

*(appeared in IEEE Transactions on Knowledge and Data Engineering,  
Vol. 6, 5, pp. 661-668, Oct 1994.)*

## **Efficient Local Search with Conflict Minimization: A Case Study of the $N$ -Queens Problem<sup>1</sup>**

Rok Sosič, Student Member, IEEE and Jun Gu, Senior Member, IEEE

Department of Computer Science	Department of Electrical Engineering
University of Utah	University of Calgary
Salt Lake City, UT 84112	Calgary, Canada T2N 1N4
sosic@cs.utah.edu	gu@enel.ucalgary.ca

### **Abstract**

Backtracking search is frequently applied to solve a constraint-based search problem but it often suffers from exponential growth of computing time. We present an alternative to backtracking search: local search based on conflict minimization. We have applied this general search framework to study a benchmark constraint-based search problem, the  $n$ -queens problem.

An efficient local search algorithm for the  $n$ -queens problem was implemented. This algorithm, running in linear time, does not backtrack at all. It is capable of finding a solution for extremely large size  $n$ -queens problems. For example, on a workstation computer, it can find a solution for 3,000,000 queens in less than 55 seconds.

**Keywords:** conflict minimization, local search,  $n$ -queens problem, nonbacktracking search.

---

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

# 1 Introduction

A constraint-based search problem has three components: variables, values, and constraints. The goal is to find an assignment of values to variables such that all constraints are satisfied. Backtracking search is generally applied to solve a constraint-based search problem. Many heuristics have been developed to improve the efficiency of backtracking search, for example, search rearrangement [4, 17, 18], backmarking and backjump [7, 8], and lookahead and forward checking [12]. Suffering from exponential growth of computing time, backtracking search techniques are not able to solve a large-scale constraint-based search problem.

We present an alternative to backtracking search – local search with conflict minimization. We describe a case study of this general framework on a benchmark constraint-based search problem, the  $n$ -queens problem. The  $n$ -queens problem is to place  $n$  indistinguishable objects on an  $n \times n$  grid so that no two objects are placed on the same row, the same column, or the same diagonal.

The purpose of this study is twofold. First, the  $n$ -queens problem is a typical problem of constraint satisfaction. A general search algorithm to the  $n$ -queens problem is useful in solving other constraint satisfaction problems [11]. Secondly, the  $n$ -queens problem is itself a model of *the maximal coverage problem*. A solution to the  $n$ -queens problem guarantees that each object can be accessed by the outside world from any one of its eight neighboring directions (two vertical, two horizontal, and four diagonal directions) without a conflict with other objects (see Figure 1). Our general search algorithm for the  $n$ -queens problem has made many applications possible [11]. Some direct applications of this result include VLSI testing, air traffic control, modern communication systems, data/message routing in a multiprocessor computer, load balancing in a multiprocessor computer, computer task scheduling, computer resource management, optical parallel processing, and data compression.

The rest of the paper is organized as follows. In Section 2, we will briefly review prior work. A linear time algorithm to solve the  $n$ -queens problem is presented in Section 3. A mathematical analysis of the algorithm is given in Section 4. Performance results are shown in Section 5. We describe a representative application of the  $n$ -queens problem in Section 6. Section 7 concludes this paper.

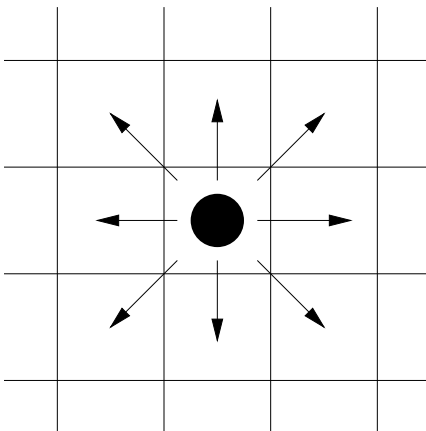


Figure 1: Eight Neighboring Directions for an Object

## 2 Prior Work

Solutions exist for the  $n$ -queens problem with  $n$  greater or equal to 4 [2]. Empirical observations of smaller sized problems show that the number of solutions increases exponentially with increasing  $n$ .

Numerous closed-form, analytical solutions to the  $n$ -queens problem have been published since 1918 [1, 2, 3, 6, 13, 19]. They basically compose a solution to the  $n$ -queens problem by providing an explicit formula for a queen placement or by patching together solutions to the smaller sized problems. As pointed out by Ahrens [2], such analytical solutions have a limitation in that they generate only a small number of solutions from a restricted subset of solutions. In addition, a problem-solving strategy derived from an analytical solution to the  $n$ -queens problem cannot be used for a general search problem [10, 14]. This is not the case with a search-based algorithm.

A search-based algorithm overcomes the above difficulties and limitations. A backtracking search systematically generates all possible solutions for the  $n$ -queens problem [4, 7, 8, 17, 18]. Here we start with the first column and first row, and assign to it the first queen. Then we select the second queen and assign it to the second column such that no queens attack. We continue with the placement of the next queen to the next column until all queens are placed. If at any point we run out of row positions for a queen to be placed,

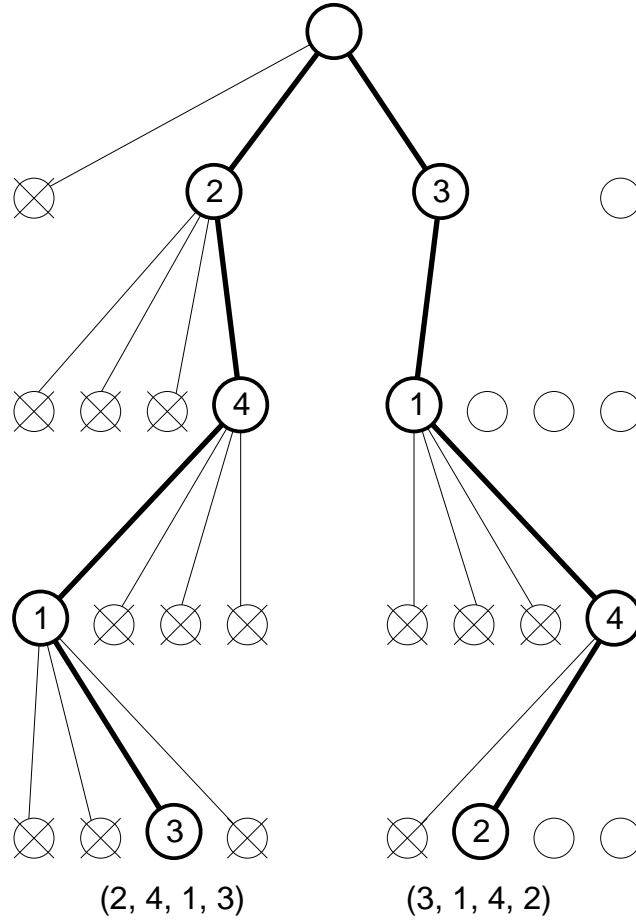


Figure 2: A Partial Search Tree for the 4-Queens Problem

we simply go back one step (backtrack), chose another conflict-free row position on the previous column, and continue the process. If we are able to assign row positions to all the queens then we have found a solution. Otherwise, there are no solutions.

A backtracking search can be represented in a (search) tree representation. As shown in Figure 2, for a 4-queens problem, each quadruple of queens forms a possible path from the root to a leaf node in the search tree. The search tree consists of only 2 leaf nodes (2 solutions). A partial search tree in this case is shown in Figure 2.

In general, a backtracking search is exponential in time. In an empirical study, Stone and Stone used various search heuristics and solved the  $n$ -queens problem for  $n$  up to 96 [23]. Recently Kalé gave a new

backtracking heuristic for the  $n$ -queens problem [15]. His heuristic starts placing queens from the edge inwards for the middle third rows and from the middle position outwards for the first third and the last third rows. The row placement is interleaved with column placement. The order of rows and columns is determined by the number of free positions. Measurements of this heuristic show that it is capable of finding a solution in time proportional to  $O(n^2)$ . Kalé's method is capable of finding all solutions to the  $n$ -queens problem. All solutions may be important for smaller  $n$ -queens problems, but the number of solutions becomes excessive for larger  $n$ . The capability of finding all solutions thus diminishes with increasing  $n$ . In practice, backtracking approaches provide a very limited class of solutions to the large size  $n$ -queens problem. Successive solutions produced by a backtracking search are very similar. For large  $n$ , it is difficult for a backtracking search to find solutions that are significantly distinct in the solution space.

A backtracking search is not able to solve a large size  $n$ -queens problem. It is desirable to investigate some alternative, efficient search approaches for the  $n$ -queens problem. Recently, Sosič and Gu have given several probabilistic local search algorithms that are based on a conflict minimization heuristic [20, 21, 22]. We have compared our local search algorithms with a backtracking search approach and found them significantly faster [9]. The algorithms run in polynomial time or linear time and do not backtrack at all. They are capable of solving a large size  $n$ -queens problem. For example, on a workstation computer, one algorithm can find a solution for 3,000,000 queens in less than 55 seconds [22].

Based on the same idea of using a local search and a conflict minimization heuristic, Minton et al. gave a different algorithm to solve the  $n$ -queens problem [16]. In the algorithm, queens are successively placed on columns such that each new queen is attacked by the minimum number of queens that are already placed. Once all queens are placed, conflicts are resolved by moving conflicting queens, one queen at a time.

### **3 A Linear Time Search Algorithm for the $N$ -Queens Problem**

Two types of collisions may occur among queens, collisions on rows or columns and collisions on diagonals. Collisions on rows and columns can simply be avoided by a permutation operation.

```

1.  procedure Queen_Search (queen : array [1..n] of integer)
2.  var
3.      k : integer;
4.  begin
5.      k := Initial_Search(queen);
6.      Final_Search(k,queen);
7.  end;

```

Figure 3: A Linear Time  $N$ -Queens Search Algorithm

Let  $queen_i$  denote the row number of the queen in the  $i^{th}$  column. For the  $n$ -queens problem, there are  $n$  queens with row positions  $queen_1, \dots, queen_n$ . Since each column has exactly one queen, no two queens will attack each other on the same column. If numbers  $queen_1, \dots, queen_n$  form a permutation of integers  $1, \dots, n$ , then each row is occupied by exactly one queen. A permutation of integers  $1, \dots, n$  thus denotes a placement of queens where no two queens 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 diagonal lines.

A linear time search algorithm for the  $n$ -queens problem is shown in Figure 3. It consists of an initial search phase that starts during problem generation and a final search phase that removes conflicts. During the initial search (function *Initial\_Search*), an initial permutation of the row positions of the queens is generated. This permutation produces a small number of collisions among queens. These collisions are removed by the final search procedure. During the final search (procedure *Final\_Search*), two queens are chosen for conflict minimization. If a swap of two queens reduces the number of collisions then two queens are swapped. Otherwise, no action is taken. The final search is repeated until all collisions are eliminated, that is, till a solution is found. The details of the initial and final search phases are described in subsequent sections.

### 3.1 The Initial Search

The initial search produces a permutation of numbers  $1, \dots, n$  in variables  $queen_1, \dots, queen_n$ . This initial placement of row positions generally produces collisions among queens on the diagonal lines. The number of collisions on a diagonal line is one less than the number of queens on the line. Exceptions are empty

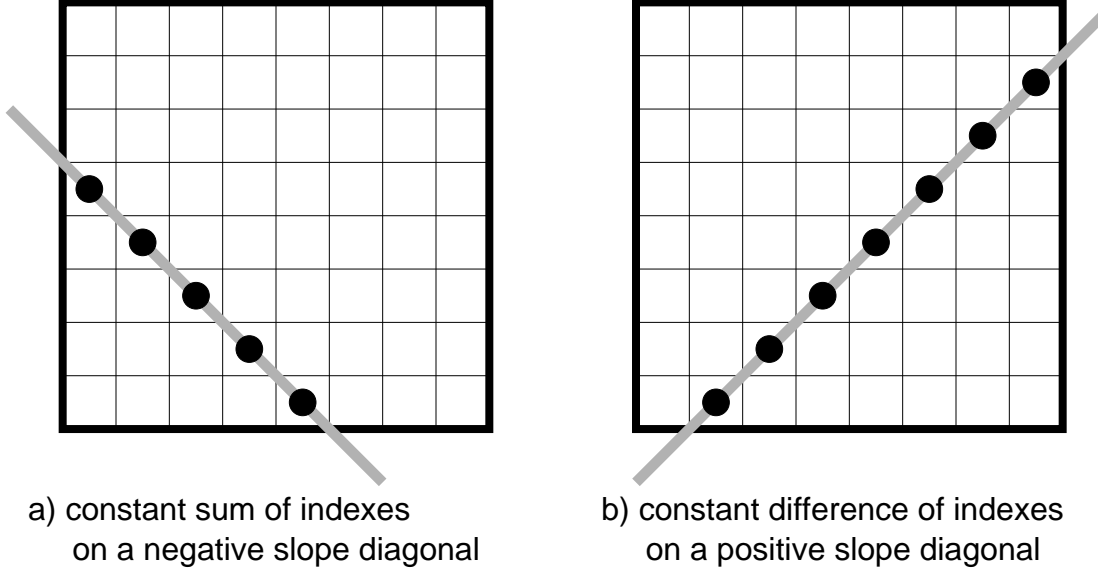


Figure 4: The Characterization of Diagonals

diagonal lines where there are no collisions. The sum of collisions on all diagonal lines is the total number of collisions among queens.

The number of collisions on a diagonal line can be computed in constant time using a characterization of diagonal lines. Let  $i$  be a *row index* and  $j$  be a *column index*, then  $i + j$  is constant on any diagonal line with a negative slope, and  $i - j$  is constant on any diagonal line with a positive slope (see Figure 4). For  $queen_i$ , its diagonal lines are calculated as  $i + queen_i$  and  $i - queen_i$ . For the  $n$ -queens problem, an array of size  $2n - 1$  keeps the number of queens on each of the  $2n - 1$  negative diagonal lines. Similarly, another array of size  $2n - 1$  keeps the number of queens on each of the  $2n - 1$  positive diagonal lines. Using these two arrays, it takes constant time to find the number of collisions on a diagonal line.

One way to produce an initial placement is to generate a random permutation of  $n$  numbers. It was observed that a random permutation of  $n$  numbers tends to generate  $0.5285n$  collisions among queens as  $n$  approaches infinity [20, 21]. A random permutation of 1,000,000 queens would generate approximately 528,500 collisions. Since this number of collisions is very high, the algorithm performance can be improved significantly if collisions in the initial permutation are minimized.

```

1.  function Initial_Search (queen : array [1..n] of integer)
2.  var
3.      i, j, m : integer;
4.  begin
5.      for i := 1 to n do queen[i] := i;
6.      j := 1;
7.          (* place queens without collisions *)
8.      for i := 1 to 3.08 * n do begin
9.          m := Random(j, n);
10.         Swap(j, m);
11.         if Partial_Collisions(j) = 0 then j := j + 1;
12.         else Swap(j, m);
13.     end;
14.         (* place queens with possible collisions *)
15.     for i := j to n do begin
16.         m := Random(i, n);
17.         Swap(i, m);
18.     end;
19.         (* return the number of queens with possible collisions *)
20.     Initial_Search := n - j + 1;
21. end;

```

Figure 5: The Initial Search

The minimization of collisions is done by function *Initial\_Search* (see Figure 5). There are some auxiliary functions and procedures. Function *Random(x, n)* returns a random integer between *x* and *n*. Procedure *Swap(i, j)* performs a swap of *queen<sub>i</sub>* and *queen<sub>j</sub>* that were generated during initialization. Function *Partial\_Collisions(i)* returns the number of collisions on diagonals to the left that pass through position (*i, queen[i]*). This function gives the number of collisions with queens in columns smaller than *i*. Function *Total\_Collisions(i)* returns the total number of collisions on diagonals that pass through position (*i, queen[i]*).

Queens are placed on successive columns from left to right. The position for each new queen is *randomly* generated from rows that are not occupied by queens in columns to the left. If the new queen is attacked by the queen in a column to the left, then a new random position is generated. New random positions are generated until a conflict free place is found for the queen. Then the algorithm places the queen in the next column.

After the random number generator is called a specified number of times, the remaining queens are placed



randomly on empty rows in columns on the right regardless of conflicts on diagonal lines. The number of queens on the right with possible collisions is returned as the value of function *Initial\_Search*.

For a problem size  $n$ ,  $3.08n$  calls to the random number generator are performed, not counting the placement of queens with collisions on the right. Each call to the random number generator represents one attempt to place a queen. Number 3.08 has been chosen based on a mathematical analysis that is presented in Section 4. Experimental measurements have confirmed such a decision.

Since  $3.08n$  steps are performed, the process of initial search requires a linear time. Experimental results and the analysis in Section 4 show that the number of queens with a possible conflict produced by the initial search can be treated as a constant or as almost a constant for all  $n$ .

## 3.2 The Final Search

During the initial search, some queens produce collisions on diagonal lines. Assume there are  $k$  such queens. A simple local search technique is used during the final search to resolve the positions of these  $k$  queens on the right.

### 1. Conflict Minimization

A conflict minimization heuristic navigates the search activity (see Figure 6). Two queens are chosen for a possible swap. The first queen is one of the queens with collisions, the second queen is chosen at *random*. If the swap of row positions of a pair of queens produces no collisions, then the swap is performed. This conflict minimization heuristic is applied until there are no collisions left, that is, till a solution is found.

### 2. Termination Criteria

If no solution could be found after a certain number of search steps, a new initial permutation is generated and a new search process is started. We have limited the number of search steps to 7000. This number has been chosen based on mathematical analysis and empirical observations (see Section 4.2). We have executed the algorithm 100 times for each  $n$  between 4 and 1000. Table 1 shows how many initial permutations were needed to obtain 100 solutions for a given  $n$ . For  $n > 400$ , the algorithm almost always finds a solution from

```

1.  procedure Final_Search ( k : integer, queen : array [1..n] of integer)
2.  var
3.      i, j : integer;
4.      b : boolean;
5.  begin
6.      for i := n - k + 1 to n do begin
7.          if Total_Collisions(i) > 0 then
8.              repeat
9.                  j = Random(1,n);
10.                 Swap(i, j);
11.                 b := (Total_Collisions(i) > 0) or (Total_Collisions(j) > 0);
12.                 if b then Swap(i, j);
13.             until not b;
14.      end;
15. end;

```

Figure 6: The Final Search

Table 1: The Number of Initial Permutations to Find 100 Solutions

Number of Queens <i>n</i>	200	300	400	500	600	700	800	900
Number of Initial Permutations	168	127	106	105	101	100	102	101

only one initial permutation.

For  $n < 200$ , the random method of choosing a pair of queens to swap does not perform very well, so a different approach has been chosen. Each of the  $k$  queens with possible collisions is systematically tested with all other queens for a possible swap. If all possible pairs are tested without a successful swap, a new initial permutation is generated and the search is repeated. For  $n$  close to 200, this method needs approximately four initial permutations to find a solution.

The running time of the algorithm can be estimated as follows. The initial search can be done in linear time as shown in Section 3.1 and Section 4. Analysis in Section 4 shows that the number of steps in the final search is constant and independent of  $n$ . This has been verified by experimental results. Taking together the initial and final search phases, it follows that the algorithm has a linear running time. This is confirmed by experimental data presented in Section 5. Using different random generator seeds, the algorithm produces different solutions, but its execution time remains fairly stable. Very robust behavior is another advantage

of this probabilistic local search algorithm.

## 4 An Analysis of the Algorithm

This section presents a mathematical analysis of the algorithm. The analysis gives an estimated number of computing steps for the initial search phase and the final search phase which shows a linear time complexity of the algorithm.

Following a common mathematical technique, we first transform the problem from a discrete domain to a continuous domain. The original chessboard is mapped to a unit square. A position  $(i, j)$  on the original  $n \times n$  board is mapped to a point  $(\frac{i}{n}, \frac{j}{n})$  in the unit square. This simplifies the analysis of the problem as  $n$  approaches infinity without affecting the results.

We assume that the queens are uniformly distributed on the board such that no two queens attack each other. In each column there is exactly one queen. Thus, the probability that a given position is occupied by a queen is  $\frac{1}{n}$ . If a diagonal spans over  $k$  positions on the chessboard, we assume that the diagonal is occupied by a queen with a probability of  $\frac{k}{n}$ . We have compared this predicted probability to measured probabilities for  $n$  equal to 10,000. The average values of 10,000 measurements are shown in Figure 7. It can be seen that predicted probabilities are close to measured probabilities. When we map the board to a unit square, a diagonal line from point  $(x_0, y_0)$  to point  $(x_1, y_1)$  covers  $|x_1 - x_0|$  positions and it is occupied by a queen with probability  $|x_1 - x_0|$ .

Let  $a$  and  $b$  be two diagonal lines that cross each other. Let  $p(a)$  represent the probability that there is a queen on diagonal line  $a$  and let  $p(b)$  represent the probability that there is a queen on diagonal line  $b$ . The combined probability  $p(a + b)$  that a common position is attacked from diagonal  $a$  or diagonal  $b$  is computed as the probability of the sum  $a$  and  $b$ :

$$p(a + b) = p(a) + p(b) - p(a)p(b). \quad (1)$$

This is a well known formula for the probability of the sum.

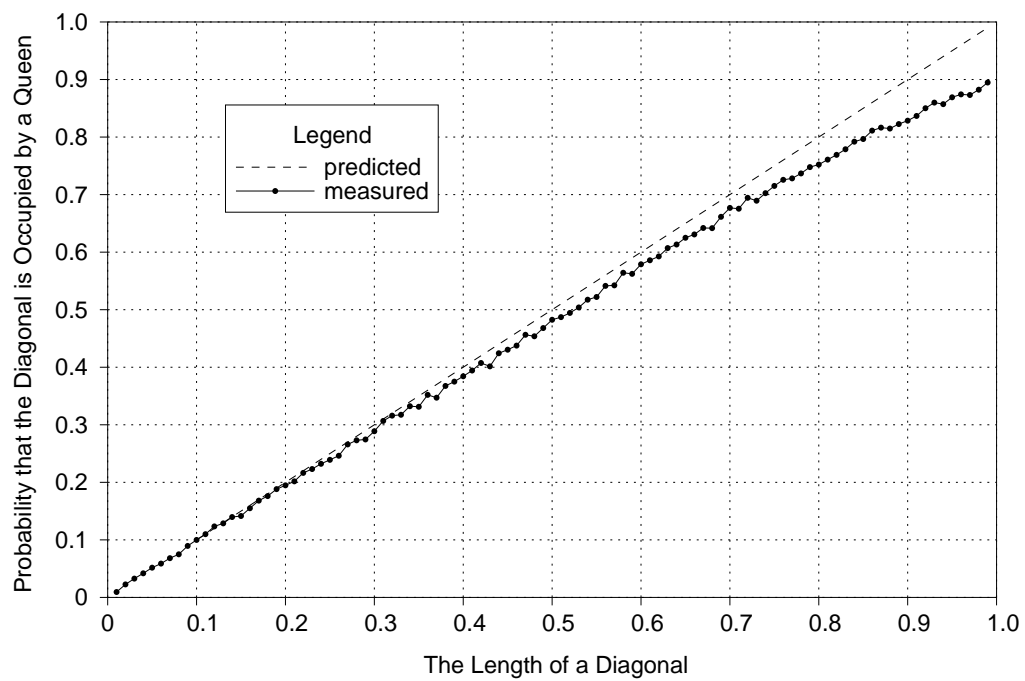


Figure 7: Predicted and Measured Probabilities That There is a Queen on a Diagonal of Length  $\frac{k}{n}$ . The Average of 10,000 Samples for  $n$  Equal to 10,000.

## 4.1 An Analysis of the Initial Search

During the initial search, all queens are placed in successive columns, starting with the column on the left.

We seek an estimated number of steps during the initial search.

Let  $f(x)$  denote the number of steps to place the first  $x$  queens. The value of  $f(1)$  is the total number of steps during the initial search. Let  $p(x)$  be the probability that a random queen on column  $x$  is attacked by some previously placed queen to the left. The value of  $f(x)$  can be computed from  $p(x)$ :

$$f(x) = \int_0^x \frac{1}{1 - p(x)} dx. \quad (2)$$

We split the calculation of  $p(x)$  into two parts, the interval with  $0 < x < 0.5$  and the interval with  $0.5 < x < 1$ . The probabilities of a collision in these two intervals are denoted as  $p_1(x) = p(x; x < 0.5)$  and  $p_2(x) = p(x; x > 0.5)$ , respectively. The algorithm prevents the construction of row and column collisions, so we take into account only diagonal collisions.

First, we calculate  $p_1(x)$ . Let  $x$  denote a column between 0 and 0.5 and let  $y$  be a point on this column. Let  $a$  and  $b$  be two left diagonal lines that pass through point  $(x, y)$ . It follows from Equation (1) that point  $(x, y)$  is occupied with probability:

$$p(x, y) = p(a) + p(b) - p(a)p(b).$$

We separate the values of  $y$  in three intervals: (a)  $x < y < 1 - x$ , (b)  $y < x$ , and (c)  $y > 1 - x$ :

**Case (a)** if  $x < y < 1 - x$ , then  $a$  and  $b$  have equal length (see Figure 8 (a)). In this case,  $p(a) = x$  and  $p(b) = x$ . It follows that point  $(x, y)$  is attacked by a queen with the probability:

$$p(x, y) = p(a + b) = x + x - x^2 = 2x - x^2.$$

**Case (b)** if  $y < x$ , then  $p(a) = y$  and  $p(b) = x$  (see Figure 8 (b)). Point  $(x, y)$  is attacked by a queen with the probability:

$$p(x, y) = p(a + b) = y + x - yx.$$

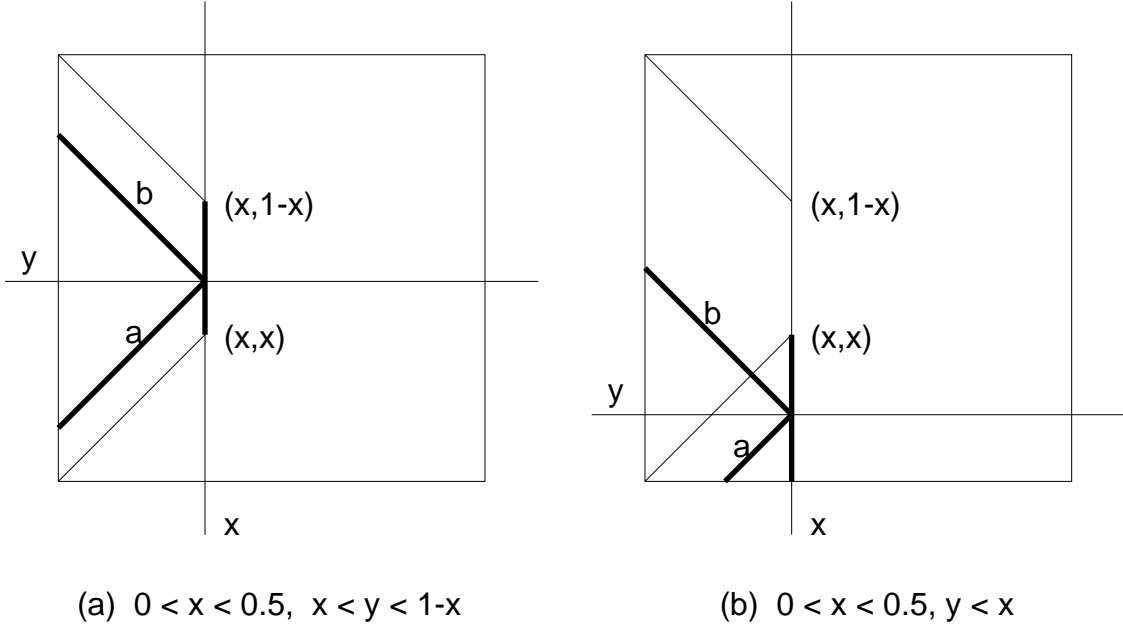


Figure 8: Left Diagonals through a Column between 0 and 0.5

**Case (c)** if  $y > 1 - x$ , the probability is equivalent to the case of  $y < x$ .

To compute the probability  $p_1(x)$  that a random position in column  $x$  is attacked by a previously placed queen, we need to find an average over all  $y$  between 0 and 1. This average is computed by summing probabilities over all  $y$ :

$$p_1(x) = 2 \int_0^x y + x - yx \, dy + \int_x^{1-x} 2x - x^2 \, dy = 2x - 2x^2 + x^3. \quad (3)$$

To calculate  $p_2(x)$ , let  $x$  denote a column between 0.5 and 1 and let  $y$  be a point on this column. Let  $a$  and  $b$  be two left diagonal lines that pass through point  $(x, y)$ . We separate the values of  $y$  in three intervals:

(a)  $1 - x < y < x$ , (b)  $y < 1 - x$ , and (c)  $y > x$ :

**Case (a)** if  $1 - x < y < x$ , then  $p(a) = y$  and  $p(b) = 1 - y$  (see Figure 9 (a)). Point  $p(x, y)$  is attacked by a queen with the probability:

$$p(x, y) = p(a + b) = 1 - y + y - (1 - y)y = 1 - y + y^2.$$

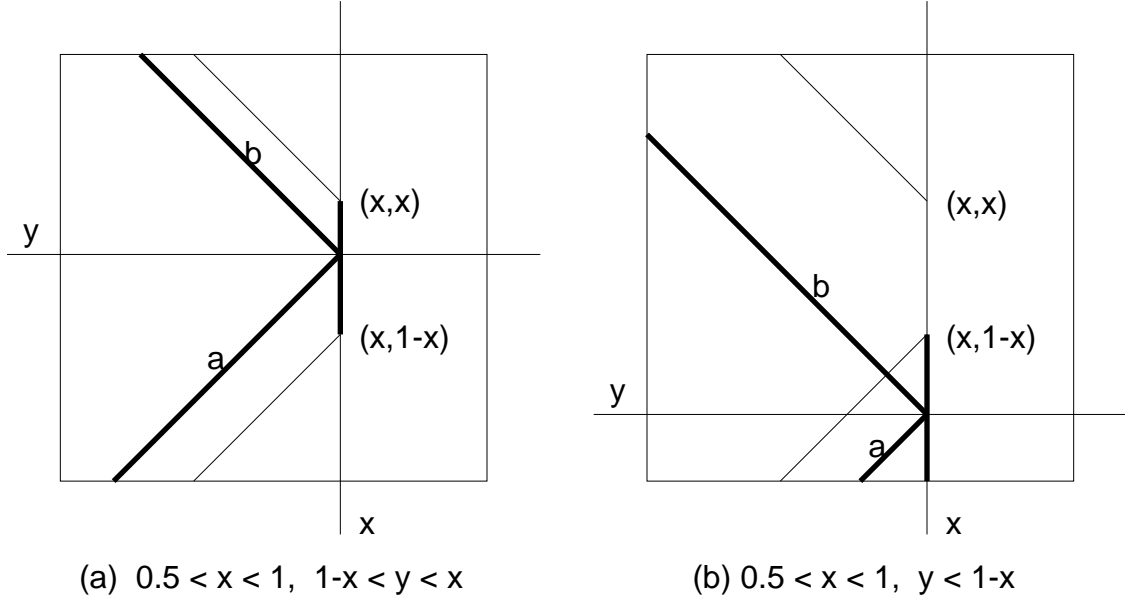


Figure 9: Left Diagonals through a Column between 0.5 and 1

**Case (b)** if  $y < 1 - x$ , then  $p(a) = x$  and  $p(b) = y$  (see Figure 9 (b)). The probability  $p(x, y)$  is:

$$p(x, y) = p(a + b) = x + y - xy.$$

**Case (c)** if  $y > x$ , the probability is equivalent to the case of  $y < 1 - x$ .

The probability  $p_2(x)$  that a random position in column  $x$  is attacked by a queen from left is computed by the following integral:

$$p_2(x) = 2 \int_0^{1-x} x + y - xy \, dy + \int_{1-x}^x 1 - y + y^2 \, dy = \frac{1}{6} + x - \frac{x^3}{3}. \quad (4)$$

Using  $p_1(x)$  from Equation (3) and  $p_2(x)$  from Equation (4), we can calculate the number of steps  $f(x)$  to place  $x$  queens (see Equation (2)). If  $x$  is less than 0.5 then:

$$f(x) = \int_0^x \frac{1}{1 - p_1(x)} \, dx.$$

If  $x$  is greater than 0.5 then:

$$f(x) = \int_0^{0.5} \frac{1}{1 - p_1(x)} \, dx + \int_{0.5}^x \frac{1}{1 - p_2(x)} \, dx.$$

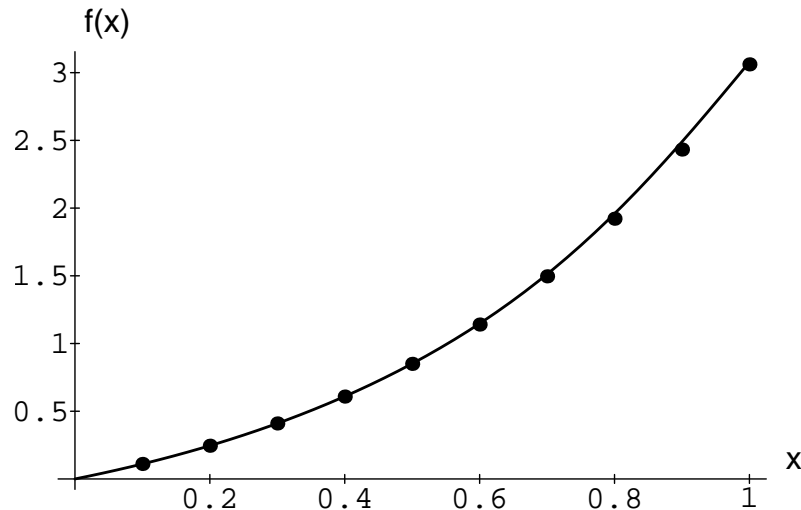


Figure 10: The Initial Search: Predicted Performance (Curve) and Measured Performance (Points)

Figure 10 shows the comparison between our model and experimental measurements of the real algorithm execution. The curve in the figure represents function  $f(x)$ . The dots represent the measured number of steps to place  $x$  queens in 0.1 increments of  $x$ . Points are an average of 100 execution runs taken for the problem with  $n$  equal to 100,000.

The number of steps required to place all queens without collisions is equal to  $f(1) = 3.08$ . That means that, for problem size  $n$ , the initial search will attempt on average  $3.08n$  positions before all queens are placed. The initial search takes time that is linear in  $n$ .

Although our model corresponds very closely to the real performance of the algorithm, the assumption at the beginning of this section that all queens can be placed randomly on each column is not accurate for the last few queens placed. Since most rows are already occupied, the number of choices for the queens on the right is very restricted. We have measured the number of queens that are placed by the initial search without collisions, using  $3.08n$  steps. Table 2 shows the average number, the minimum number, and the maximum number of queens, placed during 10 runs. These experimental results demonstrate that the initial search can place almost all queens without collisions even for very large values of  $n$ . A similar observation was made for the method described in [16].



Table 2: The Number of Queens, Placed during the Initial Search, Using  $3.08n$  Steps. (10 Runs)

Number of Queens $n$	100	1,000	10,000	100,000	1,000,000
Average	91	979	9969	99977	999975
Minimum	84	959	9919	99964	999946
Maximum	97	990	9990	99988	999987

## 4.2 An Analysis of the Final Search

During one iteration of the final search, an attacked queen on the right is swapped with a random queen, so that neither of these queens is under attack after the swap. We want to calculate the number of attempts,  $A$ , until a successful swap is performed. This number multiplied by the number of queens under an attack gives an estimate of the total number of steps during the final search.

We assume that queens are randomly distributed with one queen per column. Let  $P$  be the probability that a random position on the board is diagonally attacked by some queen and let  $F$  be the probability that a position in the column on the right is diagonally attacked by some queen. Given  $P$  and  $F$ , the expected number of attempts  $A$  for a successful swap is calculated as:

$$A = \frac{1}{(1 - F)(1 - P)}. \quad (5)$$

From Equation (4), the value of  $F$ , the probability that a position in the column on the right is diagonally attacked by some queen, is equal to  $p_2(1) = \frac{5}{6}$ .

To calculate  $P$ , because of the eightfold symmetry of a square, we restrict ourselves to the lower left region  $0 < x < 0.5$  and  $0 < y < x$ . If  $p(x, y)$  is the probability that a random point in this region is diagonally attacked by a queen, the value of  $P$  is equal to:

$$P = 8 \int_0^{0.5} \int_0^x p(x, y) dy dx. \quad (6)$$

Taking a point  $p(x, y)$  in that region (see Figure 11), there are four diagonal segments passing through it:  $a_1$ ,  $a_2$ ,  $b_1$ , and  $b_2$ . The probabilities that  $p(x, y)$  is attacked from diagonal segments are as follows:

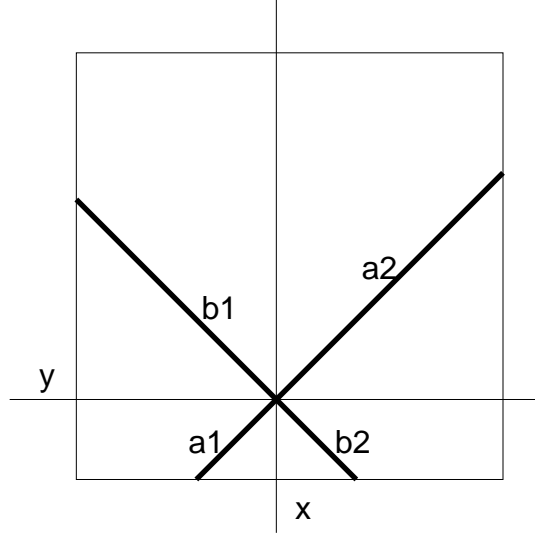


Figure 11: Diagonals through Random Point (x,y)

$$p(a_1) = y,$$

$$p(a_2) = 1 - x,$$

$$p(b_1) = x,$$

$$p(b_2) = y.$$

Segments  $a_1$  and  $a_2$  are parts of the same diagonal  $a$ . Since each diagonal contains only one queen, Equation (1) does not apply in this case. Probability  $p(a)$  is a sum of  $p(a_1)$  and  $p(a_2)$ :

$$p(a) = p(a_1) + p(a_2) = 1 - x + y.$$

A similar formula is valid for segments  $b_1$  and  $b_2$  on diagonal  $b$ :

$$p(b) = p(b_1) + p(b_2) = x + y.$$

Using Equation (1), probability  $p(x, y)$  is computed as  $p(a + b)$ :

$$p(x, y) = p(a + b) = p(a) + p(b) - p(a)p(b) = 1 - x + y + x + y - (1 - x + y)(x + y) = 1 - x + x^2 + y - y^2.$$

Substituting  $p(x, y)$  in Equation (6), we obtain the value of  $P$ :

$$P = 8 \int_0^{0.5} \int_0^x 1 - x + x^2 + y - y^2 \, dy dx = \frac{11}{12}.$$

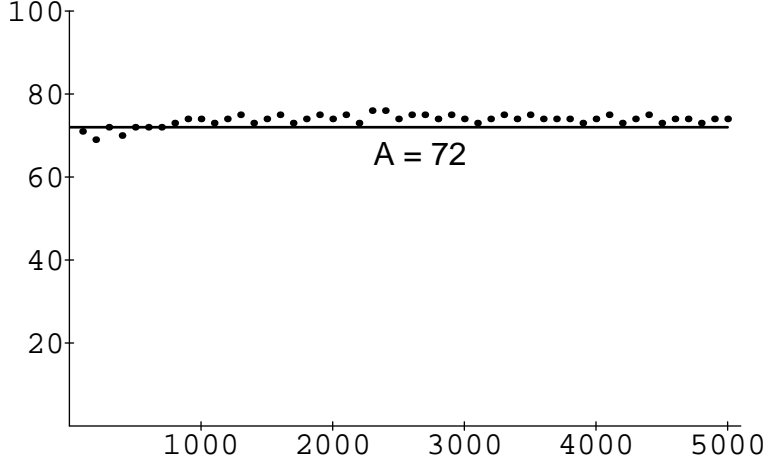


Figure 12: The Final Search: Predicted Performance (Curve) and Measured Performance (Points) (The Average of 100 Runs);  $x$ -axis represents the number of queens with collisions,  $y$ -axis represents the average number of swap attempts per queen.

The number of attempts,  $A$ , to successfully swap a queen in the final search can be calculated from Equation (5):

$$A = \frac{1}{(1-F)(1-P)} = \frac{1}{\frac{1}{6} \frac{1}{12}} = 72.$$

We have experimentally measured the number of swap attempts,  $A$ , for  $n$  equal to 100,000. A large number of experiments have been performed where the number of queens with collisions varied from 100 to 5000. Figure 12 shows the measured number of swap attempts for each successful swap, averaged over 100 runs. The measured average value is between 69 and 76 which corresponds closely to the predicted value of 72.

Using a swap between a pair of queens, collisions for one queen are resolved in a constant time of approximately 72 steps regardless of  $n$ . This constant time compares to a linear time in the method described in [16], where only one queen is moved in a single step. Assuming that the initial search produces, for any  $n$ , a constant number of queens with collisions, the final search takes constant time. In our algorithm, the limit for the number of search steps during the final search has been set to 7000. The actual average number of search steps is much smaller. For example, for  $n$  equal to 1,000,000, the average number of steps in the final search in 100 runs was measured to be 1448.

Table 3: The Execution Time of the Linear Time Search Algorithm on an IBM RS 6000/530 Computer (Average of 10 Runs; Time Units: seconds)

Number of Queens $n$	$10^4$	$10^5$	$10^6$	$2 \times 10^6$	$3 \times 10^6$
Time of the 1st run	0.1	1.1	16.9	35.7	54.8
Time of the 2nd run	0.1	1.1	17.0	35.8	54.8
Time of the 3rd run	0.1	1.1	17.1	35.8	54.6
Time of the 4th run	0.1	1.1	17.0	35.9	54.8
Time of the 5th run	0.1	1.1	17.0	35.8	54.7
Time of the 6th run	0.1	1.1	17.0	35.8	54.6
Time of the 7th run	0.1	1.1	17.0	35.8	54.7
Time of the 8th run	0.1	1.1	17.0	35.8	54.7
Time of the 9th run	0.1	1.1	17.0	35.8	54.7
Time of the 10th run	0.1	1.1	17.0	35.8	54.7
Avg. Time to Find a Solution	0.1	1.1	17.0	35.8	54.7

A mathematical analysis and measured performance of our algorithm demonstrate its linear running time. Most of the time is spent in the initial search which executes in a linear running time. The final search requires a constant time independent of  $n$ .

As is common with probabilistic algorithms [5], our algorithm to solve the  $n$ -queens problem does not guarantee worst case running time for a particular instance of the algorithm execution, but it exhibits excellent performance and a very robust behavior. The analysis puts a bound on the number of expected execution steps. Hence an unsuccessful attempt may be halted and the algorithm restarted from the beginning.

## 5 Real Execution Time of the Algorithm

The real execution time of our linear time search algorithm, that was programmed in C and run on an IBM RS 6000/530 computer, is illustrated in Table 3. For each  $n$  presented, we made 10 execution runs with different random numbers. These runs show that the algorithm exhibits very stable behavior for large  $n$ . Due to memory size limitation, the largest  $n$  we were able to run was 3,000,000. Since the algorithm was able to find a solution in less than 0.1 seconds for  $n < 10000$ , these results are not shown in the table. Approximate time to place one queen for these small values of  $n$  was around 15 microseconds.

## 6 Applications of the $N$ -Queens Problem

Since each solution to the  $n$ -queens problem forms a non-conflict pattern, the  $n$ -queens problem has many practical scientific and engineering applications. Before we conclude this paper, we give a representative example of its applications. Although this particular example requires solutions to small  $n$ -queens problems, it demonstrates a general idea behind applications.

To achieve high communication bandwidth in a narrow-banded, directional communication system, an array of  $n$  transmitters/receivers must be placed without any interference with each other. With  $n$  transmitters/receivers placed in a non-conflict pattern, which corresponds to a solution to the  $n$ -queens problem, each transmitter/receiver can communicate with the outside world freely in eight directions (i.e., two horizontal directions, two vertical directions, and four diagonal directions) without being obscured by other transmitters/receivers. Figure 13 shows one such placement of ten transmitters/receivers. This placement follows from one of the solutions to the 10-queens problem.

## 7 Conclusion

A linear time local search algorithm with a conflict minimization heuristic is presented. This algorithm is significantly faster than any backtracking search algorithm and is capable of solving the  $n$ -queens problem in linear time. The performance is achieved through the application of a general conflict minimization heuristic within a general local search framework. The algorithm demonstrates that a guided initial search can significantly reduce later search efforts.

With its general conflict minimization heuristic and its general local search framework, the ideas behind this algorithm have been applied to similar large-scale constraint-based optimization problems. This efficient algorithm has made many difficult scientific and engineering applications possible.

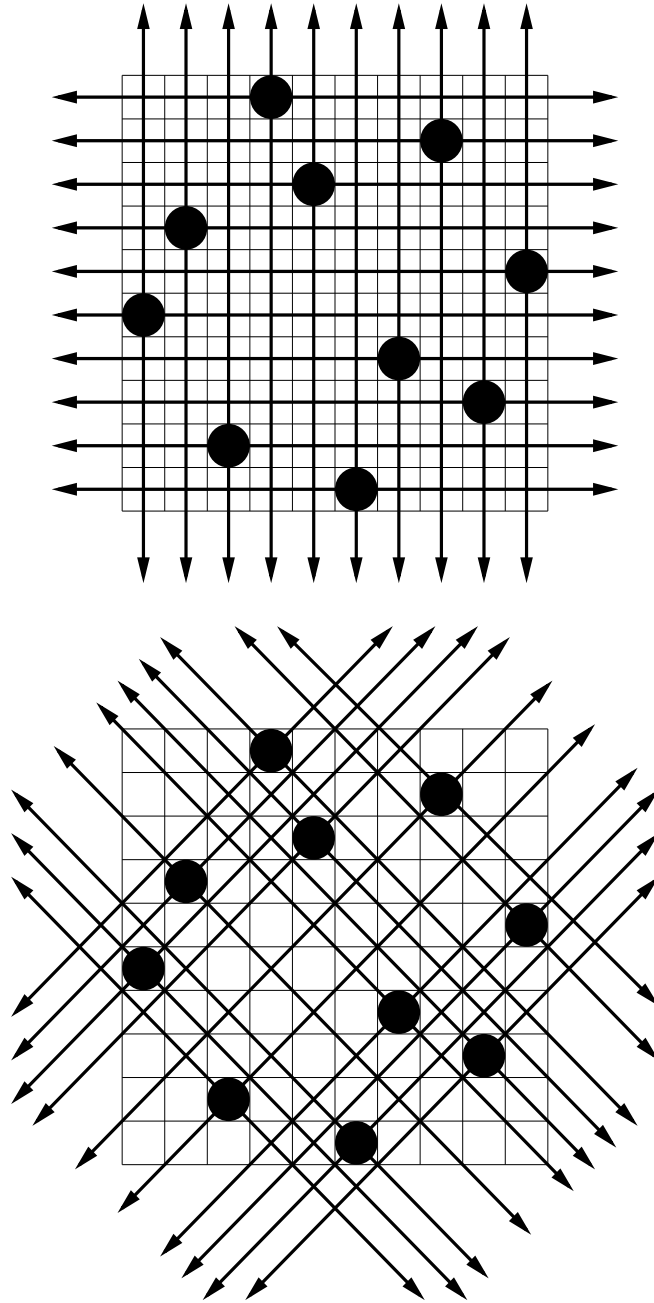


Figure 13: A Non-conflict Placement of Ten Transmitters/Receivers (Each of them can communicate in any vertical, horizontal, and diagonal direction)

## 8 Acknowledgements

Vladimir Batagelj, Tomaž Pisanski, Robert Johnson, Lewis Johnson, Benjamin Wah and Vipin Kumar provided many constructive comments during the preparation of this article. We also thank anonymous reviewers whose comments helped improve the quality of this paper.

## References

- [1] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *Journal of Parallel and Distributed Computing*, 6:649–662, 1989.
- [2] W. Ahrens. *Mathematische Unterhaltungen und Spiele (in German)*. B.G. Teubner (Publishing Company), Leipzig, 1918-1921.
- [3] B. Bernhardsson. Explicit solutions to the n-queens problems for all n. *ACM SIGART Bulletin*, 2(2):7, Apr. 1991, ACM Press.
- [4] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. ACM*, 18(11):651–656, Nov. 1975.
- [5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.
- [6] B.-J. Falkowski and L. Schmitz. A note on the queens' problem. *Information Processing Letters*, Vol. 23:39–46, July 1986.
- [7] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of 12th Annual Allerton Conf. Circuit System Theory*, 1974.
- [8] J. Gaschnig. *Performance Measurements and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, May 1979.
- [9] J. Gu. Parallel algorithms and architectures for very fast AI search (Ph.D thesis). Technical Report UUCS-TR-88-005, Univ. of Utah, Dept. of Computer Science, Jul. 1988.
- [10] J. Gu. On a general framework for large-scale constraint-based optimization. *SIGART Bulletin*, 2(2):8, Apr. 1991, ACM Press.
- [11] J. Gu. *Constraint-Based Search*. Cambridge University Press, New York, 1992.



- [12] R. M. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
- [13] E. J. Hoffman, J. C. Loessi, and R. C. Moore. Constructions for the solution of the  $m$  queens problem. *Mathematics Magazine*, pages 66–72, 1969.
- [14] Lewis Johnson. Editor letter. *SIGART Bulletin*, Oct. 1990 to Oct. 1991, ACM Press.
- [15] L. V. Kalé. An almost perfect heuristic for the  $n$  nonattacking queens problem. *Information Processing Letters*, Vol. 34:173–178, April 1990.
- [16] 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.
- [17] P. W. Purdom and C. A. Brown. An analysis of backtracking with search rearrangement. *SIAM J. Comput.*, 12(4):717–733, Nov. 1983.
- [18] P. W. Purdom, C. A. Brown, and E. L. Robertson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–113, 1981.
- [19] M. Reichling. A simplified solution of the  $n$  queens’ problem. *Information Processing Letters*, Vol. 25:253–255, June 1987.
- [20] R. Sosič and J. Gu. How to search for million queens. Technical Report UUCS-TR-88-008, Dept. of Computer Science, Univ. of Utah, Feb. 1988.
- [21] R. Sosič and J. Gu. A polynomial time algorithm for the  $n$ -queens problem. *SIGART Bulletin*, 1(3):7–11, Oct. 1990.
- [22] R. Sosič and J. Gu. 3,000,000 queens in less than a minute. *SIGART Bulletin*, 2(2):22–24, Apr. 1991, ACM Press.

- [23] 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.