Solving the n-Queens Problem Using Genetic Algorithms

Kelly D. Crawford
Tulsa University

Abstract

This paper shows two ways that genetic algorithms can be used to solve the n-queens problem for large values of n. The first uses simple heuristics for evaluating potential solutions. The second uses existing evaluation heuristics by first mapping the n-queens problem onto satisfiability which has already been solved using genetic algorithms. Results for each method are shown and some conclusions are drawn about solving NP problems with genetic algorithms.

1 Introduction

Understanding this problem requires some introductory knowledge about genetic algorithms, NP problems, the n-queens problem, and the satisfiability problem. Section 1 introduces these topics, while section 2 shows how they can be used together. Section 3 shows some test results and section 4 lists some conclusions drawn based on this work.

1.1 Genetic Algorithms

A genetic algorithm is a search technique based on various biological principles (see [2, 3, 5] for more details). Two things are required to be able to solve a problem with genetic algorithms. The first is a representation, and the second is an evaluation function.

A representation is how a potential solution to the problem being solved is encoded. Often it is a string of bits (which of course can be used to represent anything), but it can also be a list of numbers, sets of rules, etc.

An evaluation function is used to tell how good a particular solution is. The evaluation function must first understand a solution's representation, decode it if necessary, and run some sort of computation on it to come up with a relative worth. Genetic algorithms test thousands of solutions in this manner, with the objective being to minimize (or maximize) the result.

A typical genetic algorithm is structured as follows:

(1) A set of possible solutions is generated (randomly or otherwise). This is called the solution pool. This pool will hold the top p solutions found (according to their assigned value from the evaluation function) in sorted order. The best solution found so far will be at the top of the pool. Pool sizes vary greatly with the application.

(2) Two parent solutions are selected from the pool for use in crossover according to some selection bias. This bias tells how much more likely it is that the parents will be selected from near the top of the pool. There are several types of crossover operators, but one of the more common ones is called two-point crossover [2]. Two points are randomly selected in a solution and the genetic material (e.g. bits) between those points is swapped

between the two parents to generate two new child solutions. The following diagram illustrates this.

```
1 0 0 | 1 0 1 | 1 0
1 0 1 | 0 1 1 | 0 0
      becomes
1 0 0 | 0 1 1 | 1 0
1 0 1 | 1 0 1 | 0 0
```

(3) A mutation rate μ tells how often to mutate a child solution generated by crossover. This rate is usually low (e.g. $\mu = .01$ says to mutate 1% of all new child solutions). A simple mutation operator is to randomly flip a bit within the solution.

(4) Finally, the new child solutions are evaluated and inserted into the solution pool. A constant number of solutions are kept in the pool, so the worst solutions are removed when the new child solutions are better. The process continues at step (2) until a fixed number of iterations have been done, until the solution stops improving noticeably, or until some other heuristic has been satisfied.

Through this repeated selection, crossover, mutation, and insertion process, the solution pool tends to become more and more fit (i.e. on the average, the solutions in the pool are improving). If successful, after many iterations, near-optimal to optimal solutions will be found at the top of the solution pool.

1.2 NP Problems

A class of problems has been defined where no deterministic solutions exist that run in polynomial time. Problems solvable by a deterministic algorithm in polynomial time are said to be in class P. Those that can only be solved in polynomial time by nondeterministic algorithms are said to be in class NP [1, 6, 8]. Problems in class NP are difficult (if not impossible) to solve in a realistic timeframe because of their large running order (often $\bigcirc(2^n)$ or $\bigcirc(n!)$.)

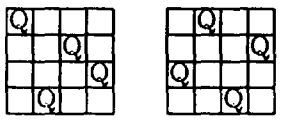

Figure 1: 4-tuple representations

Various approximation methods have been developed to solve some of these problems, but none of them promise optimal solutions. The genetic algorithm, with its ability to cover large search spaces effectively, is a useful approximation method for NP problems. Two problems that fall into class NP are the n-queens problem and satisfiability. Each are defined below.

1.3 n-Queens Problem

In the game of chess, the queen is the most powerful game piece. It has the ability to move up, down, left, right, and diagonally any number of spaces (i.e. any direction in a straight line.) The n-queens problem asks how to place n queens on an n-by-n chessboard (literally, a square grid) such that no queen can attack any other queen on the board. What makes this problem difficult is that there are $\binom{n^2}{n}$ possible solutions!

Most of these possibilities can be eliminated by using an n-tuple to represent a potential solution. For example, consider the two diagrams shown in figure 1 and their associated 4-tuples for the 4-queens problem.

Since positions in the 4-tuple represent columns, column conflicts are eliminated. Since each element of the 4-tuple is distinct, row conflicts are eliminated as well. This

brings the total number of possible solutions down to n!. Unfortunately, this is still impossible to solve for very large n in any reasonable amount of time.

1.4 Satisfiability

Given a boolean function of n inputs (or variables), can a set of inputs be found that will make the function evaluate to true? This is the problem posed by satisfiability. All of the inputs are boolean, so the total search space is 2^n, again falling into NP. In fact, satisfiability is known as the basis for NP theory because all problems falling into class NP can be mapped onto it [6].

DeJong and Spears have already used genetic algorithms to solve this problem [7]. The representation is a string of bits which represent the variables of the boolean function. Given this simple representation, they developed a set of heuristics to use in the evaluation function. To understand these heuristics, first consider the translation of boolean logic values to integer values.

$$\begin{aligned} \text{val(true)} &= 1 \\ \text{val(false)} &= 0 \\ \text{val(NOT } e) &= 1 - \text{val}(e) \\ \text{val(AND } e_1 \ \ldots \ e_n) &= \text{MIN}(\text{val}(e_1) \ \ldots \ \text{val}(e_n)) \\ \text{val(OR } e_1 \ \ldots \ e_n) &= \text{MAX}(\text{val}(e_1) \ \ldots \ \text{val}(e_n)) \end{aligned}$$

These val functions convert logical to integer, but still only give 0 or 1 for a result. An evaluation function must have some way of ranking solutions on a larger scale. The following suggestion quoted by DeJong and Spears proposes an averaging scheme [1].

$$\text{val(AND } e_1 \ \ldots \ e_n) = \text{AVG}(\text{val}(e_1) \ \ldots \ \text{val}(e_n))$$

This function will be closer to 1 when most of the values are true, and closer to 0 when most of the values are false. It returns a value between 0 and 1 inclusive, and provides the evaluation function with more information than just simple true and false[2].

Many NP problems have poor genetic algorithm representations. Since all NP problems can be mapped back to satisfiability, and satisfiability has a good genetic algorithm representation, then it follows that many NP problems might benefit from using this technique. DeJong and Spears solved the hamiltonian circuit problem as a proof-of-concept example and also discussed the possibility of solving certain cryptography problems this way. This paper will show how this mapping was used to solve the n-queens problem.

2 Solving n-Queens with Genetic Algorithms

2.1 Forming a Representation

The representation for the n-queens problem will be a list of n unique integers between 1 and n inclusive as shown earlier in Figure 1. representation is the n-tuple defined earlier. The position within the tuple denotes the grid column and the number stored at that position is the grid row.

2.2 Specialized Crossover and Mutation Operators

The two-point crossover operator described earlier would have to be modified to work here. Simply exchanging parts of two solutions will

[1] Attributed to an unpublished work by Gerald H. Smith

[2] DeJong and Spears point out that before using this technique, make sure the scope of all NOT constructs has been reduced to simple variables (i.e. apply DeMorgan's law to terms of the form $\overline{(AND \cdots)}$ and $\overline{(OR \cdots)}$).

usually result in an invalid solution (remember that the numbers in the n-tuple are all distinct.) For example,

(2 5 1 | 3 8 4 | 7 6)
(8 4 7 | 2 6 1 | 3 5)
becomes
(2 5 1 | 2 6 1 | 7 6)
(8 4 7 | 3 8 4 | 3 5)

Both of the resulting children are invalid. A special technique called Partially Matched Crossover (PMX) [4] fixes this problem be adjusting the solution elements in conflict after the crossover has been done. This operation is shown below.

(2 5 1 | 2 6 1 | 7 6)
(8 4 7 | 3 8 4 | 3 5)
becomes
(3 5 4 | 2 6 1 | 7 8)
(6 1 7 | 3 8 4 | 2 5)

Notice how in the first solution, the 2 at position 1 was in conflict with the 2 at position 4. Since the 2 at position 4 is newer (from the crossover operation), the 2 at position 1 is changed to the 3 that used to be at position 4 before the crossover. This effectively cleans up the tuple.

For mutation, a simple operator is to select a single n-tuple from the population pool, randomly select two distinct points in the n-tuple and swap their values. This creates a new potential solution that is very similar to its "parent" and preserves the validity of the n-tuple. For example, if one of the solutions shown above were mutated at positions 2 and 6,

(3 5 4 2 6 1 7 8)
becomes
(3 1 4 2 6 5 7 8)

Experiments showed that PMX did not behave well for this problem and tended to unify the solution pool (i.e. all solutions in the pool eventually became identical, which makes for very uninteresting crossover). PMX would quickly generate near solutions, but then would spend a disproportionate amount of time finding exact solutions. After testing, it was clear that although PMX began well, the simple mutation operator was the only reason the algorithm ever even converged.

As a second attempt, PMX was replaced with the current mutation operator. Thus, the same operation was used for both crossover and mutation. This began to converge more quickly and at a much steadier rate. All of the test results shown later are based on this simplified scheme. Because of the evaluation function (see below), smaller perturbations of solutions produce better results. One possibly useful combination might be to start with PMX to quickly obtain some close solutions and then switch to the simpler crossover operator at some point where smaller perturbations are needed.

2.3 Developing Heuristics for an Evaluation Function

The problem with developing an evaluation function for n-queens is that the answer is either correct or incorrect. Genetic algorithms typically do not work well when the evaluation function only returns true or false. Thus, ad hoc evaluation heuristics must be devised to say how good a particular incorrect solution is. One way to do this is by counting the number of diagonal conflicts.

Horowitz illustrates a simple method for finding conflicts [6]. Consider the ith and jth queens in an n-tuple.

$$(q_1, \ldots, q_i, \ldots, q_j, \ldots, q_n)$$

The queen on column i is on row q_i, and the queen on column j is on row q_j. These two queens are on the same diagonal only if

$$i - q_i = j - q_j$$
$$or$$
$$i + q_i = j + q_j$$

which reduces to

$$||q_i - q_j|| = ||i - j||$$

The evaluation function can use this test of absolute values to count how many conflicts exist for the proposed solution. The total number of conflicts tells the genetic algorithm how "fit" a particular solution is.

A simple approach to this problem produces an algorithm that is $\bigcirc(n^2)$ (i.e. for each queen in the n-tuple, compare it with every queen to its right). While it converges well, it is quite slow. A second look at the problem leads to a better solution. By allocating a counter for each diagonal in the grid an algorithm can be developed that runs proportional to $\bigcirc(n)$. There are $2n - 1$ top-down, left-to-right diagonals (called left diagonals from here on), and $2n - 1$ bottom-up, left-to-right diagonals (called right diagonals).

A queen on column i and row q_i is on the $n-i+q_i$ left diagonal, and on the $i+q_i-1$ right diagonal. After an evaluation, if the counters are all 0 or 1 (i.e. 0 or 1 queens on each diagonal) the answer is correct. Otherwise, for all counters > 1, subtract 1 and add them together to evaluate the fitness of the proposed solution.

This is how the ad hoc evaluation function gives a value to a proposed solution. For example, in Figure 1, the left diagram (1,4,2,3) has a value of 1, and the right diagram (3,1,4,2) shows a correct answer with a value of 0.

Because the evaluation function is based on *queen conflicts*, smaller perturbations to solutions are more effective than larger ones. This is evidenced by the fact that the PMX operator shown earlier was outperformed by a simple swap of two queen positions.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

Figure 2: Matrix representation of 4-queens

0	1	1	1
1	1	0	1
1	1	1	0
1	0	1	1

Figure 3: Boolean representation of 4-queens

2.4 An Evaluation Function Based on Satisfiability

To map the n-queens problem onto satisfiability, consider the 4-queens problem again. One simple mapping (there may be others) is to consider the 4 x 4 grid as having only 1's and 0's, where a 0 represents a queen (using a 0 for a queen simplifies the boolean function by not having to negate all inputs.) Given a grid arranged as a matrix in figure 2, think about the boolean values as in figure 3. Given this representation, the resulting boolean formula is shown in figure 4.

Notice how the satisfiability function simply looks at every diagonal to make sure exactly 0 or 1 queens exist. This can be simulated in an evaluation function by computing normalized sums of the diagonals to get the heuristics from DeJong and Spears. Figure 5 shows an algorithm for such an evaluation function that runs proportional to $\bigcirc(n)$.

It turns out that this evaluation function is very similar to the ad hoc function. Even though they look different and use different

(AND (OR $a_{31}a_{42}$)
(OR (AND $a_{21}a_{32}$)
(AND $a_{21}a_{43}$)
(AND $a_{32}a_{43}$))
(OR (AND $a_{41}a_{32}a_{23}$)
(AND $a_{41}a_{32}a_{14}$)
(AND $a_{41}a_{23}a_{14}$)
(AND $a_{32}a_{23}a_{14}$))
(OR (AND $a_{12}a_{23}$)
(AND $a_{12}a_{34}$)
(AND $a_{23}a_{34}$))
(OR $a_{13}a_{24}$)
(OR $a_{21}a_{12}$)
(OR (AND $a_{31}a_{22}$)
(AND $a_{31}a_{13}$)
(AND $a_{22}a_{13}$))
(OR (AND $a_{11}a_{22}a_{33}$)
(AND $a_{11}a_{22}a_{44}$)
(AND $a_{11}a_{33}a_{44}$)
(AND $a_{22}a_{33}a_{44}$))
(OR (AND $a_{42}a_{33}$)
(AND $a_{42}a_{24}$)
(AND $a_{33}a_{24}$))
(OR $a_{43}a_{34}$))

Figure 4: Satisfiability function for 4-queens

```
set left and right diagonal counters to 0
for i = 1 to n
    left_diagonal[i + q_i - 1]++
    right_diagonal[n - i + q_i]++
sum = 0
for i = 1 to (2n - 1)
    counter = 0
    if (left_diagonal[i] > 1)
      counter += (left_diagonal[i] - 1)
    if (right_diagonal[i] > 1)
      counter += (right_diagonal[i] - 1)
    sum += counter / (n - ||i - n||)
evaluation = sum / (4n - 2)
```

Figure 5: Evaluation based on satisfiability mapping

strategies, they both ultimately count the number of queen conflicts. During testing, both functions converged well, with the ad hoc function usually requiring fewer iterations.

3 Test Results

All n-queens testing was done with GENITOR [9], a robust genetic algorithms package that facilitates manipulation of binary, integer, and floating point representations. This package allowed quick implementation and testing, including the hand-tailored crossover and mutation operators (GENITOR offers several others to choose from) and multiple evaluation functions.

The population size was 1000 (except for the 1000-queens problem where a population size of 2000 was used), the mutation rate was .01, and the selection bias was 1.9. Both the ad hoc evaluation function and the satisfiability mapping evaluation function converged in a similar manner and found solutions in every case tried.

n	*Sparc 1+*	*RS 6000*	*Number of iterations*
10	8 sec	14 sec	900
20	31 sec	26 sec	17000
50	2 min	2 min	60000
100	9 min	8.5 min	236000
150	22 min	17 min	382000
200	50 min	30 min	510000
500	8.2 hrs	1.9 hrs	3200000
1000	n/a	20 hrs	7410000

Table 1: Times for ad hoc evaluation function

n	*Sparc 1+*	*RS 6000*	*Number of iterations*
10	11 sec	5 sec	1900
20	22 sec	11 sec	9000
50	3 min	1.2 min	99000
100	10.5 min	4.5 min	242000
150	24 min	10.5 min	416000
200	75 min	21 min	675000
500	7.5 hrs	2 hrs	1870000

Table 2: Times for evaluation function based on satisfiability

n	*Search Space* $n! \approx 10^i \approx 2^j$		
10	10!	$10^{6.56}$	$2^{21.8}$
20	20!	$10^{18.3}$	2^{61}
50	50!	$10^{64.5}$	2^{214}
100	100!	10^{158}	2^{524}
150	150!	10^{263}	2^{873}
200	200!	10^{375}	2^{1245}
500	500!	10^{1134}	2^{3767}
1000	1000!	10^{2567}	2^{8529}

Table 3: Search space sizes

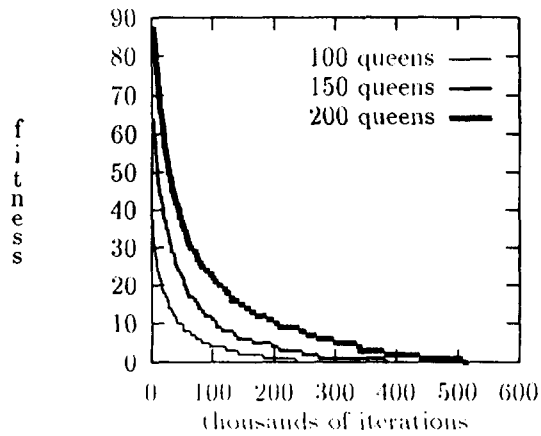

Figure 6: Convergence for 100-, 150-, and 200-queens

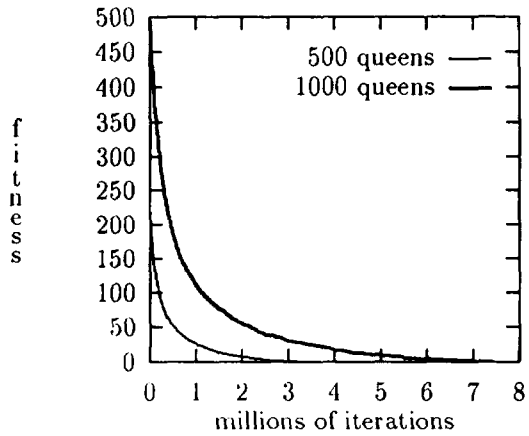

Figure 7: Convergence for 500- and 1000-queens

Convergence rates for 100-, 150-, 200-, 500-, and 1000-queens are shown in the Figures 6 and 7. Actual solutions to the 100- and 200-queens problems are shown in the appendix.

Several timed tests were run on both a SparcStation 1+ and an IBM RS6000. Tables 1 and 2 show the results of these tests for various sizes of n. Table 3 shows the search space sizes. These sizes relative to the number of iterations required to find a solution shows the power that genetic algorithms offer when solving problems of this magnitude.

4 Conclusions

While the n-queens problem itself is not very practical, it does represent a large set of practical problems in class NP, specifically, decision problems. These problems have simple "yes" and "no" answers so heuristics must be developed in order to solve them using genetic algorithms (i.e. to get a working evaluation function).

Two successful approaches to the n-queens problem were used. One showed an ad hoc approach (count the number of queen conflicts), and the other used the satisfiability heuristics developed by DeJong and Spears. Results were comparable for both. Future work in this area would include tests on other decision problems and comparisons with existing approximation techniques (on speed, convergence, % optimal, etc.)

References

[1] Sara Baase. *Computer Algorithms: Introduction to Design and Anaylsis.* Addison-Wesley Publishing Company, Reading, MA, 1978.

[2] Lawrence Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, NY, 1991.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[4] David E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*. Morgan Kaufmann Publishers, Inc, 1987.

[5] John H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[6] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Inc., Rockville, MD, 1978.

[7] Kenneth A. De Jong and William M. Spears. Using genetic algorithms to solve np-complete problems. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 124–132. Morgan Kaufmann Publishers, Inc, 1989.

[8] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[9] D. Whitney and J. Kauth. Genitor: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130. Denver, CO, 1988.

A A Solution to the 100-Queens Problem

(3,47,54,2,60,12,74,15,90,31,15,20,32,55,62,
69,100,58,14,72,86,68,81,3,47,89,84,8,85,17,
56,7,87,46,97,59,19,30,77,26,32,59,63,78,41,
29,83,61,6,76,13,26,18,64,98,5,10,88,25,37,
1,93,91,49,28,4,96,73,96,86,80,71,18,50,14,
66,95,67,23,94,22,70,42,52,82,53,34,59,9,44,
75,92,57,67,98,77,8,35,24,70)

B A Solution to the 200-Queens Problem

(162,37,117,23,78,91,105,147,132,175,188,3,198,
143,114,192,149,119,130,106,82,52,94,87,77,179,
19,104,70,112,65,141,103,128,88,39,150,126,111,
11,84,5,92,159,187,32,142,127,60,27,140,67,180,
174,83,161,30,154,56,74,173,181,47,96,186,4,54,
145,123,167,22,182,72,31,55,163,184,61,89,116,
13,49,133,146,124,176,165,168,193,18,151,109,
196,73,95,17,118,155,8,125,42,169,25,93,51,144,
46,64,200,36,107,152,178,44,120,6,69,33,63,68,
86,71,101,183,29,43,172,24,41,139,197,158,170,
102,12,35,7,45,195,171,20,166,97,50,121,75,57,
62,157,34,14,134,115,177,21,122,138,1,90,153,
190,15,81,99,185,53,100,113,58,131,129,85,110,
164,160,26,2,137,76,40,156,189,199,16,108,9,59,
10,191,194,38,135,48,28,80,98,79,148,66,136)