(appeared in SIGART Bulletin, Vol. 1, 3, pp. 7-11, Oct, 1990.)

A Polynomial Time Algorithm for the N-Queens Problem[1]

Rok Sosič and Jun Gu

Department of Computer Science[2]
University of Utah
Salt Lake City, UT 84112

Summary

The n-queens problem is a classical combinatorial problem in the artificial intelligence (AI) area. Since the problem has a simple and regular structure, it has been widely used as a testbed to develop and benchmark new AI search problem-solving strategies. Recently, this problem has found practical applications in VLSI testing and traffic control. Due to its inherent complexity, currently even very efficient AI search algorithms developed so far can only find a solution for the n-queens problem with n up to about 100. In this paper we present a new, probabilistic local search algorithm which is based on a gradient-based heuristic. This efficient algorithm is capable of finding a solution for extremely large size n-queens problems. We give the execution statistics for this algorithm with n up to 500,000.

Keywords: Artificial intelligence (AI), combinatorial search, gradient-based heuristic, local search, the n-queens problem, nonbacktracking search, fast search algorithm.

[1]This research has been supported in part by the University of Utah research fellowships, in part by the Research Council of Slovenia, in part by the 1987-88 and 1988-89 ACM/IEEE academic scholarship awards.

[2]Jun Gu is also with Department of Electrical Engineering, University of Calgary, Calgary, Canada T2N 1N4. E-mail: sosic@cs.utah.edu,gu@enel.UCalgary.CA.

1 Introduction

The n-queens problem is a classical combinatorial problem in the AI search area. We are particularly interested in the n-queens problem since it is a relatively simple yet nontrivial case study and testbed in which to explore general issues of designing efficient AI search algorithms and predicting their performance [3]. Also, it has recently found practical applications in VLSI testing and traffic control. Due to the exponential growth of the search load in the n-queens problem, even very efficient AI search algorithms can only handle the complexity (i.e., find out a solution) for about 100-queens [5, 11]. There has been little progress in exploring the n-queens problem for larger sizes during the last decade.

In this manuscript we give a new, probabilistic local search algorithm which is based on a gradient-based heuristic [8, 9]. This algorithm is capable of providing a solution for an extremely large size n-queens problem in several CPU hours on a $NeXT$ personal computer. We give the execution statistics of this fast algorithm with n up to 500,000. We believe that this new algorithm, its search technique, and the results of the n-queens problem may shed light on understanding other constraint-based AI search problems.

In Section 2, the n-queens problem is briefly introduced. Our new algorithm and its search techniques for the n-queens problem are described in Section 3. We show the run-time behavior of this new algorithm in Section 4. The conclusions are given in Section 5.

2 The N-Queens Problem

The 4-queens problem is the simplest instance of the n-queens problem with solutions. The problem is to place four queens on a 4×4 chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal. In the general n-queens problem, a set of n queens is to be placed on an $n \times n$ chessboard so that no two queens attack each other.

In the following discussion, we assume that each row will be occupied by a single queen. The four queens, in the 4-queens problem, are labeled with the numbers 1 through 4. Any possible solution of the 4-queens problem can be represented as the 4-tuple $(q_1, \ldots, q_4)$, where q_i is a column position on which the queen in the i-th row is placed.

In a little known work, Ahrens [1] describes a method to compose a solution to the general n-queens problem by patching together solutions to the smaller sized problems. This analytical solution has an inherent limitation in that it will generate only a very restricted class of solutions. This is not the case with search based algorithms.

One method for solving the n-queens problem which systematically generates all possible solutions is known as *backtracking search*. Since the nature of backtracking search is exponential in time, backtracking search is not able to solve the large size n-queens problem [3, 4, 7, 2, 10, 11]. Recent results indicate that we may only solve the n-queens problem with n up to about 100 [5, 11].

It is desirable to investigate some alternative search approaches in which there is no backtrack overhead involved. In the next section, we give a new probabilistic local search algorithm that is based on a gradient-based heuristic. The algorithm runs in polynomial time, does not use backtracking, and is capable of finding a solution for an extremely large size n-queens problem within a reasonably short time period.

```
1.  **function** queen_search($queen$ : **array** [1..$n$] **of integer**)
2.  **begin**
3.      **repeat**
4.          Generate a random permutation of $queen_1$ to $queen_n$;
5.          **forall** $i, j$; where $queen_i$ **or** $queen_j$ is attacked **do**
6.              **if** swap($queen_i$,$queen_j$) reduces collisions
7.              **then** perform_swap($queen_i$, $queen_j$);
8.      **until** no collisions;
9.  **end**;
```

Figure 1: A Fast N-Queens Search Algorithm

3 A Fast Algorithm for the N-Queens Problem

Let:

1. $\pi(i)$ $(i = 1, ..., n)$ be a permutation for integer numbers 1, ..., n, and
2. $\{row_i, column_{\pi(i)}\}$ $(i = 1, ..., n)$ be n coordinates of positions for n queens on a chessboard.

Since there is only one queen to be placed on each row, row_i can be represented by index i, and the exact position of the n queens on the chessboard can be fully specified by the column numbers (an n-tuple) of the n queens. This n-tuple of column numbers can be represented in a linear array of size n. That is, let $\{column_{\pi(i)}\}$, or abbreviated as $\{\pi(i)\}$ $(i = 1, ..., n)$, be the n positions of n queens on a chessboard.

For any permutation, the above formulation of the queens' positions guarantees that no two queens will attack each other on the same row or the same column. The problem then remains to resolve any collisions among queens that may occur on the diagonals.

Our new algorithm is shown in Figure 1. At the beginning of a search, a random permutation of the column positions of the queens is generated. This initial permutation of column positions generally produces collisions among queens on the diagonals. The number of collisions can be counted by tracing each negative (slope) diagonal line and each positive (slope) diagonal line using the method described below.

Let i be a *row index* and j be a *column index*, then the *sum* of both indexes is constant on any negative diagonal line, and the *difference* of both indexes is constant on any positive diagonal line. The values of the *sum* on different diagonal lines are different, so are the values of *differences*. Corresponding to *row index* i and *column index* j, since the column positions of n queens are specified by a permutation π, the *sum* is calculated as $i + \pi(i)$ and the *difference* as $i - \pi(i)$, for $i = 1, ..., n$.

For the n-queens problem, there are $2n - 1$ negative diagonal lines and $2n - 1$ positive diagonal lines on the chessboard. There is an array of size $2n - 1$, called d_1, that keeps track of the number of queens, i.e., the number of collisions, on each of the $2n - 1$ negative diagonal lines. If there are k queens on the m_{th} negative diagonal line, there are $k - 1$ collisions on this diagonal line. The number k is written into the m_{th} element of the d_1 array. Similarly, we choose another array with size $2n - 1$, called d_2, for $2n - 1$ positive diagonal lines.

```
1.  **repeat**
2.      swaps_performed := 0;
3.      **for** $i$ **in** $[1..n]$ **do**
4.          **for** $j$ **in** $[(i+1)..n]$ **do**
5.              **if** $queen_i$ is attacked **or** $queen_j$ is attacked **then**
6.                  **if** swap($queen_i, queen_j$) reduces collisions **then begin**
7.                      perform_swap($queen_i, queen_j$);
8.                      swaps_performed := swaps_performed + 1;
9.                  **end**;
10. **until** swaps_performed = 0;
```

Figure 2: A Gradient-Based Heuristic

As described in Figure 1, a random permutation of the column positions for n queens is generated at the beginning of the search. This initial permutation generally causes some collisions on the diagonals. The number of collisions on diagonals is counted and stored into arrays d_1 and d_2.

A gradient-based heuristic, as shown in Figure 2 (i.e., lines 5-7 in Figure 1), plays an important role in this fast queen search algorithm to navigate the search activity through a simple local search. The main idea behind this heuristic is to *swap* a pair of queens so that the total number of collisions (on both negative and positive diagonals) is reduced. Before a *swap* action is taken, a local search is performed. We must first determine the "direction" to proceed, i.e., the "gradient direction" in the search space that points to the direction that may reduce the number of collisions among the queens. The idea is pretty simple. Before and after the swap of a pair of queens, the number of collisions on the diagonals are compared. If a swap of a pair of queens reduces the number of collisions, the swap action is performed; otherwise, no action is taken.

In the fast search algorithm, the gradient-based heuristic is applied to all possible pairs of queens (see Figure 1) until there are no collisions left, that is, a solution is found. If no solution could be found for that initial permutation, a new permutation is generated and a new search process is started.

The swap action incrementally updates arrays d_1 and d_2. Since one queen can affect at most two diagonals (i.e., one negative diagonal and one positive diagonal), correspondingly at most two values in arrays d_1 and d_2, i.e., $i + \pi(i)$ and $i - \pi(i)$, are affected. A swap of two queens can affect at most eight diagonals: four for both "source" queens and four for both "destination" queens. In order to test if a swap reduces the number of collisions we need only to check these eight diagonals. The number of operations in a swap action is therefore constant, and obviously does not depend on n. This test operation and a possible subsequent swap operation are repeated for all possible pairs of queens until a solution is found. If no more swaps can be performed and collisions still exist, a new permutation is invoked. The implementation of the above algorithm is straightforward.

The running time of the algorithm can be estimated as follows. The generation of a random permutation (line 4 in Figure 1) can be done in linear time [6]. Regardless of the board size, the testing and swap operations (lines 5-9 in Figure 2) can be evaluated in constant time as described above. Therefore, the number of testing and swap operations determines algorithm performance. In the worst case, each iteration of the repeat loop in Figure 2 requires $O(n^2)$ evaluations since there are two **for** loops (lines 3-4 in Figure 2). Since each execution of the **repeat** loop must decrease the number of collisions, which is at most $n - 1$, the upper bound on the running time of the gradient-based heuristic is $O(n^3)$.

For an initial permutation, if no solution is found after the completion of the **repeat** loop, a new permutation is generated and a new search process is started. However, the number of permutations required to find a solution is very small. With increasing n, the number of permutations required to find a solution goes to 1. That is, for large n, the first permutation will always find a solution (See Table 4 in Section 4). Experimental results collected during the last several years show that the actual running time of the algorithm is, in practice, approximately $O(n \log n)$. These results are shown in the next section.

4 Results

Among the algorithm features we have studied, the following observations are of particular interest. We summarize some experimental data below.

- Real execution time of the algorithm.
- Number of initial collisions generated by a random permutation.
- The maximum number of queens on the same diagonal in a random permutation.
- The probabilistic behavior of the algorithm.

1. Real execution time of the algorithm.

The real execution time of our fast search algorithm, programmed in C and run on a *NeXT* personal computer (with a 25 MHz Motorola 68030 processor), is illustrated in Table 1. Since our algorithm takes polynomial time, it is incomparably faster than any present well-known AI search algorithms, all of which run in exponential time. Due to the memory limitation of our computer, the largest problem size we were able to run was 500,000.

2. Number of initial collisions generated by a random permutation.

The second observation made involved the number of collisions generated by a random permutation (See Table 2). This indicates the maximum number of swaps which may be required to find a solution. The results collected in Table 2 were based on the average of 100 random permutations. Theoretically, no more than $n-1$ collisions are possible on a board of size n, when all n queens are aligned on the same diagonal. So the number of collisions which must be resolved may increase only linearly in n. It is indicated from numerous real algorithm runs that the ratio between the number of collisions and the board size n in a random permutation approaches 0.5285 as n increases up to 500,000. Individual sample runs have shown a very small deviation from this number. Numbers in Table 2 actually present the upper bound on the number of swaps that may be performed to find a solution from an initial random permutation.

3. The maximum number of queens on the same diagonal.

As illustrated in Table 3, the maximum number of queens that attack each other on the same diagonal line was also analyzed. A total of 100 random permutations were generated for each board size shown and the maximum number of queens on one diagonal was recorded. The minimum and maximum values from these 100 permutations are very similar. That is, the collisions among queens on diagonals are basically evenly distributed. There are no specific diagonals that contain a large number of queens.

4. The probabilistic behavior of the algorithm.

Table 1: A Fast N-Queens Search Algorithm to Search for a Random Solution on a NeXT Machine with a 25Mhz Motorola 68030 Microprocessor (Average of 10 Runs; Time Units: seconds)

Number of Queens n	10	100	1,000	10,000	100,000	500,000
Time of the 1st run	< 0.1	0.4	2.1	27.7	1,098.4	7,500
Time of the 2nd run	< 0.1	0.2	1.9	38.2	1,081.2	9,065
Time of the 3rd run	< 0.1	< 0.1	1.8	42.6	997.6	12,617
Time of the 4th run	< 0.1	< 0.1	3.1	34.9	979.9	11,730
Time of the 5th run	< 0.1	< 0.1	1.9	34.3	1,286.4	9,934
Time of the 6th run	< 0.1	< 0.1	2.4	31.2	992.3	9,198
Time of the 7th run	< 0.1	0.2	1.9	41.2	1,425.5	9,789
Time of the 8th run	< 0.1	0.3	3.3	36.5	1,235.4	11,142
Time of the 9th run	< 0.1	0.1	2.3	52.4	1,285.7	11,788
Time of the 10th run	< 0.1	< 0.1	2.1	35.1	1,285.4	8,300
Ave. Time to Find a Solution	< 0.1	0.1	2.3	37	1,167	10,106

Table 2: Number of Collisions Among Queens in a Random Permutation (Average of 100 Permutations)

Number of Queens n	10	100	1,000	10,000	100,000	500,000
Num. of Collisions/n	0.486	0.523	0.5277	0.5283	0.528694	0.528511

Table 3: Maximum Number and Minimum Number of Queens on the most Populated Diagonal in a Random Permutation (Average of 100 Runs)

Number of Queens n	10	100	1,000	10,000	100,000	500,000
Minimum	2	5	7	7	9	10
Maximum	5	7	7	9	9	10

Table 4: Permutation Statistics (Average of 10 Runs)

Number of Queens n	10	100	1,000	10,000	100,000	500,000
Solution in the First Permutation	2	6	8	10	10	10
Max. Num. of Permutations	10	3	2	1	1	1

Table 5: Swap Statistics (Average of 10 Runs)

Number of Queens n	10	100	1,000	10,000	100,000	500,000
Num. of Pairs Tested	353	13,525	253,671	4,827,973	110,186,345	967,924,234
Num. of Swaps Tested	198	2,385	15,116	166,215	2,034,907	11,447,508

Table 4 and Table 5 were obtained from 10 sample algorithm runs. The algorithm is probabilistic. That is, if the algorithm could not find a solution from a given random permutation, a new permutation is generated and the algorithm starts a new search.

Table 4 shows the probabilistic behavior regarding algorithm success in finding a solution from an initial random permutation. The *solution in the first permutation* represents, among 10 sample algorithm runs, the number of times a solution is found based on an initial (the first) permutation. The *maximum number of permutations* is, within 10 sample algorithm runs, the maximum number of permutations that were required to find a solution in one program run. It can be seen that the number of required permutations decreases with increasing n. For n equals to 100, the algorithm succeeded in the first permutation in 6 of 10 sample runs. In the worst case, only 3 permutations were required. In our measurements, for n greater than 10,000, the algorithm always finds a solution in the first permutation.

Table 5 shows parts of the program on which the most time was spent. The *number of pairs tested* gives the total number of pairs checked for collision (line 5 in Figure 2). The *number of swaps tested* indicates a total number of calls to the swap testing (line 6 in Figure 2).

5 Conclusion

An efficient, fast search algorithm able to find a solution for millions of queens is presented. The algorithm runs in polynomial time as compared to the exponential time of the present AI search algorithms. This performance is achieved by applying a clever, gradient-based heuristic within a local search.

6 Acknowledgement

We thank Vladimir Batagelj who pointed out a reference to the earlier work by Ahrens [1]. We appreciate the relevant and instructive comments provided by several reviewers.

References

[1] W. Ahrens. *Mathematische Unterhaltungen und Spiele (in German).* B.G. Teubner (Publishing Company), Leipzig, 1918-1921.

[2] R. Dechter and J. Pearl. *Network-Based Heuristics for Constraint-Satisfaction Problems. Artificial Intelligence*, 34:1–38, 1988.

[3] J. Gaschnig. *Performance Measurements and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, May 1979.

[4] R. M. Haralick and G. Elliot. *Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence*, 14:263–313, 1980.

[5] *Panel Discussion: Parallel Processing for Non-Numeric Applications.* International Conference on Parallel Processing, Chicago, August 15 1990.

[6] L.E. Moses and R.V. Oakford. *Tables of Random Permutations.* Stanford University Press, Palo Alto, California, 1963.

[7] B. A. Nadel. *Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens. IEEE Expert*, pages 16–23, Jun. 1990.

[8] R. Sosic and J. Gu. *Fast N-queen Search on VAX and Bobcat Machines.* AI Project Report, Feb. 1988.

[9] R. Sosic and J. Gu. *How to Search For Million Queens.* Technical Report UUCS-TR-88-008, Dept. of Computer Science, Univ. of Utah, Feb. 1988.

[10] H. S. Stone and P. Sipala. *The Average Complexity of Depth-first Search with Backtracking and Cutoff. IBM J. Res. Develop.*, 30(3):242–258, May 1986.

[11] H. S. Stone and J. M. Stone. *Efficient Search Techniques – An Empirical Study of The N-Queens Problem. IBM J. Res. Develop.*, 31(4):464–474, July 1987.