Efficient search techniques—An empirical study of the *N*-Queens Problem

by Harold S. Stone
Janice M. Stone

This paper investigates the cost of finding the first solution to the *N*-Queens Problem using various backtrack search strategies. Among the empirical results obtained are the following: 1) To find the first solution to the *N*-Queens Problem using lexicographic backtracking requires a time that grows exponentially with increasing values of *N*. 2) For most even values of $N < 30$, search time can be reduced by a factor from 2 to 70 by searching lexicographically for a solution to the $N + 1$-Queens Problem. 3) By reordering the search so that the queen placed next is the queen with the fewest possible moves to make, it is possible to find solutions very quickly for all $N < 97$, improving running time by dozens of orders of magnitude over lexicographic backtrack search. To estimate the improvement, we present an algorithm that is a variant of algorithms of Knuth and Purdom for estimating the size of the unvisited portion of a tree from the statistics of the visited portion.

1. Introduction

In a recent paper, Stone and Sipala [1] showed a mathematical model of search that yields a surprising result. The average complexity of the search depends only linearly on the depth of the search tree, not exponentially, which is the complexity predicted by worst-case analysis. The key assumptions in that model are that search halts when it first reaches a fixed depth N, and that all internal search nodes behave identically and independently with regard to the probability that the search cuts off on either of the successor paths from each node.

Although the model appears to be highly constrained and unrealistic, small variations in the model do not affect the result. Hence, the probability of cutoff does not have to be identical at all nodes in order to achieve this behavior. Nevertheless, the results are limited to two types of search trees. Nicol [2] proved that the Stone-Sipala results hold either if the search tree is everywhere expanding, so that at every node the average number of live successors of a node is greater than one, or if the tree is everywhere contracting, so that the average number of live successors of a node is less than one.

Since the Stone-Sipala model predicts low complexity, and we know that there exist some searches that seem to take an exponentially long time to find the first solution, we seek a more general model that explains the exponential behavior observed in practice. One model that has been studied extensively is a probabilistic model for finding solutions to a Boolean predicate by discovering combinations of variables that set the predicate true. A practical instance of this problem is Roth's D-Algorithm [3]. Brown and Purdom [4] analyzed a lexicographic backtrack search and found that, although it has exponential average complexity, the exponent grows less than linearly in the number of variables, whereas the exponent is linear for exhaustive search.

Another strategy that proves effective for the Boolean-predicate problem is a backtrack search with a nonlexicographic ordering of solutions in which more tightly constrained variables are explored before less tightly constrained variables. Purdom and Brown [5] and Purdom [6] examine such a strategy, in which Boolean variables that are forced to a specific Boolean value are treated before variables that can take either Boolean value. Purdom and Brown [5] show that the effect of this strategy is equivalent to solving a simpler problem with one fewer variable. This changes the complexity by reducing the exponent of an exponent, which yields a dramatic improvement in running time. Purdom [6] found that search rearrangement can reduce average complexity to subexponential complexity in some circumstances in which lexicographical backtracking has exponential average complexity. Empirical studies of this algorithm and a multilevel variant appear in Brown and Purdom [7] and Purdom, Brown, and Robertson [8], respectively. The empirical studies show that one-level rearrangement yields great improvements over standard lexicographic search (as predicted by the search models) and that two-level rearranging gives a small improvement over one-level rearranging.

The results cited there are intriguing, but the question remains for that model, as it does for the Stone-Sipala model, whether the model accurately reflects any real-world problem. Moreover, the rearrangements considered in the literature are somewhat artificial, because the one-level algorithm simply considers the variables whose choices become forced ahead of variables for which a binary choice exists. Clearly, for a one-level rearrangement, the only sensible policy is to treat forced variables before treating variables for which a binary choice exists. What is more interesting is a situation in which several choices, in general, are available for each variable. It is not clear that the two-level policy studied in the papers cited is the most effective means for generalizing the one-level policy, and it is different from the "most-constrained" policy explored in this paper. Moreover, the analytical methods in the literature do not carry over to specific problems, because basic assumptions such as identical and independent probabilities do not generally hold for real problems.

This paper explores the average cost of search for a model problem that is currently a popular testbed for investigating the search complexity of Artificial Intelligence techniques. The problem is known as the *N-Queens Problem*, in which the objective is to place N queens on a chessboard so that no two queens attack each other. Although it is possible to construct solutions for some values of N without conducting a search, we limit the algorithms used in this paper to backtrack search in order to use the problem as a model for other problems in which techniques for directly constructing a solution without searching are unknown.

We have several findings of interest. First, a probability analysis does indeed show that the search tree expands for approximately $N/2$ levels, then contracts. From the Stone-Sipala model, we find that the lexicographic search algorithm is drawn rapidly down to the $N/2$ level, but below this level the probability of failure climbs quite rapidly. Consequently, a lexicographic search may explore a substantial portion of the central part of the search tree, which grows exponentially in the depth of the tree. Our experiments confirm that lexicographic search appears to take a time that grows exponentially before producing the first solution.

Because of statistical variations that depend on N, we show empirically that solutions for some values of N are harder to find than for larger values of N. Specifically, for N even and less than 30, it is generally preferable to solve a problem of size $N + 1$ and to throw away a queen to get a solution of size N, rather than to solve a problem of size N. We call this an "odd" solution to the N-Queens Problem.

The paper demonstrates the effectiveness of search rearrangement under the rule "for the next placement, place a queen in the row that has the fewest placement choices." This rule enabled us to solve quickly all N-Queens Problems up to $N = 96$ using only a personal computer. The lexicographic strategy failed to produce a solution in reasonable time for $N = 30$. Bitner and Reingold [9] suggested that this rule be used to solve the N-Queens problem, but they proposed the rule with a few others without attempting to give a relative evaluation of the rules. We show here that an early cutoff rule proposed by Bitner and Reingold yields only a small improvement, whereas the fewest-choices policy appears to yield literally dozens of orders of magnitude improvement for large problems.

To help estimate the size of these enormous search trees, we modified an algorithm originally due to Knuth [10] and improved by Purdom [11] for estimating the size of backtrack trees. They rely on statistical sampling of many paths in a backtrack tree. Rather than sample at random, we propose an algorithm that produces an estimate of total tree size at any point during a search. The algorithm keeps track of average statistics at each level of the tree as it scans nodes of the search tree. It estimates total tree size by assuming that the unscanned portion of the search tree has the same statistics as the portion already visited. The advantage of this algorithm over the Knuth-Purdom techniques is that cost estimates of the remainder of a search are available to our algorithm without requiring any special searches to be conducted. But the estimates may not have the same accuracy as the Knuth-Purdom estimates. Our algorithm may produce biased estimates, because the nodes visited lie in a limited region of the search tree rather than being distributed throughout it.

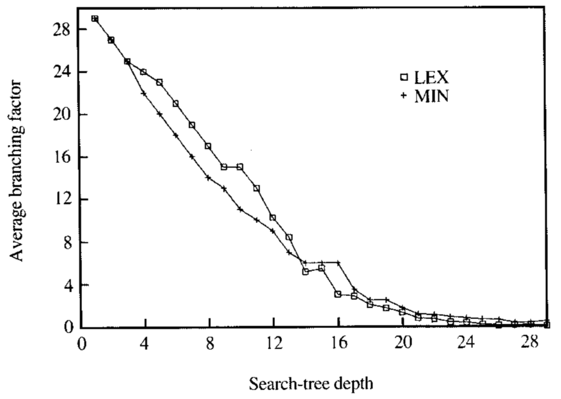


Figure 1

Average branching factor as a function of depth for 29-Queens Problem for lexicographic (LEX) and most-constrained (MIN) terse searches.

Section 2 of the paper presents the exponential complexity of the lexicographical search for one solution and compares the results to the results predicted by Stone and Sipala. In Section 3 we present the "odd" solution and the search-rearrangement method for discovering solutions. The algorithm for estimating the size of the search and the number of solutions appears in Section 4, together with an analysis of the data collected for the *N*-Queens Problem. The final section summarizes the results of this research.

2. Lexicographic search

This section presents basic empirical data for lexicographic searches that solve the *N*-Queens Problem. We also show that the results are consistent with the Stone-Sipala results, even though the average complexity discovered grows exponentially with the tree depth.

Nicol [2] defines a *terse* search to be one that terminates when the first solution is discovered. A *full* search is one that produces all solutions. Most of the results in this paper relate to terse search.

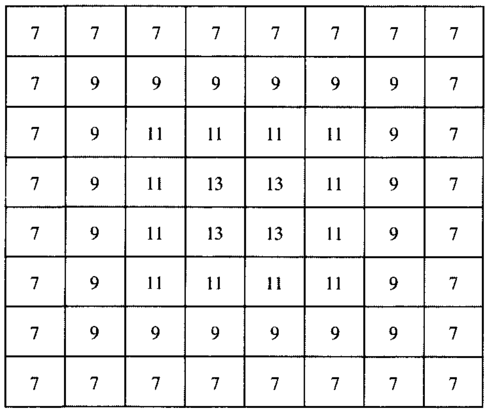
A *lexicographic* search is one that produces solutions ordered lexicographically by some natural sorting key. For the *N*-Queens Problem, the natural way to represent solutions is by an *N*-tuple whose *i*th component is the column number of the queen in Row *i*. For four queens, the solutions are (2, 4, 1, 3) and (3, 1, 4, 2). The natural sorting key for solutions to the *N*-Queens Problem is to sort *N*-tuples in ascending order so that (2, 4, 1, 3) comes before (3, 1, 4, 2) because it is lexicographically less than (3, 1, 4, 2).

Stone and Sipala [1] showed that under certain conditions a terse search has an average complexity that depends only linearly on the depth of the search tree. Those conditions include uniform probability of cutoff on the interior of the tree, but the results presented were not sensitive to small variations in cutoff probability. However, they do depend strongly on whether the tree is expanding or contracting. A search tree is said to be *expanding* at a given node if the expected number of successors of that node exceeds unity. The tree is *contracting* at a node if the expected number of successors is less than unity. Nicol [2] proves that if a tree is everywhere expanding or everywhere contracting, then the average terse-search complexity depends only linearly on the depth of the tree. (The search complexity depends as well on the work per node visited, so that total complexity may be much greater than a linear function of the tree depth.)

Given that expanding trees and contracting trees apparently lead to very efficient terse search, what trees remain that can lead to very lengthy terse search? If a tree first contracts, then expands, the contracting portion of the tree (if sufficiently large) determines the search complexity and the terse search is efficient. If the contracting portion is very small, then the expanding portion of the tree determines the search complexity, and again terse search is very efficient. So trees that contract, then expand, are relatively efficient for terse search.

The interesting case is a tree that first expands, then contracts. In the expansion portion of the tree, the tree grows exponentially. At some critical depth in the tree, the tree reaches its maximum breadth, then contracts exponentially at depths below the critical depth. The Stone-Sipala analysis predicts that a terse search will quickly reach the critical depth, but because most searches that reach the critical depth fail, a terse search may well have to visit a large number of nodes at the critical depth before it discovers one that leads to a solution. In fact, the number of nodes explored can grow exponentially in the depth of the search tree. An expansion-contraction tree is clearly a tree that can be exponentially difficult to search.

The *N*-Queens Problem produces an expansion-contraction tree when it is solved by a terse lexicographic search. The reason that this is true is illustrated in **Figure 1**, which shows the average branching factor in the search tree as a function of search depth for the 29-Queens Problem. The branching factor is given both for lexicographic search (LEX) and for most-constrained search (MIN), which is discussed later. Lexicographic search terminated after three million backtracks, having found its first solution. The average branching factor at each level of the tree is computed by counting each node at that level equally. Only a minuscule fraction of the search tree had actually been visited at the time the search terminated, so the data shown may differ somewhat from the true average branching factor for the whole tree.



7	7	7	7	7	7	7	7
7	9	9	9	9	9	9	7
7	9	11	11	11	11	9	7
7	9	11	13	13	11	9	7
7	9	11	13	13	11	9	7
7	9	11	11	11	11	9	7
7	9	9	9	9	9	9	7
7	7	7	7	7	7	7	7

Figure 2

Number of squares attacked on diagonals from each square on a chessboard.

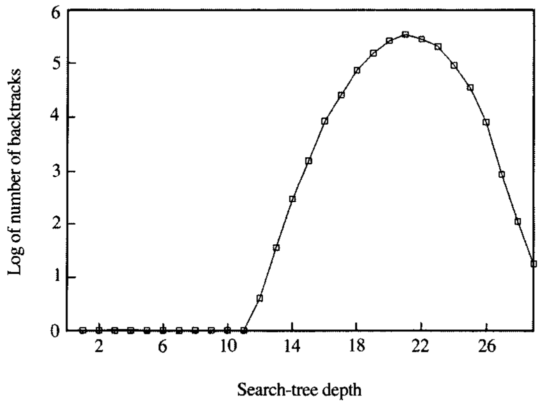


Figure 3

Number of backtracks at each search level for 29-Queens Problem with a terse lexicographic search.

Note that at depth 1 the branching factor is 27 because the first queen is placed in a corner and attacks two squares in the second row. This leaves 27 choices for the placement of the second queen. The figure shows that the average branching factor diminishes by roughly two as each new queen is placed, until roughly half of the queens are placed. So each new queen placed attacks roughly two otherwise unattacked squares of the next queen to be placed, and this holds until about half of the queens are placed.

To see in general why this should be the case, consider how one queen attacks other squares on the board. Obviously, a queen attacks every square in its own row, but since we never attempt to place two queens in one row, we can ignore these cells. A queen attacks as few as one and as many as three cells in any other row. The minimum number attacked is one, and this is due to an attack along a column. The variable number of cells attacked is due strictly to attacks along diagonals. **Figure 2** shows the number of cells attacked along diagonals for each possible placement of a queen on an 8-by-8 board. Note that, in general, a queen in a corner attacks $N - 1$ cells along a diagonal. If you move the queen in the corner along the periphery of the board, it continues to attack $N - 1$ cells, because as the queen moves from cell to cell, the length of one diagonal shortens and the other diagonal lengthens by the same amount. Hence, in Figure 2, we show that each cell on the periphery attacks seven cells along diagonals.

If we move inward toward the center of the board, we see immediately that as we move from "ring" to "ring" the number of cells attacked along diagonals increases by two. For example, a queen in the corner can be moved one step along a diagonal toward the board center. As it moves one step, it continues to attack $N - 1$ cells on the main diagonal, but adds an additional two cells on the minor diagonal. Thus, the board may be viewed as composed of concentric rings as shown in Figure 2, with each ring closer to the center attacking two more cells than its nearest outer neighboring ring.

Since a queen attacks from one to three cells in any other row, the probability that a queen attacks a specific one of the N cells in a row falls somewhere between $1/N$ and $3/N$. If we examine row $N/2$ and ask how many cells will be left as possible choices when we place queen $N/2$, we can obtain an approximate answer by assuming that on the average $2/N$ cells are deleted as each new queen is placed. The actual number of cells deleted per step is not uniform, because there is a higher probability of attacking new cells when the row has relatively few cells attacked than when most cells in the row are attacked and relatively few free cells remain. In fact, Figure 1 shows that the average branching factor does not fall to 0 at half of the depth, but that the slope becomes less steep than -2 somewhat earlier.

The point at which the average branching factor falls below unity is the point at which the tree changes from expanding to contracting. It is at this level that most of the work in searching is performed. In Figure 1, that point occurs at a depth of 21. **Figure 3**, which shows the number of backtracks performed at each level, confirms this finding. Nevertheless, to estimate the size of the backtrack tree, let us

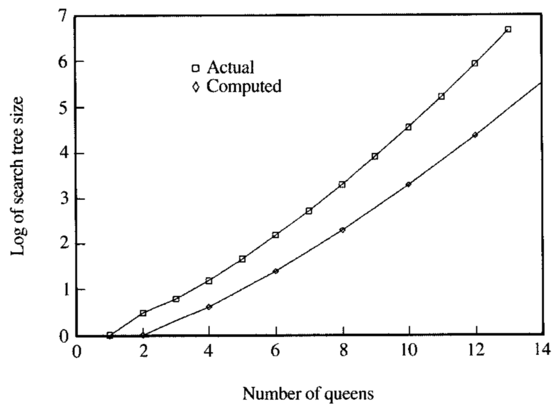


Figure 4

Actual and computed search tree size for full lexicographic search for N up to 13.

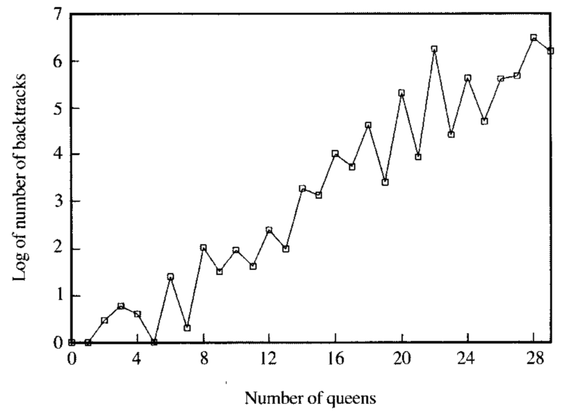


Figure 5

Number of backtracks for terse lexicographic search for N up to 29.

assume that the branching factor diminishes by two for each of $N/2$ levels. Then the tree breadth at level $N/2$ is given by

$$\text{breadth } (N/2) \cong \prod_{i=1}^{N/2} (N - 2i),$$

which can be reduced to

$$\text{breadth } (N/2) \cong 2^{N/2} \left(\frac{N}{2}\right)!. \qquad (1)$$

Figure 4 compares Equation (1) to the number of nodes visited during a full lexicographic search for N up to 13, and we see that Equation (1) behaves something like the complexity of full lexicographic search, but lies strictly below that curve. Note that Equation (1) counts only the nodes at the widest part of the tree, and fails to count the nodes above or below this level.

How good is a terse lexicographic search? The statistical analysis of terse lexicographic search for the N-Queens Problem is rather complicated, and we have not been able to produce an analysis that yields accurate predictions. The number of backtracks for terse lexicographic search grows exponentially in N, as shown in **Figure 5** for N up to 29. This graph shows the number of backtracks to obtain the first solution. Although the data curve is rather jagged, the trend is exponential because the points lie along a linear slope on a logarithmic scale. The curve stops at $N = 29$, because we were not able to obtain solutions beyond the 29th in a reasonable time on the IBM PC/AT computer on which the computations were performed.

As we pointed out above, terse search in this case is exponential because the search is dragged down quickly to the widest portion of the search tree from where successful paths are very rare and difficult to discover. The number of ways of placing $N/2$ nonattacking queens on an N-by-N board grows much faster than exponentially in $N/2$, as indicated in Equation (1). These partial solutions form the search space. But the full solutions are very sparse in the search space, so even with the efficiency of backtracking, a terse search appears to take an amount of time that grows exponentially in N.

3. Two speedup techniques

This section treats an interesting technique that solves a problem of size $2N$ by attacking a larger problem $2N + 1$ that produces the solution more quickly. The section closes with an empirical study of an extremely powerful technique based on search rearrangement that leads to speed improvements measured in dozens of orders of magnitude over lexicographical search.

• *An odd solution to the* N-*Queens Problem*

Consider the data shown in Figure 5, and notice the jagged nature of the curves through $N = 29$. Finding the first lexicographic solution turns out to be much easier for odd N than for even N. The speed increase is obtained because a solution for size $2N + 1$ requires anywhere from 2 to 70 times fewer backtracks than a solution for size $2N$ for $2N < 30$ except for $2N = 26$. If there is a solution with a queen in the corner, then the first solution found by lexicographic search has a queen in the corner, so we can take any such solution for size $2N + 1$ and turn it into a solution for size $2N$ by removing the queen in the corner, and the row and column occupied by that queen. The solution technique is rather odd because it proposes to solve a problem faster by solving a bigger problem.

What is happening is that the solution for a board of size $2N + 1$ contains a solution for a board of size $2N$ that has no queen on the major diagonal. Consequently, we can add to the $2N$ solution a bordering row and column with a queen placed at the intersection, and the new solution is correct for $2N + 1$ queens. The data indicate that it appears to be much easier to discover solutions that have no queens on the major diagonal when N is even than when N is odd. Hence, we are speeding up the computation by adding a special constraint to focus the search to a particular region of the search space. We are not actually increasing the size of the problem being solved. When we attempt to solve the $2N$-Queens Problem directly, we seek a solution with a queen in the corner. When we solve the $2N$-Queens Problem by solving the $2N + 1$-Queens Problem, what we are really doing is seeking a solution to the $2N$-Queens Problem in which no queen lies on the major diagonal. For N even, solutions are apparently easier to find when queens are forbidden to lie on a diagonal than when the solution must have a queen in the corner.

Although we cannot explain this particular characteristic, clearly the solution density is not uniform throughout the search space. We must expect that there exist search regions where solution density is many times above or below average solution density. **Figure 6** shows the solution density for a full lexicographic search for the 12-Queens Problem. The plot shows solutions per backtrack at each solution, normalized to a rate of 1.0, which is the overall average solution rate. The vertical divisions show the point at which the queen in Row 1 moves to a new column. Note that the solution rate is a few times higher than average for a queen placed near the center of Row 1 and a few times less than average for a queen placed at either end of Row 1. Also note that the solution density tends to be highest in midsearch for any given placement in Row 1, which corresponds to a placement of the second queen near the center of the board.

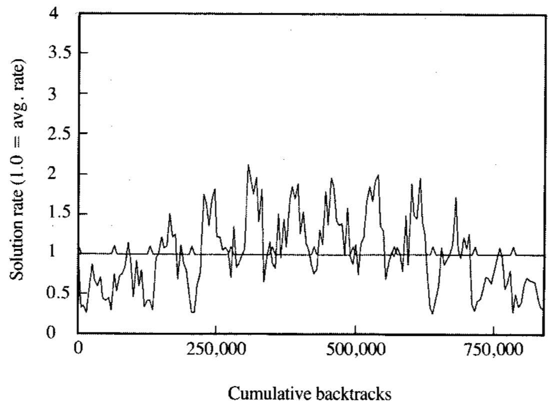


Figure 6

Solution rate for full lexicographic search for 12-Queens Problem.

• *Most-constrained search*

Bitner and Reingold [9] proposed a search technique that advances the search along the *most constrained* path. That is, at any point in the search when seeking to place a new queen, place the queen that has the fewest possible choices of moves. (Break ties arbitrarily.) They proposed other techniques as well, such as sharpening the cutoff criteria, and branch-and-bound when a bound exists. It so happens that most-constrained search alone solves terse search for the N-Queens Problem for N up to 96. While all of their suggestions generally produce reductions in computation time, the results obtained from most-constrained search are spectacular compared with their other suggestions. For example, it is relatively easy to initiate a backtrack when any remaining row has no choices for a placement. When this rule is used and compared to the rule that forces a backtrack when only the next row has no choices, we discover that the search does indeed visit fewer nodes in the search tree. But the reduction in the search effort is a small factor, and in part is balanced by a small increase in computation time due to the extra testing. The sharper cutoff criterion is worthwhile but does not alter the computation time dramatically. This observation is consistent with the Stone-Sipala model, which shows that small changes in cutoff probability have a small effect on effort expended.

But the most-constrained rule leads to enormous reductions in complexity. **Figure 7** compares the average number of backtracks for LEX (lexicographic) and MIN (most-constrained) search. The figure compares the average number of backtracks per solution for one and ten solutions. Note that the logarithmic scale tends to deflate the differences. But careful inspection shows that MIN is three orders of magnitude more efficient than LEX for N in the high 20s. **Figure 8** shows MIN plotted for N up to 96. This evidence shows a rather horizontal trend for the bulk of the points, although the upper envelope might be viewed as following an exponential rise. However, at $N = 97$, MIN did not terminate after a reasonable running time, and we suspect that $N = 97$ may be a fairly difficult case to solve. If so, this may be a point where exponential behavior of MIN begins to be observable. The running time for most points on the curve was a few seconds of time on the IBM PC/AT.

Figures 7 and 8 together suggest that LEX running time increases exponentially and MIN running time increases only polynomially over the range of N studied. The relative improvement of MIN over LEX is virtually impossible to

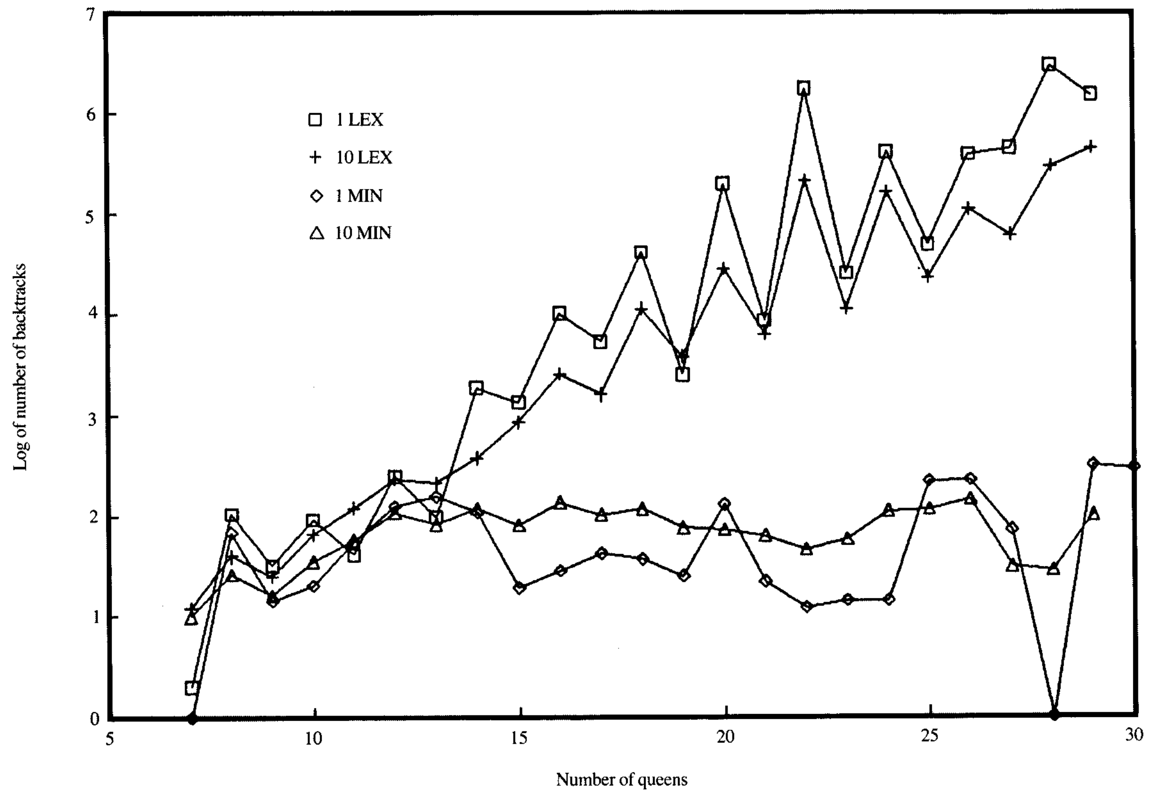


Figure 7

Average number of backtracks per solution for one and ten solutions for both lexicographic and most-constrained searches.

measure, because the running time of LEX becomes years and centuries as N grows, even though the search is terse and LEX can stop as soon as it finds one solution. Obviously, we cannot run LEX and compare its running time with MIN. A reasonable means for comparison is discussed in the next section.

4. Estimating the search size

This section presents an algorithm that produces an estimate of total search size and number of solutions during the running of a terse search. If the search were extended to a full search, the estimates would converge to the correct values. The algorithm is a derivative of algorithms proposed by Knuth [10] and Purdom [11], who propose random sampling techniques to make such estimates.

Since we propose to run a terse search, it becomes an interesting challenge to produce estimates on the fly to discover how much searching remains to be done and how many solutions remain to be found. Knuth and Purdom produced estimates for the size of the full search tree by random sampling and by assuming the full tree to have characteristics similar to the characteristics of the portions of the tree actually visited. Our algorithm is similar in that it observes some portion of the search tree and projects the observed characteristics across the portion of the tree as yet unvisited. The means for projecting the estimates across the whole tree is the same in the new algorithm as in Purdom's algorithm.

The new algorithm requires that code be added to a backtrack algorithm in the following places:

1. At initialization.
2. When visiting a node for the first time.
3. When returning to a node from a backtrack.
4. When preparing to visit the jth choice for a node.

Two quantities are estimated—the breadth at level i, and the size of the subtree from level i to level N. For these estimates

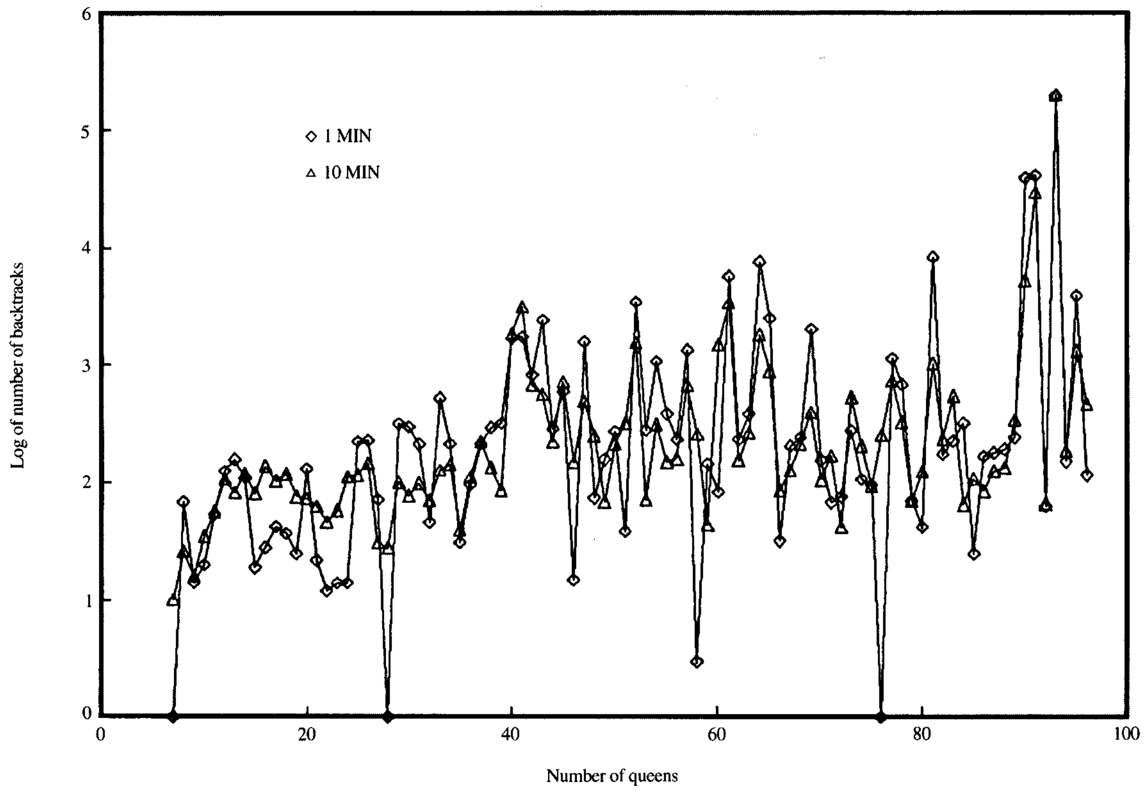


Figure 8

Average number of backtracks per solution using most-constrained search for one and ten solutions.

we use N-vectors *breadth* and *treesize*. For initialization, we set *breadth*[1] and *treesize*[1] to 1, and set the remaining elements of the two vectors to 0. This is the correct state for a search tree that has a single node.

When a new node at level i in the search tree is entered, the estimates are updated with the following statements:

```
breadth[i]: = breadth[i − 1] × choices
treesize[i]: = breadth[i];
```

The first statement increases the breadth by the number of choices at this node. This follows from the algorithms of Knuth and Purdom. The second statement estimates the tree size from this node downward to a leaf node to be equal to the breadth at that node.

When returning to a node from a backtrack, the size of the subtree below that node is accumulated into the estimated tree size at that node. The code uses the statement

```
treesize[i]: = treesize[i] + treesize[i + 1];
```

The last place that requires additional code is the place where a node is preparing to make its jth choice, for $j > 1$. To give proper weight to choice j, the exploration of the new choice must be given weight $1/j$ and each of the $j - 1$ choices already explored must be given the same weight. However, the choices already explored have each to be given a weight of $1/(j - 1)$ in anticipation of terminating the search after visiting just those $j - 1$ choices rather than continuing to the next choice. So we must reduce previous estimates by a factor of $(j - 1)/j$. The code to do so is

```
factor: = (j − 1)/j;
breadth[i]: = breadth[i] × factor;
treesize[i]: = treesize[i] × factor + breadth[i];
```

By reducing *breadth* by the given factor, the estimates produced for choice j are weighted by $1/j$ of the estimates that would be made if only a single choice were available. The last statement makes a similar reduction on the accumulated tree-size estimate and adds in the current

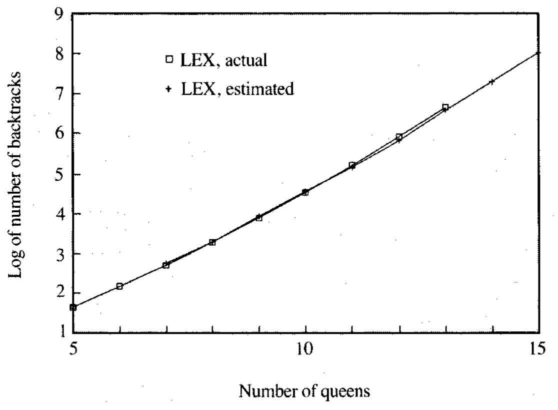


Figure 9

Actual and estimated tree sizes for full lexicographic searches of up to 13 queens.

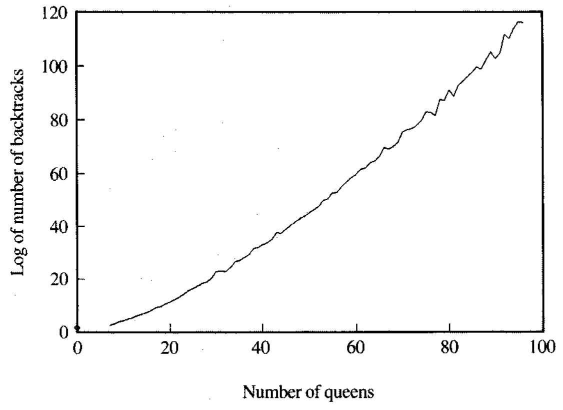


Figure 10

Estimated search-tree size for full search after ten most-constrained solutions.

breadth value, which is the tree size of choice *j* before it is explored. The net effect of the statements is to average the subtrees for *j* choices by giving each subtree equal weight in the total estimate. The reweighting step is done for $j > 1$ because the case for $j = 1$ is treated above when a node is first entered.

The search can be stopped at any point, and the estimated tree size is the sum of *treesize*[*i*] for *i* running from 1 to the tree level at which the search terminated.

To estimate the number of solutions, we use the same technique, with a small modification. Instead of setting the initial estimate of *treesize*[*i*] when a node is first entered or adding to it the value of *breadth* when a new choice is explored, *treesize*[*i*] is set to *breadth*[*i*] at the point a solution is discovered. This is equivalent to weighting solution nodes by a factor of unity and nonsolution nodes by a factor of zero when estimating the total size of the search tree.

Figure 9 compares the estimated and actual sizes of the trees for full lexicographic searches of up to 13 queens. The estimates were produced by running terse lexicographic searches for one solution. The agreement is excellent. **Table 1** gives detailed statistics on actual tree sizes and estimated tree sizes produced from terse LEX and MIN searches. Note the close agreement between the estimated tree sizes and the actual tree sizes, even though only a very small region of the search tree is actually explored. As *N* grows large, we do not expect the agreement to be as good because the variance in search statistics across the search tree is potentially very large. Figure 6 suggests that the solution density is greater when the first queen is placed near the center of the first row than when the first queen is placed at a corner. For 12 queens the difference in density is a small but significant factor. For 90 queens, the difference may be huge. But the search estimate for a terse MIN search is based only on the statistics for a queen placed in the corner, which may be rather uncharacteristic of the full tree.

Nevertheless, the estimated full size of the search tree appears in **Figure 10**. The graph is logarithmic so that a straight line represents an exponential growth. The curve

Table 1 Actual and estimated sizes of search trees.

N	*Estimate 1 MIN*	*Estimate 10 MIN*	*Actual MIN*	*Estimate 1 LEX*	*Estimate 10 LEX*	*Actual LEX*
7	568	457	416	1,090	569	552
8	1,210	1,300	1,415	1,870	1,980	2,057
9	4,250	6,540	5,610	8,030	8,790	8,394
10	23,100	20,900	20,863	32,500	37,100	35,539
11	63,200	72,400	89,670	152,000	150,000	166,926
12	243,000	304,000	432,103	715,000	669,000	853,189
13	1,030,000	1,510,000	2,230,980	3,300,000	3,880,000	4,674,890

shown grows slightly faster than exponentially in N. The estimated size for $N = 96$ is 10^{116} nodes, yet a MIN search produced a solution in less than five seconds. The most costly MIN search took 1100 seconds to find a solution for $N = 93$, whose estimated search tree has 10^{113} nodes.

Why is MIN so efficient? There are at least two different possible reasons, both of which definitely contribute to its efficiency. The first reason is that MIN tends to produce a narrower backtrack tree than LEX because MIN tends to have a narrower branching factor in the first several levels of the search tree. This holds in Figure 1, but the difference in the average is not dramatic. Since average branching factor impacts the estimated tree size, we can compare the effects of branching factor reduction by comparing the estimated tree sizes produced by MIN and LEX searches. For $N = 29$, the estimates are $1.15 \cdot 10^{20}$ and $2.87 \cdot 10^{20}$ for LEX and MIN searches, respectively, which are not terribly different.

The more dramatic effect of MIN over LEX is that it produces a much higher success rate in the contracting portion of the search tree than does LEX, because MIN quickly discards paths that do not lead to solutions. How does this happen? Consider, for example, the difference in two strategies. Suppose that Row i has nine choices and Row j has three choices left. LEX places a queen in Row i, then places several more queens, then places a queen in Row j. MIN places the queen in Row j, then several more queens, then the queen in Row i. In this case, since there are only three choices for Row j, we have to seek some combination of choices for Row i and intervening choices that satisfy Row j. If, for example, some placement in Row i is incompatible with any placement in Row j, LEX will not discover this fact until it has explored a potentially large search tree. On the other hand, MIN will not explore that tree at all because, by choosing Row j first, it does not generate choices incompatible with Row j. Since Row j has fewer choices, it is the more difficult constraint to satisfy. Satisfying the most difficult constraints first reduces the effort expended on rejecting paths that do not lead to solutions.

We conjecture that the power of MIN is felt mostly at levels near the middle of the search tree. At early levels, selection is not critical because most arrangements of the first few queens lead to solutions. In the middle levels, many queen placements lead to eventual search failure, but potentially large searches are required to discover this fact. At these levels, MIN is far more effective than LEX at finding placements that eventually lead to solutions, and it does so by ensuring that the queens most difficult to place are placed first, and the remaining queens are filled in around them.

One surprise uncovered by the studies is that MIN was not dramatically better then LEX in a full search in spite of its excellent performance on terse search. But full search was conducted only up to $N = 13$, and MIN's performance may be relatively better for larger N. Data for terse searches

Table 2 Terse search: Number of backtracks per solution.

N	*1 MIN*	*10 MIN*	*1 LEX*	*10 LEX*
7	0	10	2	12
8	68	26	105	40
9	14	16	32	25
10	20	35	92	65
11	54	57	41	120
12	125	107	249	235
13	157	82	98	213
14	109	119	1,885	385
15	19	81	1,344	859
16	28	137	10,036	2,546
17	42	103	5,357	1,628
18	37	118	41,281	11,114
19	25	76	2,526	3,797
20	130	73	199,615	28,088
21	22	63	8,541	6,353
22	12	46	1,737,166	215,652
23	14	58	25,406	11,029
24	14	111	411,584	165,661
25	220	116	48,658	23,061
26	227	145	397,673	111,125
27	72	31	454,186	61,343
28	0	28	3,006,270	301,220
29	313	101	1,532,210	453,699

comparing MIN to LEX appear in **Table 2**. The data show that MIN is several orders of magnitude more efficient than LEX for N between 20 and 30. We conjecture that the cost of terse LEX search climbs exponentially for $N \geq 30$, and consequently, MIN could easily be 50 to 100 orders of magnitude faster than LEX for $N > 90$.

5. Conclusions

The conclusions of Stone and Sipala, confirmed by Nicol, indicate that the average complexity of search can be very low in spite of an exponential upper bound on complexity. But those conclusions were for a theoretical model, and not based on actual statistics for a real problem. This paper extends that research by showing the following results:

1. The N-Queens Problem has an exponentially increasing complexity for lexicographic search.
2. The N-Queens Problem does not fit the Stone-Sipala model because it is not everywhere expanding or everywhere contracting.
3. The exponential behavior of lexicographic search is due to explorations in the middle of the search tree that follow paths that almost always fail.
4. The search complexity for terse MIN search grows very slowly with N for N up to 96. Above that value, the complexity may grow much faster.
5. The statistical variations within a search lead to regions in the search tree where the solution rate is much faster or much slower than the average solution rate.
6. Because of statistical variations, a constrained search

might produce a solution much faster than an unconstrained search. We demonstrated this by showing that solutions for N even can be produced from 2 to 70 times faster by solving a problem for larger N.

The most important conclusion to draw from this work is that MIN appears to be extremely effective for solving the search problem posed here, and is sufficiently general to fit in many other contexts. This confirms the suggestions of Bitner and Reingold, and the intuition underlying the work of Purdom, Brown, and others. The fact that MIN brought the complexity of terse search down so dramatically was unexpected.

A second major conclusion is that the average complexity of search need not be exponential. We have demonstrated by example that search for which lexicographic ordering yields exponential average behavior can be solved orders of magnitude faster by other search strategies. It is essential in finding such strategies that some cleverness be used, and in developing a strategy we recommend that the search-tree statistics be studied to see where they can be exploited to make the search go faster. Through the use of constraints to guide search, a MIN strategy, or other methods, it may be possible to find solutions to huge searches in a reasonable time.

References

1. Harold S. Stone and Paolo Sipala, "The Average Complexity of Depth-First Search with Backtracking and Cutoff," *IBM J. Res. Develop.* **30,** 242–258 (May 1986).
2. D. Nicol, "Expected Performance of *m*-Solution Backtracking," *NASA ICASE Report No. 8655*, August 1986.
3. J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. Electron. Computers* **EC-16,** 567–580 (1967).
4. C. Brown and P. W. Purdom, Jr., "An Average Time Analysis of Backtracking," *SIAM J. Comput.* **10,** 583–593 (August 1981).
5. P. W. Purdom, Jr., and C. A. Brown, "An Analysis of Backtracking with Search Rearrangement," *SIAM J. Comput.* **12,** 717–733 (November 1983).
6. P. W. Purdom, Jr., "Search Rearrangement Backtracking and Polynomial Average Time," *Artif. Intell.* **21,** 117–133 (1983).
7. C. Brown and P. W. Purdom, Jr., "An Empirical Comparison of Backtracking Algorithms," *IEEE Trans. Pattern Anal. & Machine Intell.* **PAMI-4,** 309–316 (May 1982).
8. P. W. Purdom, Jr., C. A. Brown, and E. L. Robertson, "Backtracking with Multi-Level Dynamic Search Rearrangement," *Acta Informat.* **15,** 99–113 (1981).
9. J. Bitner and E. M. Reingold, "Backtrack Programming Techniques," *Commun. ACM* **18,** 651–655 (1975).
10. D. E. Knuth, "Estimating the Efficiency of Backtracking Programs," *Math. Comp.* **29,** 121–136 (1975).
11. P. W. Purdom, Jr., "Tree Size by Partial Backtracking," *SIAM J. Comput.* **7,** 481–491 (1977).

Received August 29, 1986; accepted for publication February 12, 1987

Harold S. Stone *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Stone is the manager of advanced architecture studies at the IBM Thomas J. Watson Research Center. He has been a faculty member at the University of Massachusetts and Stanford University and has held visiting faculty appointments at institutions throughout the world. He is the author, coauthor, or editor of six textbooks, and has produced over sixty technical publications. The series he has produced as a consulting editor to Addison-Wesley, McGraw-Hill, and University Microfilms contain more than seventy titles in all areas of computer science and engineering. Dr. Stone received a Ph.D. in electrical engineering in 1963 from the University of California at Berkeley. His research contributions have been primarily in computer architecture and digital systems design. He has been active in both the IEEE Computer Society and ACM. For the ACM he has served as Associate Editor of the *J. ACM*, and for the IEEE Computer Society he has served as Technical Editor of *Computer Magazine* and Governing Board Member. He is a Fellow of the IEEE.

Janice M. Stone *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Ms. Stone is a staff programmer in the experimental languages and concurrent systems group at the IBM Thomas J. Watson Research Center. She received the A.B. degree in mathematics from Duke University, Durham, North Carolina, in 1962, and pursued graduate studies in mathematics at Georgetown University and in logic and philosophy of science at Stanford University. She has worked as a software consultant and independent software producer, and as an adjunct assistant professor at Hampshire College. In 1984 Ms. Stone joined IBM Research; her research interests focus on parallel algorithms and tools for development and analysis of parallel programs.