Enhancing the Performance of Genetic Algorithms in Combinatorial Optimization of Large and Difficult Problems

Using Variable Adaptive Mutation Rates Controlled by 'Inbreeding'

Daniel Smullen*
UOIT
2000 Simcoe St. N
Oshawa, ON, Canada
daniel.smullen@uoit.net

Jonathan Gillett
UOIT
2000 Simcoe St. N
Oshawa, ON, Canada
jonathan.gillett@uoit.net

Joseph Heron
UOIT
2000 Simcoe St. N
Oshawa, ON, Canada
joseph.heron@uoit.net

Shahryar Rahnamayan
UOIT
2000 Simcoe St. N
Oshawa, ON, Canada
shahryar.rahnamayan@uoit.ca

ABSTRACT

Blah blah blah, here is our abstract.

Keywords

Genetic Algorithms, Combinatorial Optimization, N Queens Problem, Variable Mutation

1. INTRODUCTION

Genetic algorithms serve an important role in various applications, but particularly useful areas are those where they can perform stochastic generation of solutions for combinatorial problems. One notable example of this is the N-Queens problem. One approach to solving the N-Queens problem using a genetic algorithm is to implement chromosomes which model the position of each queen on the chessboard. The permuted values stored in the chromosome represent the sequential row positions of each queen, encoded into binary values. Each column value is the index, since there can never be more than one queen per column. Thus, the column positions increase by one for each queen and the intersections (referred to as collisions) are calculated per row or on the diagonal to determine whether a solution has been found. As with any genetic algorithm, evaluating the fitness of the chromosomes per generation is required to determine if a usable solution has been generated.

By tuning the parameters of the genetic algorithm, performance can increase or decrease based on the solution landscape of the problem encountered. We have found that by replacing a static mutation value with an adaptive value controlled by chromosome similarity, better performance can be achieved than using static values, particularly when there is no *a priori* knowledge about the ideal static mutation rate. This is compounded when the solution landscape is large, such as in higher-order N-Queens problems.

*Corresponding author.

GECCO2014 2014 Victoria, BC, Canada

1.1 Background and Problem Domain

Like many other complex optimization problems, the N-Queens problem becomes orders of magnitude more complex as the number of queens and the size of the chessboard increases. Finding all solutions is a simple but non-trivial problem. When two or more queens share a row or diagonal, a collision occurs for each pair of queens, meaning that the board state is not a solution.

The Eight-Queens problem is the classical version of the puzzle, and even with this configuration the problem is computationally expensive. There are $\binom{64}{8}$ arrangements of the queens on a standard 8×8 board. Exhaustive deterministic evaluation has shown that there are only 92 distinct solutions. Even including optimizations such as imposing a constraint to place only one queen to a single row, there are still 8^8 possible distinct arrangements. In higher-order versions of the problem, combinatorial explosion occurs. The Nine-Queens problem has 352 distinct solutions. The Ten-Queens problem has 724. Table 2 in section A.1 shows the growth of the number of solutions.

While the number of solutions is known for problems involving up to 26 queens, there is currently no known formula to determine the exact number of distinct solutions. That is to say, higher order N-Queens problems are currently intractable with existing methods. Deterministic methods for approaching these problems are largely useless, and the problem remains unsolved.

- (ADD CITATIONS!
- Cite me![2, 4, 1, 6, 7, 5, 3]

1.1.1 The N-Queens Puzzle

The natural question raised when approaching simple problems is why a deterministic or brute force strategy is not

used. Unfortunately, these approaches cannot work within human time-frames except in smaller problem landscapes. The issue is that a direct linear traversal of the landscape does not yield fruitful results easily in the case of the N-Queens problem. This stems from the fact that the fitness of each sequential solution doesn't matter. The problem cannot be solved by producing a solution that has queens which will attack each other. Therefore, only solutions with the absolute maximum, 100% fitness, are useful solutions. What is more, as the problem landscape increases in size, the number of distinct solutions does not increase proportionally. A full brute force traversal of the landscape would be required to find all the distinct solutions, which is increasingly expensive in higher order problems.

Other stochastic solutions are impractical with this type of problem. This is because finding solutions among the landscape of possible board states yields a tremendous amount of duplicate solutions. Some method is required to direct the random traversal of the landscape toward distinct solutions and away from those which have already been found. The problem with this statement, is that the nature of randomness cannot be constrained in this way - it will no longer be random. Therefore methods such as genetic algorithms become an attractive solution approach because they represent a 'smarter' traversal of the problem landscape than what is offered by a linear brute force traversal, or a uniformly random traversal of the landscape. Less time is spent on areas of the landscape which are not likely to be solutions. This leads us toward the core motivation of our solution's behavior.

- (ADD CITATIONS!)

1.1.2 Motivation

Modeling the randomness of the solution landscape as seen in the N-Queens problem is impossible with known mathematical methods. To produce a geometric solution landscape would require that the dependent axis of the coordinate plane maps each sequential value to a uniformly random value in the actual solution landscape. Conceptually this is very difficult to imagine, but the reason why this conceptual strategy is useful will become clearer. First, we will discuss why a new approach is necessary in comparison to traditional GA approaches.

There are several 'short-cuts' which are available when approaching the N-Queens problem because of the nature of the chess board. Since it is square, one solution in fact can be used to yield many other unique solutions using symmetry operations (reflection and rotation of the board). This means that up to 8 solutions can be generated by finding one distinct solution. However, symmetry operations cannot be used to find all distinct solutions because many symmetrical configurations of the board are equivalent. In the example of the Eight-Queens problem, there are 11 board configurations which yield 8 solutions, and 1 which yields only 4. The result is still 92 unique solutions, with the majority of them calculable through symmetry operations alone.

Using symmetry operations, our initial exploration into solving the N-Queens problem yielded solutions quickly, but not quickly enough to beat deterministic solution generation methods. This was especially apparent in lower order problems where the solution landscape is small. We began to explore biological definitions of genetics, and sought inspiration from the fact that nature has an apparent system for maintaining genetic variation within organisms. For example pure-bred dogs often tend to have significant health problems which directly result from inbreeding. Royal families historically used inbreeding to keep their line 'pure', which also resulted in genetic abnormalities. (CITE THIS)

These facts led us toward a theoretical approach to apply the negative effects of inbreeding towards genetic algorithms. Our approach would make the mutation rate variable based on chromosome similarity, suggesting the natural mutant products of inbreeding as a result of a too-similar biological chromosome. Our initial solution worked exceptionally well, providing all 92 solutions nearly instantaneously in comparison to our previous static mutation rate GA which required a few seconds. This motivated us to pursue the implications of our strategy further.

1.2 Related Work

- This is where you can cite all of the references Joseph found regarding other types of variable mutation. - Joseph DO THIS

2. OUR SOLUTION APPROACH

Ordinary genetic algorithms use three parameters to govern their evaluation. The first is the population size. We have not adjusted this parameter in our experiments, and kept them at a fixed value of 64 chromosomes. Second is the chromosome crossover rate, which governs the likelihood and percentage of a chromosome to splice with another. This was also kept static, at a fixed rate of 70%. The third parameter is mutation rate, which governs the likelihood that a chromosome will have a random gene modified. This is achieved using a bit-flipping operation on the selected gene's binary encoded values. To support the 'in-breeding' variant of the algorithm, we have selected to replace the fixed mutation rate parameter with a variable rate, governed by a separate parameter which was kept at a fixed value for the sake of experimentation. The separate parameter, chromosome similarity threshold, is expressed as a percentage. We have selected 15% as an arbitrary threshold which seemed to work well during experimentation. Future studies into the nature of this parameter may yet yield improved results for specific circumstances. The offset by which the mutation rate is adjusted (based on the chromosome similarity parameter) is set at a fixed value of 1%.

(cite paper justifying why we used these values, they are often recommended/used values, Genetic Algorithms by Goldberg)

2.1 Applying Adaptive Variable Mutation Rates

In order to apply our adaptive variable mutation rate, a two step process is required. For each generation of chromosomes, the population chromosome similarity must be evaluated. The similarity algorithm is detailed in a separate subsection.

When the chromosome similarity is less than the specified threshold, the mutation rate must decrease. For the next generation, the mutation rate is decreased. If the chromosome similarity increases again, and it goes above the specified threshold, the mutation rate decreases. Due to the stochastic nature of the algorithm, wild variations in chromosome similarity may result regardless of the mutation rate. Since the crossover of the genes may be up to 70%, the resultant permutations of the genome are likely to

be similar unless some mutation occurs. The result of this adaptive change in mutation rate is that the chromosome similarity will approach an equilibrium close to the specified inbreeding threshold.

2.1.1 Chromosome Similarity Algorithm

Chromosome similarity is calculated using a greedy algorithm seen in Algorithm 1. First, the chromosomes' genes are decoded into integer values by direct translation of their binary values in sequence to an integer. next, the array of chromosomes' integer representations is sorted. Our implementation uses the quick-sort algorithm for minimal computational overhead. This gives the algorithm its' characteristic asymptotic behavior, as the complexity of the actual sorting function itself is of higher complexity than the main similarity calculation algorithm. That is to say, the sorting function used runs on average in $O(n \log n)$ complexity, although it should be noted that in the theoretical worst case, quick-sort works in $O(n^2)$ which is highly unlikely. The main algorithm runs in $O(n)$ linear complexity.

```
Function Similarity(chromosomes)
    input  : An array of chromosomes
    output: Fraction of chromosomes that are similar
    similar ← 0
    matched ← false
    length ← length of chromosomes
    // Sort using an arbitrary sorting algorithm
    sorted ← Sort(chromosomes)
    for i ← 0 to length − 1 do
        if sorted[i] == sorted[i + 1] then
            similar ← similar + 1
            matched ← true
        else if matched then
            similar ← similar + 1
            matched ← false
        end
        // Case where the last item is a match
        if matched and (i + 1 == length − 1) then
            similar ← similar + 1
        end
    end
    return similar/ length
end
```

Algorithm 1: Chromosome similarity function

2.2 Implementation Specific Considerations

In order to gain an appreciation for the specific implementation and how it might compare to alternatives, we will attempt to elaborate upon the special implementation considerations we made. Our experiments were performed on the same code as our development code, which was written in Java. Concurrent agile development was used as our main development methodology, and GitHub was used extensively for version management and simultaneous development of the code, documentation, and this report.

The project source code is available at `http://www.github.com/gnu-user/`

2.2.1 Chromosome Design

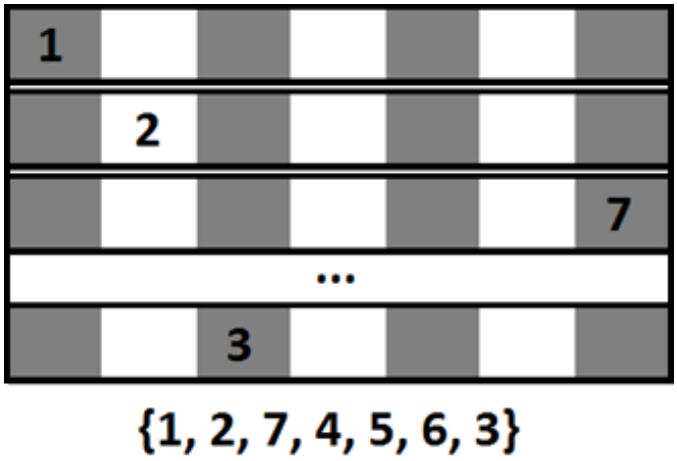

Figure 1: Illustrating the chromosome data structure.

Chromosomes are designed with an optimization that allow the stochastic search space to be minimized significantly, mentioned originally in section 1.1.2. The result is that the dimensionality of the problem decreases. This is accomplished by encoding each gene as a row (or column) position on the chess board, and each single index in the array of values means there can only be one queen placed at each index. This allows for an extendable chromosome up to N. Figure 2.2.1 shows this chromosome pictorially for a 7×7 chessboard. In the figure, the 4th, 5th and 6th rows are not shown, but are filled with a queen at the 4th, 5th and 6th column positions respectively in this chromosome. As a result, the encoded chromosome is represented by 7 Java Integer class values in sequence. To encode this into one mutable array, the integers are simply concatenated together. The array specified in Figure 2.2.1 would thus be translated to 1274563.

2.2.2 Fitness Function

Fitness is evaluated by determining the number of collisions between queens on the chessboard - in other words, when at least two queens can attack each other, a collision occurs for each queen. This means that in a board state where only one queen can attack another, two collisions result. Because it is impossible for there to be more than one queen per column given the data structure, only collisions across rows and diagonals must be found. The algorithm for determining the fitness of a given board state is given in Algorithm 2, showing the mechanisms we used to find collisions across rows and diagonals. The fitness is calculated as $1 - \frac{1}{(1/N)}$, where the maximum fitness (a unique board solution) is 1. The length of the chromosome corresponds to the number of queens and rows on the square chessboard, N.

2.2.3 Selection Method

As per traditional genetic algorithms, roulette wheel selection was used. This process works by filling the roulette wheel with fractional parts of a total. This is analogous to a probability distribution between zero and 1 - in our implementation the range will lie between 0 and the sum of all

```
Function Fitness(chromosome)
    input  : A single chromosome
    output: A fitness value for the chromosome
    collisions ← 0
    length ← length of the chromosome
    for i ← 0 to length − 1 do
        // Check each gene against the current
        j ← (i + 1)  mod length
        while j != i do
            y_i ← chromosome[i]
            y_j ← chromosome[j]
            // Check for vertical collision
            if y_i == y_j then
                collisions ← collisions + 1
            end
            // Check for diagonal collision
            if abs((i − j) / (y_i − y_j)) == 1 then
                collisions ← collisions + 1
            end
            j ← j + 1
            j ← j  mod length
        end
    end
    if collisions == 0 then
        return 1
    else
        return 1 / collisions
    end
end
```

Algorithm 2: Fitness function

the fitness values of each chromosome in a generation.

Each chromosome is given a fraction of the roulette wheel, whose size is equal to the chromosome's fitness divided by the total fitness of all values. A random floating point value is generated between zero and the sum of the total fitness values. This floating point value will lie within one of the intervals representing one of the chromosomes. Whichever interval it falls in is the selected chromosome. The order in which the chromosomes' intervals are placed doesn't matter, as each 'spin' of the roulette wheel is random - the floating point number generated is created based on a uniformly random distribution.

2.2.4 Chromosome Operations

As mentioned in section 2, the crossover rate is set at a fixed 70%. This means that 30% of the time, the genes in the child chromosomes will be an exact clone of the parents. the remaining times, a random crossover occurs. This is done the traditional way by selecting a random index as the bounds for the portion of the gene to cross over. All indexes to the right of this index (including the index value itself) are swapped. Between the two parents, the first parent's values to the right of the index, and the second parent's values to the left of the index are combined to create one child (and the other way around for the second child).

Background mutation, naturally, affects chromosomes randomly after the crossover or cloning process takes place. Each chromosome has a percentage chance of being affected by mutation, and this likelihood changes every generation due to the adaptive variable mutation rate. When a chromosome is selected for mutation, one of the genes in that chromosome is selected randomly. It is then set to a random, valid value. Valid values are between 0 and N corresponding to each row on the chessboard.

2.2.5 Chromosome Evaluation

Each generation, there is a likelihood that some of the chromosomes in the current population are valid solutions - any of the current generation's chromosomes with a fitness of 1 is some sort of solution. When a solution is found, it must be compared to the list of previously found solutions to ensure it is not a duplicate. Due to the stochastic nature of the algorithm, it is a strong possibility that many solutions can be found which are duplicates of previous solutions. To add to this fact is that the symmetry operations applied to each solution found may further increase the likelihood that the new population's solutions that have already been found. Duplicate solutions are discarded without having symmetry operations applied as they would be identical to symmetric configurations of existing found solutions.

Upon saving any newly found solutions, the new generation replaces the previous generation and the genetic algorithm process cycle continues again.

2.3 Experimental Methodology

- Implementation in Java

- Experiments were conducted on the HPC facilities provided by SHARCNET

- describe the study and control (diff. fixed mutation rates, list them all, vs. variable mutation rate with a fixed inbreeding threshold of 15%), this had 1 fixed param (inbreeding threshold)

–> This should probably be put in a table. I'll create one

for you

- N queens problem sizes used: For 8 - 16 queens used each n queens size, for 16 - 26 used each even sized N queens problem, lastly a test using 32 queens.

- Number of generations for each N queens problem (10 million generations for all except 32 queens), 50 million generations were used for 32 queens given the increased complexity of the problem.

- sample sizes (30, 15, 10)

–> I will give you a table with each n queens problem, the sample sizes used for each and the number of generations, this would reduce a lot of text needed to explain the items.

–> justify why the sample sizes were reduced for larger problems given the computational limitations of SHARCNET (max 256 jobs, 7 days CPU time, regardless and your job would be killed). We would like larger sample sizes (100+ runs) but given the CPU time limitations of SHARCNET could not.

- describe the data collected, explain how each of the attributes such as population fitness, similarity are calculated based on the mean of the population for 1000 generations at a time (mean of means), rather than the mean of each population for each generation (TOO MUCH DATA!)

2.4 Why Not Fixed Mutation?

- This should be part of the end of results basically summarizing the pros and cons of fixed mutation and variable with reference to how our results show that in our case (for certain larger problems) variable mutation was better.

3. RESULTS

- cite specific results and why we think they are important, what is the significance of them and how they support our results/hypothesis that in the case of N Queens var. mutation is better than fixed.

- cite the result showing the best fixed mutation vs. the variable mutation for each N-queens problem, try and use both the figure and the table, the table has some additional interesting information which cannot be conveyed in the image alone.

–> I will create the table for you, essentially showing each N queens problem, the best results for fixed mutation vs. the best results for variable mutation

- cite interesting results of the fat boxplots in the range of chromosome similarity for the optimal fixed mutation rates, use the "person stepping" analogy and how that certain fixed mutation rates had the widest range in chromosome similiarty allowing it to hone in on solutions faster.

- Try and incorporate one of the scatter plots as well that show very interesting results and see if you can use it to compare/contrast the results of the following plots

- variable mutation rate scatter plot - variable mutation rate similarity scatter plot - best fixed mutation rate similarity scatter plot

4. CONCLUSIONS

- Wait until the rest of the paper & we have more feedback before writing the conclusions.

- Further research into adjusting the inbreeding threshold, for the purpose of the research a constant inbreeding threshold of 15% was used, however further research could be done in testing different thresholds.

- Comparing the results of variable mutation with fixed mutation using other types of combinatorial/optimization problems such as TSP, constraint satisfaction problem (CSP), etc.

- Variable population size based on the amount of inbreeding (if you have a lot of inbreeding in nature the organisms will have higher mutatation rate and deformities, the population will shrink)

- Having the mutation operator affect more than one gene if it goes > 100%?

5. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you.

- Shahryar
- SHARCNET

6. REFERENCES

[1] P. S. Andrews. An investigation into mutation operators for particle swarm optimization. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 1044–1051. IEEE, 2006.

[2] K. D. Crawford. Solving the n-queens problem using genetic algorithms. In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's*, SAC '92, pages 1039–1047, New York, NY, USA, 1992. ACM.

[3] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[4] A. Homaifar, J. Turner, and S. Ali. The n-queens problem and genetic algorithms. In *Southeastcon '92, Proceedings., IEEE*, pages 262–267 vol.1, 1992.

[5] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):656–667, 1994.

[6] A. Tuson and P. Ross. Adapting operator settings in genetic algorithms. *Evolutionary computation*, 6(2):161–184, 1998.

[7] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

APPENDIX

A. RESULTS

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with **subsection** as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Tables

- Additional tables can go here

Table 1: Number of unique solutions to the N-Queens Problem

N	Distinct Solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14,200
13	73,712
14	365,596
15	2,279,184
16	14,772,512
17	95,815,104
18	666,090,624
19	4,968,057,848
20	39,029,188,884
21	314,666,222,712
22	2,691,008,701,644
23	24,233,937,684,440
24	227,514,171,973,736
25	2,207,893,435,808,352
26	22,317,699,616,364,044

Table 2: Best Solution Comparison Between Fixed and Variable Mutation Rates

Queens	Mutation Rate	Unique Solutions	N	Min Generations	Mean Generations	Max Generations
8	0.95	92	30	1,793	4,746	10,815
	variable	92	30	51,23	15,293	42,002
9	0.95	352	30	48,811	144,456	372,281
	variable	352	30	229,242	1,246,215	3,038,980
10	0.9	724	30	188,372	408,402	771,208
	variable	724	30	531,588	1,629,893	6,195,099
11	0.85	2,680	30	2,400,188	3,799,795	5,933,213
	variable	2,680	5	6,900,512	7,909,269	9,449,545
12	0.85	13,690	1	9,890,455	9,890,455	9,890,455
	variable	11,986	1	9,973,787	9,973,787	9,973,787
13	0.8	32,128	1	9,998,058	9,998,058	9,998,058
	variable	26,308	1	9,978,215	9,978,215	9,978,215
14	0.8	41,520	1	9,996,747	9,996,747	9,996,747
	variable	29,520	1	9,999,583	9,999,583	9,999,583
15	0.8	30,356	1	9,982,638	9,982,638	9,982,638
	variable	30,324	1	9,996,584	9,996,584	9,996,584
16	0.8	15,016	1	9,987,788	9,987,788	9,987,788
	variable	30,132	1	9,992,266	9,992,266	9,992,266
18	0.75	16,392	1	9,997,986	9,997,986	9,997,986
	variable	27,120	1	9,998,858	9,998,858	9,998,858
20	0.75	7,872	1	9,972,915	9,972,915	9,972,915
	variable	25,608	1	9,931,631	9,931,631	9,931,631
22	0.7	6,008	1	9,959,712	9,959,712	9,959,712
	variable	25,376	1	9,991,089	9,991,089	9,991,089
24	0.7	6,440	1	9,929,626	9,929,626	9,929,626
	variable	24,560	1	9,998,508	9,998,508	9,998,508
26	0.7	6,280	1	9,969,840	9,969,840	9,969,840
	variable	23,008	1	9,983,128	9,983,128	9,983,128
32	0.65	15,216	1	49,732,320	49,732,320	49,732,320
	variable	104,080	1	49,996,701	49,996,701	49,996,701

A.2 Figures

- Additional figures can go here

A.3 Source Code

- Could link to github and possibly the github wiki, or explanation of the R analysis source code.

A.4 Raw Data

- Link to the dropbox URL containing the actual data from SHARCNET used for the research