f91/ ~~See A2562 for MS~~

2562, 170

Programming Techniques

G. Manacher, S. Graham Editors

Backtrack Programming Techniques

James R. Bitner and Edward M. Reingold
University of Illinois at Urbana-Champaign

[RES]

The purpose of this paper is twofold. First, a brief exposition of the general backtrack technique and its history is given. Second, it is shown how the use of macros can considerably shorten the computation time in many cases. In particular, this technique has allowed the solution of two previously open combinatorial problems, the computation of new terms in a well-known series, and the substantial reduction in computation time for the solution to another combinatorial problem.

Key Words and Phrases: backtrack, depth-first search, exhaustive search, macros, combinatorial computing, non-attacking queen's problem, difference-preserving codes, pentominoes, tiling problems, squaring the square

CR Categories: 5.30

This work was supported in part by NSF grant GJ-41538, Authors' address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

Introduction

Using a computer to answer questions such as "how many ways are there to . . .," "list all possible . . .," or "is there a way to . . ." usually requires an exhaustive search of the set of all potential solutions. One general technique for organizing such searches is *backtrack*, which works by continually trying to extend a partial solution. At each stage of the search, if an extension of the current partial solution is not possible, we "backtrack" to a shorter partial solution and try again. This technique was first called backtrack by D.H. Lehmer in the 1950's, but has been discovered and rediscovered many times. One early description of such a method used to thread mazes is given in [16]. Currently, this method is used in a wide range of combinatorial problems including, for example, parsing [1], game playing [21], and optimization [14]. Other applications are discussed in [9] and [12].

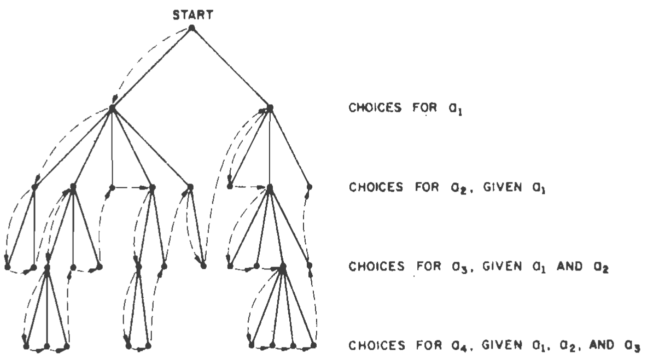
Walker [24] was the first to state backtrack in its full generality. Assume that the solution to a problem consists of a vector $(a_1, a_2, \ldots)$ of undetermined length. This vector satisfies certain constraints on the components, which makes it a solution. Each a_i is a member of a finite, linearly ordered set A_i. Thus the exhaustive search must consider the elements of $A_1 \times A_2 \times \ldots \times A_i$ for $i = 0, 1, 2, \ldots$ as potential solutions. Initially we start with the null vector Λ as our partial solution, and the constraints tell us which of the members of A_1 are candidates for a_1; we call this subset S_1. We choose the least element of S_1 as a_1, and now we have the partial solution (a_1). In general, the various constraints which describe solutions tell us which subset S_k of A_k comprises candidates for the extension of the partial solution $(a_1, a_2, \ldots, a_{k-1})$ to $(a_1, a_2, \ldots, a_{k-1}, a_k)$. If the partial solution $(a_1, a_2, \ldots, a_{k-1})$ admits no possibilities for a_k, then $S_k = \varnothing$, so we backtrack and make a new choice for a_{k-1}. If there are no new choices for a_{k-1} we backtrack still further and make a new choice for a_{k-2}, and so on.

The algorithm can be described formally as follows:

```
$S_1 \leftarrow A_1$
$k \leftarrow 1$
while $k > 0$ do
    while $S_k \neq \varnothing$ do
        $a_k \leftarrow$ smallest element in $S_k$
        $S_k \leftarrow S_k - \{a_k\}$
        if $(a_1, a_2, \ldots, a_k)$ is a solution then record it
        $k \leftarrow k + 1$
        compute $S_k$
    $k \leftarrow k - 1$
stop all solutions have been found
```

It is helpful to picture this process in terms of a tree traversal (see [11]). The subset of $A_1 \times A_2 \times \ldots \times A_i$, $i = 0, 1, 2, \ldots$ which is searched is represented as a tree as follows: The root of the tree (the 0th level) is the null vector. Its sons are the choices for a_1, and in general the nodes at the kth level are the choices for a_k, given the choices made for $a_1, a_2, \ldots, a_{k-1}$ as indicated by the ancestors of these nodes. In the tree shown in Figure 1 backtrack traverses the nodes of the

Fig. 1. The tree of partial solutions. The depth-first order in which backtrack explores it is shown by the dotted arrows.


START
CHOICES FOR a_1
CHOICES FOR a_2, GIVEN a_1
CHOICES FOR a_3, GIVEN a_1 AND a_2
CHOICES FOR a_4, GIVEN a_1, a_2, AND a_3

tree as shown by the dotted lines. Since the traversal goes as deep as possible in the tree before backing up to explore other parts of the tree, such a search is called *depth-first*.

Tarjan [22] has used depth-first search for various graph algorithms, giving a nice recursive procedure for such a search. In full generality this becomes

```
procedure backtrack(vector, $i$)
  if vector is a solution then record it
  compute $S_i$
  for $a \in S_i$ do backtrack(vector $\oplus$ $(a)$, $i + 1$)
  return
```

This procedure is invoked by the call backtrack(Λ, 1). $\oplus$ is used to denote concatenation of vectors.

Programming Techniques

Many applications of backtrack require relatively little storage and so it is not unreasonable to increase the storage requirements in order to decrease the running time of a program. This can be done in a language with macro expansion facilities (typically an assembly language) by using macros in such a way that some of the work is done once at assembly time instead of many times at run time. If a macro facility is not available, the same results can be achieved by hand coding, so these techniques may also be used in high level languages. If all solutions have length n, this can be done, for example, by eliminating part of the loop structure as follows: Write a macro called, say $CODE_i$, whose body consists of

```
       compute $S_i$
$L_i$:  if $S_i = \varnothing$ then goto $L_{i-1}$
       $a_i \leftarrow$ minimum element in $S_i$
       $S_i \leftarrow S_i - \{a_i\}$
```

This macro $CODE_i$ is expanded for $i = 1, 2, \ldots, n$ to produce the program

```
       $CODE_1$
       $CODE_2$
         ⋮
       $CODE_n$
       record $(a_1, \ldots, a_n)$ as a solution
       goto $L_n$
$L_0$:  stop—all solutions have been found
```

As we shall see later, macros can also be useful when the length of the solution is not known.

This technique results in several obvious savings. First, the step "compute S_i" may be very different for different i; we can use the macro to "customize" each block to compute S_i in the most efficient way. Second, the technique often facilitates the greater and easier use of registers since each block can have exclusive use of one or more registers. This can be accomplished during the expansion of the macro, but is in general difficult to accomplish in a nonmacro program. Finally, loop counters and end checks are unneeded and branching occurs only when backtrack is necessary.

Macros provide a method of increasing the speed of the program (that is, the rate at which we process nodes in the tree). Of even greater importance is to decrease the number of nodes in the search tree, and hence, also speed up the program. There are four general methods to accomplish this:

(1) *Preclusion*. This is a very general technique found in nearly all backtrack programs. In the generation of solutions, backtracking should occur as soon as it is discovered that the current partial solution cannot produce any solutions.

(2) *Branch merging*. When possible, do not search branches of the tree that are isomorphic to branches that have already been searched.

(3) *Search rearrangement*. In general, nodes of low degree should occur early in the search tree, and nodes of high degree should occur later. Since preclusion frequently occurs at a fixed depth, fewer nodes may need to be examined. When faced with the choice of several ways of extending the partial solution (i.e. which square to tile next or in which column to place the next queen; see below), we choose the one that offers the fewest alternatives.

(4) *Branch and bound*. This technique is used when we are searching for a solution of minimum "cost." Once a solution is found, all partial solutions with greater cost (assuming the costs are additive) can be discarded. In using this technique, it is beneficial to get a good solution early in the search, and so it is best to arrange the search so good solutions will be found early.

An example of the usefulness of the first two techniques is in the $n \times n$ nonattacking queens problem: In how many ways can n queens be placed on an $n \times n$ chess board so that no two queens are attacking each other? This problem was first proposed in about 1850 by Franz Nauck and has since been extensively investigated; a complete history of it and related problems can be found in [2]. The most thorough pre-computer results are in [13] and [20]. Walker [24] used backtrack on SWAC to find the number of solutions for $6 \leq n \leq 13$. Lin [15] found the number of solutions (although not the number of inequivalent solutions) for $n = 14$.

Since exactly one queen must be in each column, a solution can be represented as a vector $(a_1, a_2, \ldots, a_n)$

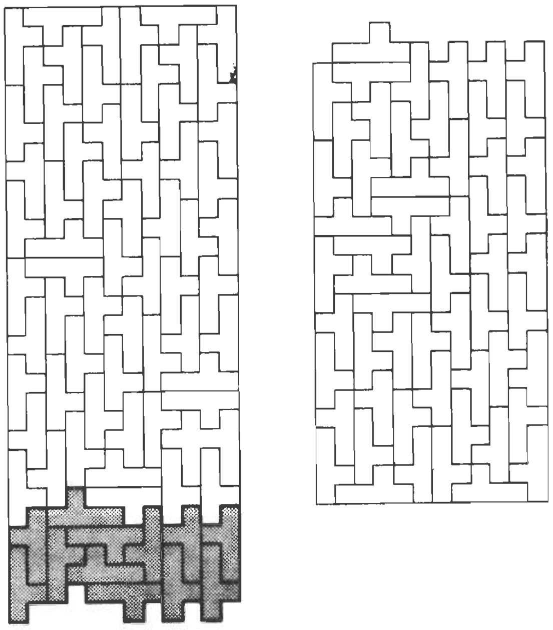
in which a_i represents the row of the queen in the ith column. This illustrates a use of preclusion. We do not consider all placements of queens, only ones with one queen in each column. Since all others are automatically illegal, they can be immediately discarded.

Two solutions are called equivalent if one can be transformed into the other by a series of 90° rotations and/or reflections. Clearly, if we find all inequivalent solutions, then we can easily produce the set of all solutions. Notice that a queen in any corner attacks the other corners and so there are no solutions with queens in more than one corner. Thus any solution with a queen in the (1, 1) square can be transformed by 90° rotations and/or a reflection into an equivalent solution in which the (1, 1) square is empty. Thus ~~we~~ know that we will still get all inequivalent solutions under the restriction $a_1 \geqq 2$. Moreover, if $a_1 > \lceil \frac{1}{2}n \rceil$ the solution can be reflected to obtain an equivalent solution in which $a_1 \leqq \lceil \frac{1}{2}n \rceil$. Since this does not interfere with the restriction that $a_1 \geqq 2$, we can restrict $2 \leqq a_1 \leqq \lceil \frac{1}{2}n \rceil$. When n is odd and $a_1 = \lceil \frac{1}{2}n \rceil$, we may restrict $a_2 \leqq \lceil \frac{1}{2}n \rceil - 2$ by the same principle.[1] These restrictions provide an example of branch merging. We can ignore solutions known to be isomorphic to solutions that will be generated.

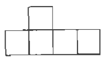
Including these tests in the general backtrack algorithm is expensive since they must be made each time through the inner loop, even though they will rarely be successful. Using the macro approach outlined above, however, the instructions for these tests would not be included in $CODE_i$ for $i > 2$ where they are not needed. Essentially, separate (although similar) programs are written for each level in the tree. On an IBM System/360 this allows us to do almost all of the work in the registers instead of in the (relatively) slower memory. Thus registers 0 through 14 can be used, respectively, to hold the positions of the queens in columns 1 through 15 in the 15×15 case. Using such a technique the 14×14 case was run in 5 min and the 15×15 case was run in 25 min. (An earlier attempt had taken five times as long on the same computer [5].) The 16×16 case required 168 min.[2,3]

There are other, less general, ways to use macros to speed up backtrack programs. For example, in tiling problems a macro can be written which produces a separate section of code for each position to be tiled or

Fig. 2. Solutions for the $5n \times 12$ problem for $n \geqq 10$. The 50×12 solution is formed by using the tiling on the left (without the shaded area) as the upper half of the solution and the tiling on the right as the bottom half. To produce a solution for $n > 10$, the shaded area is repeated $n - 10$ times.



for each tile. This technique also results in significant savings, since each section of code can now be tailored to the idiosyncracies of a specific piece. This approach was first used by Fletcher [7] for pentomino problems, and was subsequently used for soma-cube problems by Peterson [18]. The macro approach described for the queens problem can be viewed as $CODE_i$ placing the ith queen on the board.

Macros can be used in backtrack applications even if the length of a solution is unknown. For example, Fletcher's method was successfully extended to find the smallest n such that a $5n \times 12$ board can be tiled with the pentomino .

This problem was posed in [6] and the smallest known value was $n = 16$, due to D. Klarner. A backtrack with macros technique found tilings for $n = 10$ and $n = 11$, and demonstrated (by exhaustive search) that no tilings exist for $n < 10$. Furthermore, the solution for $n = 11$ gives rise to solutions for all $n \geqq 10$, as shown in Figure 2. Related results of this type can be found in [4].

The program considers each of the eight reflections and/or rotations of the pentomino as a separate piece and tries to cover the board with them. A clever method due to Fletcher [7] is used to determine which pieces can cover a given square. Each piece is put on the board, and the lowest numbered square it covers is called the *lead square*. Offsets from the lead square to the other four squares are then calculated. Each node in the tree has an offset as its value, and each path from the root to a terminal node corresponds to one piece. The offsets along this path tell us which squares the piece will

[1] In fact, after the programs had been run, A.I. Stocks pointed out that by reasoning as in the case $a_1 = 1$, the case $a_1 = \frac{1}{2}(n + 1)$, n odd, need not be considered. Thus we can always restrict $2 \leqq a_1 \leqq \lfloor \frac{1}{2}n \rfloor$.

[2] While peripheral to this paper, it should be noted that the following results were obtained:

n	number of solutions	number of inequivalent solutions
14	365596	45752
15	2279184	285053
16	14772512	1846955

[3] All of the computer runs described in this paper were done on the IBM System/360-75 at the University of Illinois at Urbana-Champaign.

170 2562

6. Chvatal, V., Klarner, D.A., and Knuth, D.E. Selected combinatorial research problems. Tech. Rep. STAN-CS-72-292, Computer Sci. Dep., Stanford U., 1972.
7. Fletcher, J.G. A program to solve the pentomino problem by the recursive use of macros.*Comm. ACM 8*, 10 (Oct. 1965), 621–623.
8. Gardner, M. Mathematical games. *Scientific American*, Sept. 1966 and Jan. 1967.
9. Golomb, S.W., and Baumert, L.D. Backtrack programming. *J. ACM 12*, 4 (Oct. 1965), 516–524.
10. Hall, M., and Knuth, D.E. Combinatorial analysis and computers. *Amer. Math. Mo. 72*, 2 (Part II) (Feb. 1965), 21–28.
11. Knuth, D.E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass., 1973.
12. Knuth, D.E. Estimating the efficiency of backtrack programs. *Math. Comp. 29* (1975), 121–136.
13. Kraitchik, M. *Mathematical Recreations*. W.W. Norton, New York, 1942; Revised ed., Dover, New York, 1953.
14. Lawler, E.L., and Wood, D.E. Branch-and-bound methods: a survery. *Oper. Res. 14* (1966), 699–719.
15. Lin, S. (personal communication).
16. Lucas, E., *Récréations Mathématiques*, 2nd ed. Gauthier-Villars, Paris, 1891.
17. Nievergelt, J. (personal communication).
18. Peterson, G. (personal communication).
19. Preparata, F., and Nievergelt, J. Difference-preserving codes *IEEE Trans. on Inf. Theory 20* (1974), 643–649.
20. Sainte-Laguë, M.A. *Les Reseaux* (*ou Graphes*). Memorial des Sciences Mathematiques, Fasc. 18, Gauthier-Villars, Paris, 1926.
21. Slagle, J.R. *Artificial Intelligence—The Heuristic Programming Approach*. McGraw-Hill, New York, 1971.
22. Tarjan, R.E. Depth first search and linear graph algorithms. *SIAM J. Comput. 1* (1972), 146–160.
23. Tutte, W.T. The quest of the perfect square. *Amer. Math. Mo. 72*, 2 (Part II) (Feb. 1965), 29–35.
24. Walker, R.J. An enumerative technique for a class of combinatorial problems. *Combinatorial Analysis* (*Proceedings of Symposium in Applied Mathematics, Vol. X*), Amer. Math. Soc., Providence, R.I., 1960.