3,000,000 Queens in Less Than One Minute[1]

Rok Sosic
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
sosic@cs.utah.edu

Jun Gu
Department of Electrical Engineering
University of Calgary
Calgary, Canada T2N 1N4
gu@enel.ucalgary.ca

Summary

The n - queens problem is a classical combinatorial search problem. In this paper we give a linear time algorithm for this problem. The algorithm is an extension of one of our previous local search algorithms [3, 4, 6]. On an IBM RS 6000 computer, this algorithm is capable of solving problems with 3,000,000 queens in approximately 55 seconds.

Keywords: Gradient–based conflict minimization heuristic, local search, the n - queens problem, probabilistic search algorithms.

1 Introduction

The n - queens problem is a classical combinatorial search problem. The problem is to place n queens on an n x n chessboard so that no two queens attack each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal line.

One method for solving the n - queens problem which systematically generates all possible solutions is known as *backtracking search*. Due to the exponential growth of the search load in backtracking, this type of search is not able to solve large size n - queens problems. Even very efficient AI search algorithms can only find a solution for the n - queens problem with n less than 100 [1, 2, 8, 9]. Recently we gave a polynomial time local search algorithm that employs a gradient–based conflict minimization heuristic. [3, 4, 6]. This algorithm can find a solution for very large size n - queens problems. In [5], we have compared the real–time running results of this algorithm with constraint–based backtracking search. To distinguish the original algorithm from the new algorithm, we call it the *Queen Search 1 (QS1)* algorithm.

We present a linear time *Queen Search 4 (QS4)* algorithm, which is derived from the *QS1*. This algorithm runs in linear time and uses the same conflict minimization heuristic as in *QS1*. The running time of *QS4* is much faster than the approximately $O(n \log n)$ execution time of the *QS1*, and is much faster than two of our previous near linear *QS2* and *QS3* algorithms. On an IBM RS 6000/530, for example, *QS4* solves 1,000,000 queens, 2,000,000 queens, and 3,000,000 queens problems in 17 seconds, 36 seconds, and 55 seconds, respectively. In this paper, we describe the *QS4* algorithm only.

Another fast algorithm based on the same conflict minimization heuristic and the same local search idea for solving the n - queens problem was presented in [7]. This algorithm solves the 1,000,000 queens problem in approximately the same time as our previous *QS3* algorithm.

2 The *QS1* Algorithm

The *QS1* algorithm and its execution statistics were presented in detail in [3, 4, 6]. Here we briefly outline the *QS1* algorithm.

Let n be the size of the board and let each queen be placed in one row only. When n queens are arranged on the board, their column positions are stored in array *queen* of length n. The i^{th} queen is placed on the board at row i and column *queen*[i]. We require that at any moment the array *queen* contains a permutation of integers 1, . . ., n. This guarantees that no two queens attack each other on the same row or the same column. The problem remains to resolve any collisions among queens possibly occurring on the diagonal lines.

At the beginning of *QS1*, a random permutation is generated. Collisions on the diagonal lines are eliminated simply by testing all possible pairs of queens. If the swap of queens in a pair reduces the number of collisions on the diagonal lines, then the swap is performed, otherwise no action is taken. The process is repeated until all collisions among queens are eliminated. This efficient gradient–based conflict minimization heuristic is used in QS1, QS2, QS3, and QS4 algorithms.

In our past several years of experimental experience with *QS1*, we found that this simple local search algorithm with a conflict minimization heuristic is able to find a solution within a small number of random permutations. It always found a solution in the first random permutation for a problem size greater than or equal to 1000. The real execution time of the *QS1* algorithm, programmed in C and run on an IBM RS 6000/530 computer is illustrated in Table 1.[2]

1. This research had been supported in part by the University of Utah research fellowships, in part by the Research Council of Slovenia, and in part by the ACM/IEEE academic scholarship awards.

2. Numbers in Table 1 are 3.5 times smaller than numbers in our previous paper [6], because IBM RS 6000 is about 3.5 times faster than NeXT which was used in our previous measurements.

3 The *QS4* Algorithm

The initial permutation in the *QS1* algorithm is completely random. It was observed in [3, 4, 6] that a random permutation of n integers generates approximately $n \times 0.53$ collisions on the diagonal lines. For example, a random permutation of 500,000 numbers generates approximately 265,000 collisions among queens.

The *QS1* algorithm can be made more efficient if the collisions in the initial permutation can be minimized. This is the basic idea behind the *QS4* algorithm.

In the *QS4* algorithm, an initial random permutation is generated such that the number of collisions among queens is minimized. Queens are placed on successive rows. The position for a new queen to be placed on the board is randomly generated from columns that are not occupied until a conflict free place is found for this queen. After a certain number of queens have been placed in a conflict free manner the remaining queens are placed randomly on free columns regardless of conflicts on diagonal lines. The number of queens with a possible conflict is denoted as c. This process of generating the initial permutation does not require backtracking.

The number of queens placed in a conflict free manner $n - c$ needs to be chosen carefully. If this number is too small the *QS4* algorithm shows no improvement over the *QS1* algorithm. If this number is too large the initialization either takes too long or does not terminate. The number of queens with a conflict c that we have chosen in our real time runs varies with n. Number c is shown together with n and the real execution time of the *QS4* algorithm in Table 2. It can be observed from experiments that c does not increase with increasing n. Although c could be set to 100 for all values of n, its value is optimized for smaller values of n.

After the initial permutation is generated, at most c queens need to be moved to find a solution. A search step for the *QS4* algorithm is similar to that of the *QS1* algorithm. The same gradient-based conflict minimization heuristic of *QS1* algorithm is applied here. Two queens are chosen. The first queen is systematically chosen from the c queens with a conflict, the second queen is chosen completely at random. If n is less than 1000, then the second queen is also chosen systematically. If the swap of the queens' column positions reduces the number of conflicts, the swap is performed, otherwise no action is taken. Search steps are performed until a solution is found.

Execution results of the *QS4* algorithm are shown in Table 2. Compared to our previous results of the *QS1* (see Table 1), the execution speed of the *QS4* is approximately 300 times faster for problem size 100,000. Numbers in the table are the total running time including initialization.

Number of Queens n	10	100	1,000	10,000	100,000
Time of the 1st run	0.0	0.1	0.4	7.4	340
Time of the 2nd run	0.0	0.0	0.3	8.6	371
Time of the 3rd run	0.0	0.1	0.5	5.9	327
Time of the 4th run	0.0	0.1	0.4	11.9	355
Time of the 5th run	0.0	0.1	0.3	19.0	320
Time of the 6th run	0.0	0.0	0.5	7.7	264
Time of the 7th run	0.0	0.0	0.5	5.8	298
Time of the 8th run	0.0	0.0	0.4	7.6	472
Time of the 9th run	0.0	0.1	0.5	10.5	278
Time of the 10th run	0.0	0.0	0.4	8.9	261
Avg. Time to Find a Solution	0.0	0.1	0.4	9.3	327

Table 1: The Execution Time of the QS1 Algorithm on an IBM RS 6000/530 Computer (Average of 10 Runs; Time Unit: seconds)

Number of Queens n	10	10^2	10^3	10^4	10^5	10^6	2×10^6	3×10^6
Queens with Conflict c	8	30	50	50	80	100	100	100
Time of the 1st run	0.0	0.0	0.0	0.1	1.1	16.9	35.7	54.8
Time of the 2nd run	0.0	0.0	0.0	0.1	1.1	17.0	35.8	54.8
Time of the 3rd run	0.0	0.0	0.0	0.1	1.1	17.1	35.8	54.6
Time of the 4th run	0.0	0.0	0.1	0.1	1.1	17.0	35.9	54.8
Time of the 5th run	0.0	0.0	0.0	0.1	1.1	17.0	35.8	54.7
Time of the 6th run	0.0	0.0	0.0	0.1	1.1	17.0	35.8	54.6
Time of the 7th run	0.0	0.1	0.1	0.1	1.1	17.0	35.8	54.7
Time of the 8th run	0.0	0.0	0.1	0.1	1.1	17.0	35.8	54.7
Time of the 9th run	0.0	0.0	0.1	0.1	1.1	17.0	35.8	54.7
Time of the 10th run	0.0	0.0	0.0	0.1	1.1	17.0	35.8	54.7
Avg. Time to Find a Solution	0.0	0.0	0.0	0.1	1.1	17.0	35.8	54.7

Table 2: The Execution Time of the QS4 Algorithm on an IBM RS 6000/530 Computer (Average of 10 Runs; Time Unit: seconds)

Recently, Minton et al. [7] reported a fast algorithm for the n - queens problem. For a million queens problem, on a SUN SPARC Station 1, it took 90 to 240 seconds to find a solution, depending on algorithm optimization. In Figure 2 of their paper, they showed that it took their algorithm on average approximately 240 seconds to find a solution. We have run the same size problem on a SUN SPARC Station, it took *QS4* steadily 38 seconds to find a solution, which is significantly faster than Minton's results.

The *QS4* algorithm spends most of its time in the initialization. The search process takes a negligible amount of time and is constant for all n greater than or equal to 1000. The effort in initialization increases linearly. For example, approximately 3,060,000 queen positions are tried for n equal to 1,000,000; approximately 9,180,000 queen positions are tried for n equal to 3,000,000. In summary, for each queen only three positions are tested on average before a conflict free position is found.

4 Conclusion

A linear time probabilistic local search algorithm that employs a gradient-based conflict minimization heuristic is presented. This algorithm is significantly faster than any presently known algorithm and is able to find a solution for extremely large size n - queens problems.

5 Acknowledgement

J. Mostow pointed out a reference of [7]. We gratefully acknowledge IBM for providing access to IBM RS 6000 for data collection. Comments from SIGART editor and reviewers are greatly appreciated.

6 References

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

[2] R. M. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence,* 14:263-313, 1980.

[3] R. Sosic and J. Gu. Fast n - queen search on VAX and Bobcat machines. Unpublished CS 547 AI Class Student Project Report, Winter Quarter1988.

[4] 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 (also available from authors).

[5] J. Gu. Parallel Algorithms and Architectures for Very Fast AI Search (PhD Thesis). Technical Report UUCS-TR-88-005, Univ. of Utah, Dept. of Computer Science, Jul.. 1988.

[6] R. Sosic and J. Gu. A polynomial time algorithm for the n - queens problem. *SIGART Bulletin,* 1(3):7-11, Oct. 1990.

[7] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI90,* pages 17-24, Aug. 1990.

[8] 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.

[9] H.S. Stone and J.M. Stone. Effecient search techniques - an empirical study of the n - queens problem. *IBM J. Res. Develop.,* 31(4):464-474, July 1987.