricted (as in many computer languages). To use the single-track algorithms then requires twisting the language around carefully to remove these potential local ambiguities.

The width of the process trees is also of interest in debating whether to use backtracking or multiple-tracking with a parsing algorithm. Fat trees speak in favor of backtracking because we may avoid a complete traversal of the tree, and process on the average only half the nodes at a branch point. Our observation that process trees for parsing are thin means that multiple-tracking will not suffer much from complete traversal.

7. **Storage Requirements**

The storage required for backtracking or multiple-tracking is to save the information in nodes which have been passed by, but remain to be processed. For backtracking, these nodes are all the nodes above the one being processed which have any branch to the right of the one which leads to the current node. For multiple-tracking, the nodes stored are all the nodes at the current distance from the root. The storage requirement for a small process tree are illustrated in Figure 13. It is easily seen that the storage requirement for multiple-tracking is proportional to the width of the tree, for backtracking to the height of the tree.

There are some choices for storing information at a node, too. We can store the whole state of the process at the node (most of the diagrams we have shown suggest this) or we can store the difference between a node and its parent. Which method is used depends on the amount of storage required to store the state of a process vs the amount to store incremental differences between states. The choice of backtracking or multiple-tracking also affects this decision as well as other considerations such as whether some nodes have to be stored anyway to be used as output of the process.
For the parsing algorithms of Section 6 the process trees are typically thin (3 or 4 nodes) and long (several hundred nodes), thus we would expect multiple-tracking to have a very substantial advantage in storage requirement. It happens, however, that the backtracking system can use its output to double as node storage, so that the advantage is not so substantial after all. We shall describe the detailed requirements for both systems.

The information which has to be obtained in order to proceed from a node is the following:

1. The input word at the node: the nth word of input-if the node is at distance n from the root (S,I).
2. The goal stack listing all the as yet unsatisfied goals (GOAL, G).
3. The current location in the syntax graph (J).
4. Where the output is to be stored (OUT, O).

S is usually a vector of words produced by a preprocessor, and so is perhaps several hundred words of 15 bit numbers (allowing a vocabulary of about 32000 words). The vector S is constant for the whole parsing process, so if we have to store it at all, we need store it only once, not once for each node. The goal stack GOALS is usually quite short (say 20 entries) and each value in it is a pointer to the syntax graph. The output OUT must contain all the information about bracketing. This can be done most compactly by giving an ordered list of the brackets drawn and usually will result in one 10 to 15 bit number per bracket. Since the number of brackets is a few times the number of input words, we could expect the order of 1000 of these numbers to be typical. Saving the state of the output list can be a significant problem, but fortunately has a reasonable solution for both backtrack and multiple-track parsing.

To summarize, the amount of information to be stored per node is:

1. I : One 15 bit input pointer.
2. G and GOALS : Twenty 15 bit graph pointers.
3. J: One 15 bit pointer
4. O and OUT : One thousand 15 bit output indicators.
The input vector S of five hundred 15 bit numbers may have to be stored once. It is clear that items 2 and 4 may be a significant problem if many nodes need be stored. We shall discuss the requirements for multiple-tracking and backtracking separately.

In the backtracking algorithm, the number of nodes of the process tree to be stored is large (roughly, the number of words in the input). However, the backtracking traversal is such that the output values need be stored only once, and only the pointer need be stored in the nodes. This is true because of the sequence of node processing; every node stored for processing is closer to root than the current node, so returning to one of these nodes means throwing away output, or reducing the output pointer. This leaves only I, O, J, G and GOALS to be stored at each node. We can make further economies by observing that GOALS changes at each step only by adding or taking off the last value, so only I, O, J, G and the change in GOALS need be kept. I and O also change by at most 1 and G by ±1 so we need only 4 bits to indicate these changes. The storage requirement is now reduced to the following:

1. The change in I (1 bit).
2. The value added or subtracted from GOALS (15 bits).
3. The change in G (2 bits).
5. The change in O (1 bit).

We must store S and OUT as well, but if S, OUT and the number of nodes stored are about equal, as we expect, we are required to store only 64 bits per word processed. If additional links are stored in the syntax graph to allow moving backwards along the arrows, it is possible to reconstruct the stored nodes entirely from the output pointers so that only the input and output need be stored. In the backtracking algorithms, the output cannot be used until the entire parsing process is over anyway, since backtracking may cause discarding previously generated output, so the storage required for the parsing is only that to hold the result, and the input.
For the multiple-track algorithms, the information required from a stored node is the same as for backtracking, but tailoring the storage scheme to multiple-track control leaves a quite different result than for the economized backtracking program. First of all, we need not store the input text $S$, because each component of $S$ is used to process all the nodes at its distance from the root, and then never used again. Output may be stored in common as for backtracking, but not quite so cheaply. As long as the input is ambiguous so that there is more than one node at a given distance from the root of the process tree, there is one stream of output for each of these nodes. Each stream reflects the output which will ultimately be used if its node is on a tree branch which leads to the end of the tree. The different outputs may conveniently be stored in a tree with each item of the output paired with a pointer to the item just before it. This tree will be shaped like the process tree. Each node of the process tree where brackets were closed will correspond to a group of successive nodes in the output tree which contain the output. Process tree nodes where no output was produced will be skipped in the output tree. The arrows in the output tree will be in the opposite direction to those in the process tree, and since there is only one arrow out of a node in this direction, storing the tree requires only one pointer and the output item to be stored for each node. The number of nodes in the output tree will be somewhat greater than the number of items in the final output, but since we expect the tree to be narrow, not very much greater. We can take advantage of another characteristic of the process tree for parsing, however, to eliminate the need to store all but a small part of the output. Each time there is only one node at a given distance from the process tree root, all ambiguity has been resolved temporarily and all the output accumulated for the single node left will occur in the final output. Consequently, it can be compacted into a single output vector or even passed on to the process which uses the output. As observed in Section 6, we expect this situation to occur frequently in the process tree, so we need be prepared to store only the output from a few nodes.
The remaining requirement is to store $G$, GOALS, $J$, and $O$ for each node. Since we expect only a few nodes to be stored, this is a modest requirement indeed.

8. **Conclusions**

Backtrack and multiple-track programming are alternate ways of organizing computer programs in which a branching of the analysis occurs. Which way should be used depends on characteristics of the particular program, but the following are a few guide lines.

1. **Processing time:** Backtracking will take about half the time of multiple-tracking if there is not a requirement to traverse the whole process tree. If information is available which allows choosing the correct branch with high probability, then backtracking can be significantly faster for fat trees, in fact, by up to $n$ times faster if the tree has average width $n$. On the other hand if complete traversal is required, for example to find all parses of an ambiguous sentence, processing time is the same for both methods.

2. **Storage requirements:** Backtracking is superior for short, fat trees, multiple-tracking for long, thin trees since the number of nodes stored is proportional to tree length for backtracking and width for multiple-tracking. However, other characteristics of the algorithm and the variables which characterize its state may substantially affect storage requirements as illustrated in Section 7, so this statement should be taken only as a rough guide.

3. **Infinite process trees:** Infinitely branching process trees can be processed with backtracking provided the branches can be suitably ordered and complete traversal is not required.

4. **Ambiguity:** A multiple-track program can identify and delimit genuinely ambiguous processes, whereas backtracking will select the first path to the end of the process tree and ignore others unless a complete traversal is carried out. Even with a complete traversal, however, delimiting the ambiguity can be done only by comparing output from different paths which reach the end of the tree. Ambiguity appears in the process tree as several branches of the process tree whose
adjacent (same distance from the root) nodes have identical contents except for information regarded as output. Since nodes which are adjacent in this sense are processed at the same time in multiple-tracking, the identical nodes are easily detected.

5. Error Recovery: Where errors exist in the data being processed so that the process tree terminates prematurely, multiple-tracking is superior in identifying the location and nature of the error. The reason for this is that the erring data forces the backtracking program to examine all the nodes which have been bypassed earlier before it is certain of an error. The position of the erring data can be determined as near the last input used, but more than this is usually difficult to determine. In the multiple-tracking program, however, an error is indicated when no nodes are produced from any of those at the current level of the process tree. In this situation, the position of the error is near the current position in the process tree and the nodes which failed to produce any offspring are all available for analysis.

6. Concurrent Processing of Output: Multiple-tracking can be used where it is desired to do further processing of the output from the multiple-tracking process before the complete tree has been traversed. Backtracking cannot be used for such concurrent processing because the output of a backtracking process is not known to be correct until the end of the process tree has been reached. In some cases the ability to generate the input, process it, and process the output of a multiple-track program concurrently can offer significant savings of storage. Also it allows one level of processing to affect another as, for example in parsing, changes in the grammar to result from parts of the sentence being parsed.

9. The Programs

Programs 1, 2 and 3 perform the tree traversals discussed in Section 3. Three arrays INFO, DOWN and RIGHT have been used to store the trees as binary trees. Node I is stored in INFO [I], DOWN [I], and RIGHT [I]
with information, DOWN [I] containing the downward pointer from the node, and
RIGHT [I] containing the rightward pointer. The pointers are integer
indices of the node pointed to in the three arrays. A zero pointer
value indicates a missing pointer. A small tree and its representation
in these arrays is given in Figure 14. The stack for Program 2 is
called STACK. Program 3 uses two vectors THIS and NEXT to hold nodes
yet to be processed. The N! nodes in THIS are the nodes to be processed
at the current distance from the root; NEXT holds the nodes to be
processed at the next distance (one greater) from the root. PROCESS
(INFO [L]) indicates the action taken at the node, whatever that may
be. Both programs are initialized to traverse the tree in Figure 14.

Programs 4, 5, 6, 7, 8 and 9 are the three parsing algorithms in two
guises discussed in Sections 6 and 7. The syntax graphs are stored
with three arrays LINK, POINT and TYPE, with several entries in each
used to store each node. All the entries used to store one node are
linked together with pointers in the LINK array. The first POINT
entry for a node contains the value of the node, i.e. what is stored
there, and the corresponding TYPE entry is a two bit value giving the
type of the node. The rest of the entries
use POINT to store the arrows emanating from the node. Figures 15
and 16 give the contents of these three vectors corresponding to the
graphs of Figure 12. The algorithms use vector GOALS as a stack of
goals for the parsing process and variable G to indicate the top
GOALS. OUT and its index O hold the parse output, S and its index I,
the input and J the current spot in the syntax graph. J points to the
pointer to the current node rather than to the node itself, so alternatives
to the current node can still be found. Thus the information in the
current node is POINT[POINT[J]] the first node following J is LINK
[POINT[J]] and the next alternative for the current node is LINK [J].
Storing the state of a node in these programs means copying all these
values onto the end of the indicated state stack, and popping the
state from a stack means restoring the values from the end of the stack
and shrinking the stack accordingly. Actually, considerably less than
the entire list of variables given above need be stored to retain a
state as indicated in Section 7, but we shall not give the details here of those economies.

Programs 8 and 9 use GOALS and G to hold the stack for the operator precedence parsing program. The parsing algorithm and matrices used are those of Floyd [6] except that conflicts are allowed in the matrix; that is, the relationship \( \triangleright \triangleleft \) and \( = \) may hold simultaneously between two symbols.
Fig. 1. Process Tree for the Space Scenario

Fig. 2. A Tree (left) and its Binary Tree Equivalent.
Fig. 3. Backtracking Traversal

Fig. 4. Multiple-tracking Traversal
Fig. 5. Recognition of Handwritten Phrases.
Fig. 6. Process Tree for Parsing the Phrase "daab."
Fig. 7. Process Tree for Bottom-up Parsing of daab.

Fig. 8. Abbreviated Notation for Left-gathering Brackets
Fig. 9. Process Tree for parsing daab using abbreviated bracket notation.

Fig. 10. Organization for the Parsing Programs.
1 \[ S::= d \ A \ b \]
2 \[ S::= d \ E \ c \]
3 \[ A::= a \]
4 \[ A::= A \ a \]
5 \[ B::= a \]
6 \[ B::= a \ B \]

Fig. 11. The Syntax Graph for a Grammar.

Bottom-up graph

Top-down Graph

Fig. 12. Syntax Graphs for Bottom-up and Top-down Parsing.
Node Processed

A
B
E
F
C
G
D
H
I

Nodes Saved

A
A B
A
A
A
A
A D

Backtracking

A
B
C
D
E
F
G
H
I

Multiple-tracking

B
B C
C D
C D
D

Fig. 13. Node Storage for Backtracking and Multiple-tracking.
Fig. 14. Tree Representation for Programs 1, 2 and 3.

<table>
<thead>
<tr>
<th>I</th>
<th>INFO [I]</th>
<th>DOWN [I]</th>
<th>RIGHT [I]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 15. Storage of the Bottom-up graph of Figure 12 for Programs 4 and 5.
<table>
<thead>
<tr>
<th>I</th>
<th>ENTRY [I]</th>
<th>I</th>
<th>LINK [I]</th>
<th>POINT [I]</th>
<th>TYPE [I]</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>d</td>
<td>Terminal</td>
</tr>
<tr>
<td>A</td>
<td>13</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>19</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>Non-terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>7</td>
<td>b</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0</td>
<td>S</td>
<td>Result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
<td>B</td>
<td>Non-terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>c</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>14</td>
<td>a</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>0</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>16</td>
<td>A</td>
<td>Result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>a</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
<td>0</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>19</td>
<td>20</td>
<td>a</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>21</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>21</td>
<td>0</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>22</td>
<td>0</td>
<td>B</td>
<td>Result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23</td>
<td>24</td>
<td>B</td>
<td>Non-terminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>0</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 16. Storage of the Top-down Graph of Figure 12 for Programs 6 and 7.
Fig. 17. Program 1: Backtracking Traversal with Recursive Subroutine.
Fig. 18. Program 2: Backtracking Traversal with Explicit Stack.
Fig. 19. Program 3. Multiple-track Traversal.
Fig. 20. Backtracking Framework for Programs 4, 6 and 8.

Fig. 21. Multiple-tracking Framework for Programs 5, 7 and 9.
Fig. 22. Core Algorithm for Top-down Parsing, Programs 6 and 7.
Fig. 23. Core Algorithm for Bottom-up Parsing, Programs 4 and 5.
Fig. 24. Core Algorithm for Operator Precedence Extended, Programs 6 & 7.
REFERENCES


