How to Search for Millions of Queens [1]

Rok Sosič and Jun Gu

UUCS-TR-88-~~008~~ 025

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

Feb. 1988
Rev. Dec. 1989, Feb., Mar., Apr. 1990

Summary

The n-queens problem is a classical combinatorial problem in artificial intelligence (AI) area. Since its simplicity and regular structure, this problem has widely been chosen as a testbed to develop and benchmark new AI search problem-solving strategies in the AI community. Due to its inherent complexity, so far even very efficient AI search algorithms can only find a solution for n-queens problem with n up to about 100. In this manuscript 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 over 1,000,000 queens in several CPU hours on a 25Mhz Motorola 68030 computer.

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.

1 Introduction

The n-queens problem is a classical combinatorial problem in AI search area. We are not interested primarily in the n-queens problem *per se*, but rather as a relatively simple yet nontrivial case study and testbed in which to explore general issues, principally the issue of designing efficient AI search algorithms and predicting their performance [4, 9]. Due to the exponential growth of the search load in the n-queens problem, in present day even very efficient AI search algorithms can only handle the complexity (i.e., find out a solution) for about 100-queens [9]. There is 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. This algorithm is capable of providing a solution for over 1,000,000 queens in several CPU hours on a 25Mhz Motorola 68030 computer. 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 Sections 2 and 3, the n-queens problem and the basic techniques to solve the n-queens problem are briefly introduced. Our new algorithm and its search technique for the n-queens problem are described in Section 4. We show the run-time behavior of this new algorithm and its comparisons with other previous work in Section 5. The conclusions are given in Section 6.

2 The N-Queens Problem

The 4-queens problem is the simplest instance of the n-queens problem. This problem is to place four queens on a 4×4 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. 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 one queen only. 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.

3 Techniques to Solve the N-queens Problem

One method for solving the n-queens problem which systematically generates all possible solutions is known as *backtracking search.* A search problem involving backtracking can be represented in a (search) tree representation. As shown in Figure 2, for the 4-queens problem, each 4-tuple forms a possible path from the root to a leaf node in the search tree.

Here we start with the first column and first row, and assign to it the first queen from the 4-tuple. Then we select the second queen from the 4-tuple and assign it to the second row such that no queens attack. We continue with the placement of the next queen from the 4-tuple to the next row until all queens are placed. If at any point we run out of column positions for a queen to be placed, we simply go back one step (backtrack), chose another conflict-free column position on the previous row, and continue the process. If we are able to assign column positions to all the queens then we have found a solution; otherwise, there are no solutions.

In the 4-queens problem, the backtracking search tree consists of only 2 leaf nodes (2 solutions). One of the solutions and a partial search tree in this case are shown in Figures 1 and 2.

There are many variations of backtracking search that improve search efficiency, for

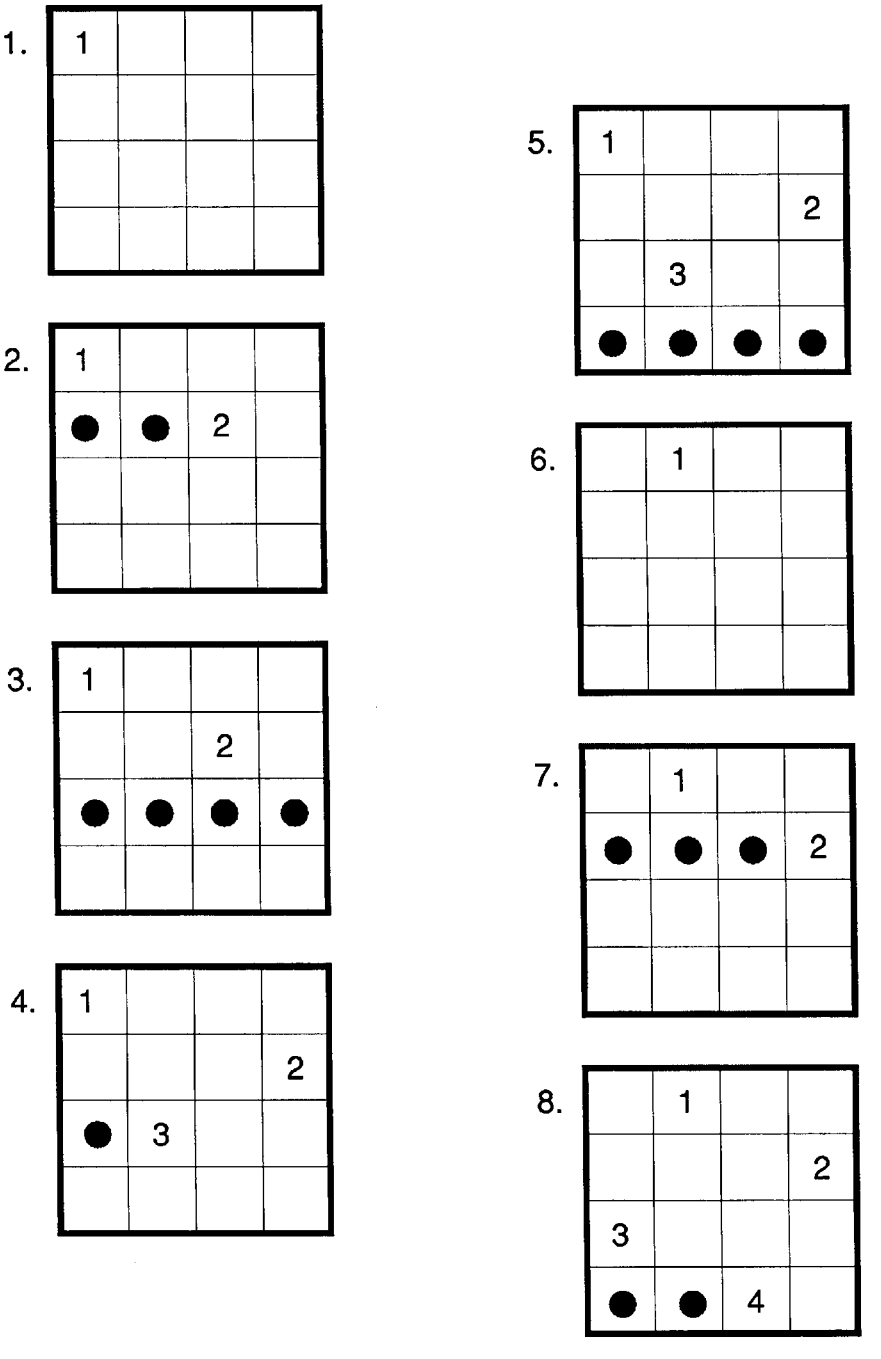

Figure 1: A Backtracking Solution for the 4-Queens Problem

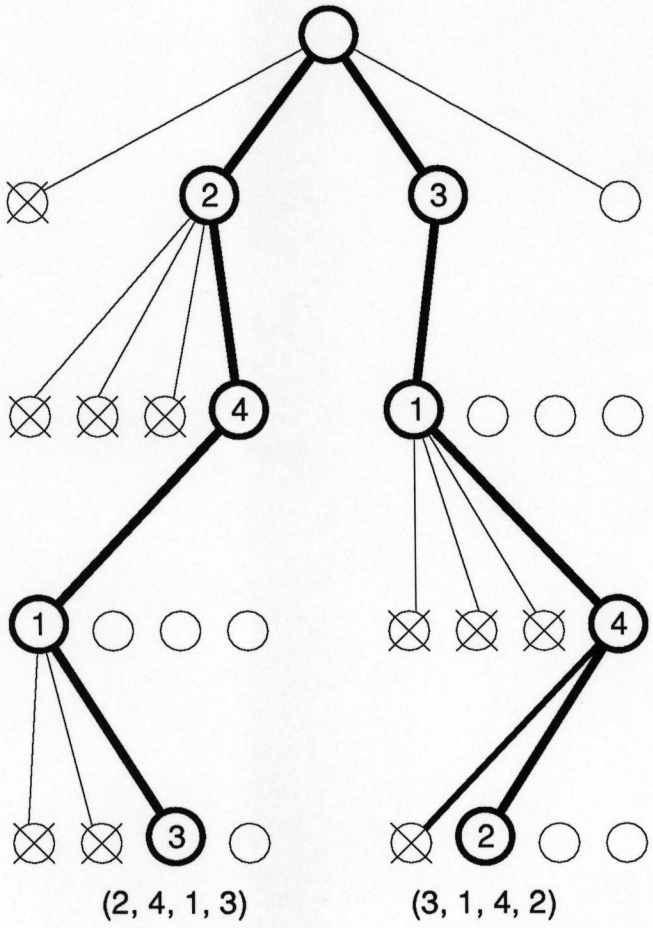

Figure 2: A Partial Search Tree for the 4-Queens Problem

example, backmarking, backjump, partial lookahead, and forward checking [2, 3, 4, 5]. Since the nature of backtracking search is exponential in time, however, none of them are able to solve the large size n-queens problem. Recent results indicated that we could only solve the n-queens problem with n up to about 100 [8, 9].

In general the upper bound on backtracking search effort is exponential in n (the depth of the search tree). The capability of backtracking search is thus insignificant for large size n-queens problems. It is therefore desirable to investigate some alternative search approaches in which there is no backtrack overhead involved. In next section, we give a new probabilistic local search algorithm that is based on a gradient-based heuristic. The algorithm runs in polynomial time and does not use backtracking at all. It is capable of finding a solution for over 1,000,000 queens within a reasonably short time period.

4 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 fully be specified by the column numbers of the n queens (an n-tuple). 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 queens' positions guarantees that no two queens will attack each other on the same row or the same column. The problem then

```
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 3: A Fast N-Queens Search Algorithm

remains to resolve any collisions among queens that may occur on the diagonals.

Our new algorithm is shown in Figure 3. 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 amount of collisions can be counted by tracing each negative (slope) diagonal line (See Figure 4a) and each positive (slope) diagonal line (See Figure 4b), using an unique method as 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 (See Figure 4a and Figure 4b). There is an array of size $2n - 1$, called d_1, that keeps tracking 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

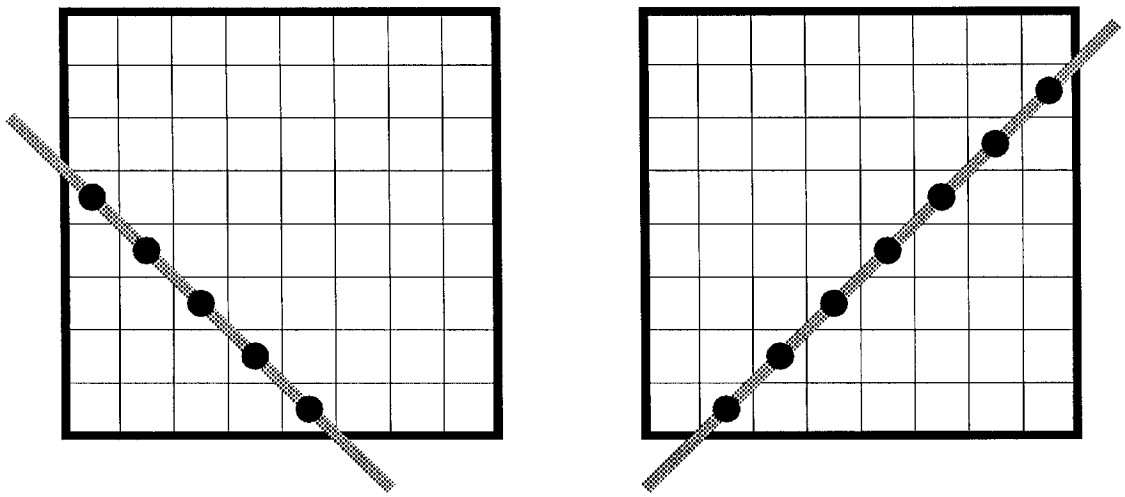

Figure 4: A Characterization of Diagonals

diagonal line, there are $k-1$ collisions on this diagonal line. The number $k-1$ 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.

As described in Figure 3, a random permutation of the column positions for n queens is generated at the beginning of the search. This initial permutation generally generates some collisions on diagonals. The amount of collisions on diagonals is counted and stored into arrays d_1 and d_2.

A gradient-based heuristic, as shown in Figure 5 (i.e., lines 5-7 in Figure 3), 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 the number of collisions among queens can be reduced. The idea is pretty simple. Before and

```
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 5: A Gradient-Based Heuristic

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 amount of collisions, the swap action is performed; otherwise, no action is taken.

In the fast search algorithm, this gradient-based heuristic is applied for all possible pairs of two queens (see Figure 3) until there is 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, one negative diagonal and one positive diagonal, and 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 there are still collisions existing, a new permutation is

invoked. The implementation of 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 3) can be done in linear time [7]. The testing and swap operations (lines 5-9 in Figure 5) can be evaluated in constant time regardless of the board size as described above. Thus the number of testing and swap operations determines algorithm performance. At the worst case, each iteration of the repeat loop in Figure 5 requires $O(n^2)$ evaluations since there are two **for** loops (lines 3-4 in Figure 5). 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 $O(n^3)$. Experimental results presented in the next section show that the actual running time is in practice approximately $O(n \log n)$.

5 Results and Comparison

There is very little literature in which some real simulation or execution results of an n-queens problem could be found. In addition, there have seldom been any results that were obtained on a clear comparable basis. Thus it is obviously infeasible for one to make any absolute comparisons and derive any firm conclusions. In what follows next, we list some results we have found so far together with the explanation of their specific problem instances (e.g., algorithm, solution category, implementation language, machine architecture, and result, etc.).

5.1 Backtracking-Based Search

Table 1 gives real algorithm execution statistics for three special consistent labeling algorithms [5] to search for one solution for the n-queens problem. The algorithms were programmed using Common Lisp and were run on an HP BobCat workstation. To reduce the

Table 1: Backtracking Algorithms to Search for One Solution for the N-Queens Problem (Time Unit: seconds).

Number of Queens n	5	7	9	11	13	15
1. Full Lookahead	3.06	17.44	75.00	193.32	421.16	806.06
2. Partial Lookahead	2.04	10.22	45.02	112.86	240.84	445.46
3. Forward Checking	1.02	3.46	18.16	28.82	53.66	76.82
4. Speedup (3 vs. 1)	3.00	5.04	4.12	6.71	7.84	10.49
5. Speedup (3 vs. 2)	2.00	2.95	2.48	3.92	4.49	5.80

performance variation caused by different factors (language, machine, etc.), speedup figures are also illustrated in Table 1. The last row shows the speedup figures between the forward checking algorithm and the full lookahead algorithm.

Table 2 shows the real algorithm running results obtained within a logic programming environment (implemented inside an MU-Prolog interpreter on a DEC $VAX-785$ machine) for several constraint satisfaction algorithms. The results for forward checking (FC) (in the second row) are better than those in Table 1 for $n \leq 14$. This work indicates that logic programming can be used to solve small size n-queens problem with an efficiency comparable to those codes written in imperative languages [6].

Stone and Stone [9] indicate that they found one solution for 96-queens problem using MIN search in less than 5 seconds. It takes the most costly MIN search 1,100 seconds to find one solution for the 93-queens problem.

5.2 Our Fast Search Algorithm

Among a good number of features we have studied, the following several experiments are of our particular interests. We summarize some experimental data below.

- Real execution time of the algorithm;

Table 2: Logic Programming of CLP to Search for the First Solution for the N-Queens Problem (Time Unit: seconds).

Number of Queens n	4	6	8	10	12	14	16	96
1. Standard Backtracking	0.11	0.68	3.40	4.38	14.81	130.92	813.64	
2. Forward Checking (FC)	0.10	0.46	1.53	1.70	4.22	34.03	185.78	
3. Generalized FC (GFC)	0.09	0.25	0.74	0.91	2.08	12.26	70.07	
4. GFC with Fail-First	0.09	0.29	0.77	0.57	1.74	1.73	1.09	36.23

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

1. Real execution time of the algorithm.

The real execution time of our fast search algorithm that was programmed in C and run on a *NeXT* computer (with a 25 MHz Motorola 68030 processor) is illustrated in Table 3. Since our algorithm takes polynomial time, it is incomparably faster than any presently best-known AI search algorithm which all run in an exponential time. Due to the memory limitation of our computer, the largest problem size we are able to run is 500,000.

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

The second observation was made on the number of collisions generated by a random permutation (See Table 4). This indicates the maximum number of swaps which may be required in order to find a solution. The results collected in Table 4 were averaged based on 100 random permutations. Theoretically, at most $n-1$ 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

Table 3: 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 4: 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

random permutation approaches 0.5285 with increasing n up to 500,000. Individual sample runs have shown a very small deviation from this number. Numbers in Table 4 actually present an upper bound on the number of swaps that may be performed in order to find a solution from an initial random permutation.

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

As illustrated in Table 5, 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 minimal and maximal values from these 100 permutations are very close. That is, the

Table 5: 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 6: 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

collisions among queens on diagonals are fairly evenly distributed. There are no specific diagonals that contain a large number of queens.

4. The probabilistic behavior of the algorithm.

Table 6 and Table 7 were obtained from 10 sample algorithm runs. The algorithm is probabilistic. If the algorithm could not find a solution from a given random permutation, a new permutation is required and the algorithm starts a new search.

Table 6 shows some probabilistic behavior regarding how successful is the algorithm 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

Table 7: 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

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. At the worst case, only 3 permutations were required. For n equals to 10,000 or larger, the algorithm always finds a solution at the first permutation.

Table 7 shows parts of the program of which the most time was spent. The *number of pairs tested* gives a total number of pairs checked for collision (line 5 in Figure 5). The *number of swaps tested* indicates a total number of calls to the swap testing (line 6 in Figure 5).

6 Conclusion

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

References

[1] W. Ahrens. *Mathematische unterhaltungen und spiele.* B. G. Teubner, Leipzig, 1918-21 (in German).

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

[3] J. Gaschnig. *A Constraint Satisfaction Method for Inference Making.* In *Proceedings of 12th Annual Allerton Conf. Circuit System Theory*, 1974.

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

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

[6] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, 1989.

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

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