Search & Optimisation

Machine Learning I

Edward Fleri Soler

Table of Contents

1. Introduction

This assignment is focused on the implementation of search and optimisation algorithms to find solutions to the NP hard problem that is the N-Queens problem. Three different search and optimisation algorithms will be considered in this assignment: Creeping Random Search, Simulated Annealing and Genetic Algorithms.

1.1 N-Queens

The N-Queens problem has evolved from the original 8-queens problem, first proposed by Chess composer Max Bezzel. The proposal was that of filling an 8x8 chess board with 8 queens, and trying to find a position for each queen whereby no rules of chess are violated. This means that no two queens may lie on the same column, row or diagonal.

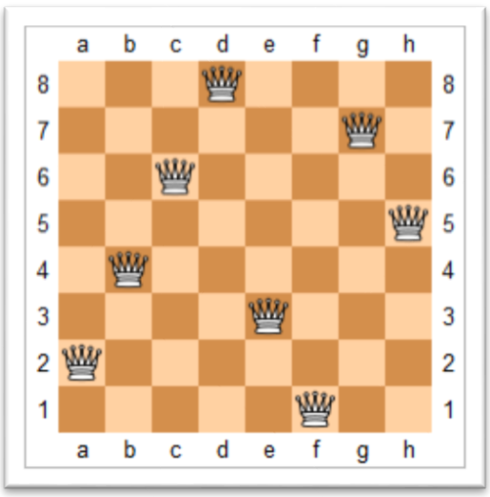

Figure 1: One possible solution to the 8-Queens problem

This problem was later expanded to an NxN board with N queens, earning the name "N-Queens problem". Today this problem has gained popularity for its NP (non-deterministic polynomial-time) quality, meaning that the running time to compute a solution for such a problem grows exponentially as the search space grows linearly. In N-Queens, for a board of size N, $^{(NxN)}C_N$ possible arrangements exist, however only a miniscule amount of these are solutions. Therefore, for a board of size 4x4, 1820 possible arrangements exist, however for a board of size 8x8, 4,426,165,368 arrangements exist, enforcing the exponential growth in the search space of this problem.

Searching for a solution to such problems in a linear fashion is therefore impossible for large values of N. That is why this problem is considered to be ideal as a test-case for search and optimisation algorithms.

1.2 Creeping Random Search

Creeping Random Search is one of the three search and optimisation algorithms that will be considered in this paper. The principle behind this algorithm is to iteratively traverse the search space in search of a better solution by sampling a neighbourhood of solutions within a hypersphere. This algorithm is therefore owed its name by its fashion of search and can be considered to be a greedy algorithm. However, since a search space may be dotted with a number of local maxima (which are not the global maximum), global searches which fall outside of the current hypersphere are incorporated, in the hope of finding one of the possible global maxima (solutions).

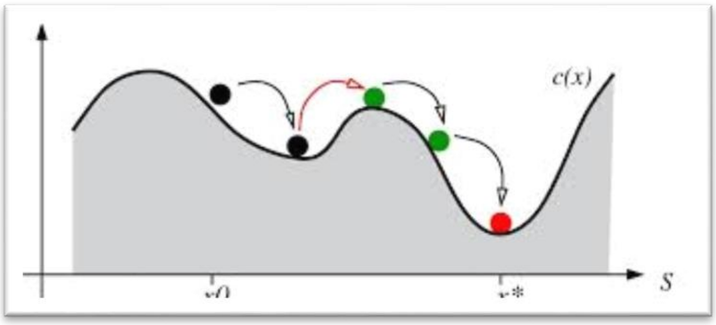

Figure 2: Possible search space traversal

1.3 Simulated Annealing

Simulated Annealing is a search and optimisation problem inspired by the physical properties of metal. Annealing involves heating metals up to high temperatures and letting them cool at different rates, to allow the atoms to form new unique bonds and structures, giving the metal new properties such as malleability. Different cooling rates give different amount of time to the metal atoms to rearrange themselves, resulting in different properties of the metal.

Figure 3: Heating of metal

Simulated annealing involves mimicking this process in a search space for a problem. A virtual starting temperature is set as well as a static or dynamic cooling rate. The temperature at any point in time during the execution of the algorithm dictates the acceptance of sub-optimal solutions. Therefore, when the temperature is high, the algorithm is more likely to accept sub-optimal solutions, resembling a completely random search. However, as the temperature gradually decreases, the tolerance for sub-optimal solutions decreases, resembling a greedy hill-climbing algorithm.

This unique combination allows for the general location of a general maximum to be found at the early stages, while the actual maximum; that is the peak (or trough) to be located as the temperature decreases.

1.4 Genetic Algorithms

As was the case with Simulated Annealing, Genetic Algorithms are inspired by nature, specifically the nature of evolution. In the late 1860s the English naturalist Charles Darwin proposed the theory of natural selection, which is nowadays widely accepted as one of the leading biological explanations. This theory suggests that species of any kind live on by means of the healthy and strong. Therefore, species containing weak traits, genes or other characteristics will eventually die out, and the better traits, genes and characteristics will reproduce and live on.

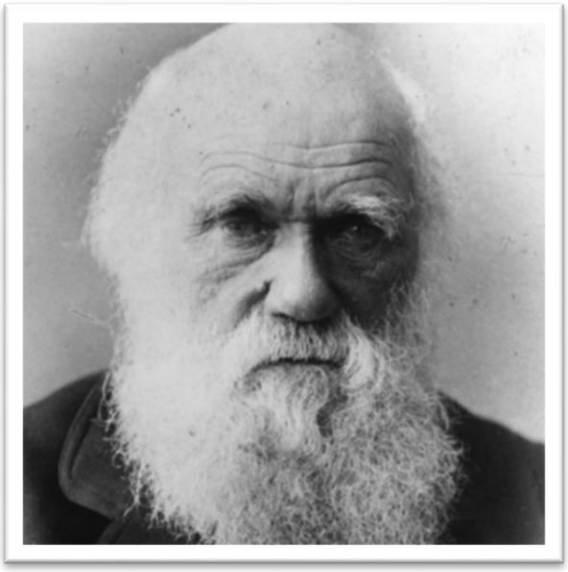
Figure 4: Charles Darwin – The founder of the Theory of Natural Selection

This theory has great implications in search and optimisations, as solutions to a problem may be treated as organisms within a given species which may reproduce and carry down their traits, or die out and be forgotten. Therefore in Genetic Algorithms, a population is defined which is constantly evolving, with the stronger living on. Three key components make up this search and optimisation approach: selection, reproduction and mutation. Selecting optimal solutions to live on to the next generation can prove vital, keeping good ‘genes’ within the gene-pool. Crossover (reproduction) of selected solutions may result in new improved solutions being formed, bettering the current population. Mutation is also a key feature in this algorithm.

Multiple mutations occur to a subset of a species in the real-world. In most cases, these mutations are harmful, resulting in that subset dying out and the mutation not carrying on. However, if a giving mutation provides an advantage to the subset, they will most likely live on and even thrive, bearing a whole new generation of organisms with this mutation. This exact same concept is implemented in Genetic Algorithms, with a miniscule percentage of solutions being mutated in the hope of making a jump out of a possible local maxima into the general maximum.

2. N-Queens Problem

We shall start by discussing the part of the program which deals solely with the N-Queens problem and does not involve any of the search and optimisation algorithms.

2.1 Data Representation

In this assignment, it was decided to represent a specific arrangement for the N-Queens problem in the form of an array, with each index storing the row number for the queen in that column. Therefore if the 3rd index in the array stored a value of 7, then a queen was present in column 4, row 8. In this manner, no two queens may lie in the same column, ensuring that one of the many rules governing the problem is not violated.

2.2 Initialisation

When initialising the queens arrangement in the **makeQueens** function, the user was asked to enter the size of the NxN board which they would like to consider. Once a decent value for the size was selected, a queens arrangement was initialised in the form of an array; as discussed above. When initialising the queens arrangement, each index in the array was populated with its own index value. Therefore index 1 stored the value 1 and index n stored the value n. In this way, no two queens lie on the same row, ensuring that another important rule is not violated. In this manner, no two queens could lie on the same row or column.

The only other possible violation is if two queens lie on the same diagonal, and this was the focus of all three search and optimisation algorithms. Therefore, by always performing swaps when searching for a new solution, and never directly changing a value in the array, the row and column conditions were never violated, making this arrangement highly efficient.

2.3 Errors

For all three approaches, queen arrangement errors were tackled in the exact same manner. The **queensTester** function, which in-turn calls the **adder**, **subtractor** and **countOcc** functions, was implemented to compute the number of violations each queen in a given arrangement was involved in. These functions performed the addition/subtraction of the index number with/from the value stored at that index.

These values where then passed to the **countOcc** function, which tested whether any two queens held the same values for these tests, signifying that they lie on the same positive or negative sloped diagonal. This function then computed the number of violations occurring at each queen instance and then returned these results. These results may later be passed onto the fitness function to test the fitness of a given queens arrangement.

2.4 Fitness Function

As is the case with the error calculation, the same fitness function was used in all three algorithms, with the fitness being based upon the number of queens violating at least 1 property of the N-Queens problem. The errors computed in the **queensTester** function are then passed on to the **fitness** function, which traversed each index testing whether that queen violated any rule. If a violation occurred, then the fitness of the arrangement was incremented. Therefore a fitness score of N signified the worst possible arrangement for a problem of size N, while a fitness score of 0 signified a solution to the given problem.

2.5 Output of Results

The results of the three algorithms were output on screen once the algorithm terminated. This was performed by means of the **printBoard** function, which printed out a representation of the solution found to the problem. The number of iterations of each algorithm was also output onscreen so as to compare the efficiency of each one.

3. Creeping Random Search

In this section we shall move through the workflow of the algorithm, understanding the approach taken to this search and optimisation problem, and the reasons behind any made decisions.

A seed arrangement was provided to the algorithm as the starting point to work upon. This seed was simply a randomised queens arrangement which acted as the starting point to this algorithm. The main body of the algorithm involves a loop which terminates only upon the finding of a correct solution.

At the start of each iteration, the current queens arrangement is tested for success. Therefore if the current arrangement has a fitness of 0, then a solution to the problem has been found and the program terminates. Otherwise, control continues and the number of iterations performed is incremented.

The next test involves testing whether to perform a local search or a global search. This test is a simple statistical test based upon the ratio of local to global searches input by the user. Based upon the outcome of this test, the control flow enters the local or global section of the algorithm.

3.1 Local Search

In the case of a local search, the first task to be performed is making a copy of the current arrangement, so that it may be compared to the newly found arrangement. Next, the hypersphere (neighbourhood) must be tested for a suitable arrangement which may be closer to the local maximum. The approach chosen for selecting a neighbouring arrangement involved singling out the queen causing the most violations in a given arrangement, selecting the queen with the least number of violations in the given arrangement and swapping them, in the hope that the new arrangement will involve less violations. The functions which perform these computations are the **badCandidate** and **goodCandidate** functions.

The **badCandidate** function works by iteratively observing each queen within a given arrangement. The greatest number of violations held by one single queen is stored as the maximum. Any queen with a violation greater or equal to the maximum is stored in the bad candidate pool. If a new queen with a worse violation record is found, then the pool is emptied, the maximum is set to the new worst number of violations and the pool is populated with the new ‘worst candidate’. This cycle is repeated until the whole arrangement has been traversed. At the end, if a single queen is left in the pool, it is returned as the ‘bad candidate’, otherwise, if multiple queens are left in the pool, one queen is selected at random and returned.

The **goodCandidate** function works in the exact same manner, however the candidates with the least number of violations are considered. The index of these selected candidates are then returned, and these queens are destined to be swapped so as to test the neighbourhood of the previous solution. Again, we see that swapping is preferred over random change of a value, so as to ensure that the row/column properties are never violated.

Following the swap of these candidate queens, the new queens arrangement has its fitness tested, and the result is compared to the previous arrangement. If the fitness of the new arrangement is better, then it is saved and replaces the previous one. Otherwise, it is discarded and a new local or global search will take place on the previous solution.

3.2 Global Search

In the case of a global search, the search space is randomly probed in the hope of finding a global maximum and ultimately a solution. A copy of the current arrangement is made so as to later compare it. The **globalSearch** function, which swaps each index with one of the other randomly selected indices, is used. Once again, we see that swapping of indices instead of directly changing values, is much more convenient in preventing the violation of row/column rules. Once a global search is performed, the fitness of the newly found arrangement is compared to the previous arrangement. If the new solution is fitter, then it is kept. Otherwise, it is discarded and the algorithm proceeds.

4. Simulated Annealing

In this section we shall explore the implementation of the simulated annealing algorithm, explaining how this algorithm was implemented and why certain parts of the algorithm were implemented how they were.

As was the case with the Creeping Random Search, a seed arrangement is required for this algorithm. This seed arrangement was once again a randomly selected arrangement in the search space by means of the **globalSearch** function.

The user is first asked to enter parameters which will manipulate the starting temperature and rate of decrease of the temperature. These parameters may be changed to achieve greater efficiency with problems of varying sizes. The main flow of the algorithm involves a while loop which iterates until a solution is found or until the temperature reaches 1, and therefore the search has failed.

As was the case with the Creeping Random Search, the first step in the loop is to test whether a solution to the problem has been found, in which case the algorithm terminates. The next phase of the algorithms is highly similar to that of the Creeping Random Search, as a bad candidate queen and a good candidate queen are selected by means of the **badCandidate** and **goodCandidate** functions. These selected queens are then swapped so as to explore the neighbouring search space.

If the newly found arrangement has a better fitness than the previous arrangement, then it replaces the previous arrangement and the next iteration of the program commences. However, this is where the similarity to the Creeping Random Search algorithm ends.

If the neighbouring arrangement found is sub-optimal, a test is performed to see whether to accept this sub-optimal solution in the hope of escaping local maxima. This test is performed by the **keepTester** function, which considers the fitnesses of the previous and current arrangement and the current temperature. The following equation is computed, allowing for a statistical test to conclude whether to accept the sub-optimal solution or not:

$$\boldsymbol{probability = e^{\frac{-(F\ suboptimal\ -\ F\ optimal)}{temperature}}}$$

This result varies according to the difference in the arrangements' fitnesses and the current temperature value. Therefore, for high temperatures, arrangements which are greatly sub-optimal are more likely to be accepted, but as the temperature decreases, the acceptance of sub-optimal arrangements decreases. Therefore for high temperatures and small differences in fitness, the equation is likely to equate to a value closer to 1, while for smaller temperatures and greater differences in fitness, the equation is most likely to equate to values closer to 0. This implemented the principle of annealing discussed in the introduction section of this paper.

The result of this equation is then statistically tested by means of a random number, which decides whether to accept the sub-optimal solution or not by means of probability. In the case of acceptance, the new arrangement is kept and the previous is discarded, and vice-versa occurs in the case of denial.

At the end of each loop iteration the temperature is decreased by the established amount and the counter storing the number of iterations is incremented. Therefore, with each iteration, the algorithm is likely to accept less and less sub-optimal arrangements.

5. Genetic Algorithms

In this section we shall discuss the implementation of the Genetic Algorithm for solving the N-Queens problem. We shall understand the work flow of this algorithm, the functions involved and the manner in which these functions were implemented.

Differently from the two previously mentioned search and optimisation algorithms, this one does not start with a single seed arrangement, but generates a seed population, made up of a number of arrangements. The size of the population was left as a user decidable parameter in this implementation, as was the parameter for the problem size, the crossover rate and the mutation rate.

5.1 Population Generation

The starting population is the set of arrangements on which this algorithm will run. This starting population of size P is made up of P random arrangements of size N. The **globalSearch** function is employed to generate these random arrangements and add them to a linked list storing the population.

5.2 Elite extraction

Elite extraction involves extracting only the elite arrangements with a greater fitness to crossover and live on in the population. This therefore involves testing all arrangements in a given population. A function similar to that of the **goodCandidate** is implemented in the **getElite** function.

A pool of the best candidates is maintained as was done in the **goodCandidate** function. However, instead of selecting one from the pool, all elements in the pool at the end of the test (currently the fittest elements) are temporarily stored. These elements are deleted from the previous population and are kept aside as the new elite population.

This good candidate extraction process is repeated multiple times, skimming off the best candidates from the pool each time. This is repeated until the keep size (calculated from the crossover percentage) is reached, creating a population of size ‘keep’ of only the fittest arrangements.

The keep size is equal to the inverse cross over ratio multiplied by the population size. Therefore if we have a population size of 10 and a crossover rate of 80%, then the keep size is 2.

The elite arrangements are then returned to be passed on to the new generation and crossed over.

5.3 Crossover

Crossover is the simulated equivalent of reproduction in the natural world, whereby two parent arrangements create a new child arrangement which contains traits of both its parents.

Since the problem in hand involves no two queens lying in the same row or column, the crossover process devised was rather tricky to design and implement. This process involves copying the first X values of the second parent node onto the first parent node, where X is a randomly selected number between 1 and N-1. However, one may realise that this would most likely violate the row/column property of N-Queens. Therefore, instead of directly copying these values, the required value is fetched from its position elsewhere in the array, and the value is swapped with the other value in the position it needs to be in:

Parent$_1$	**1**	**4**	**2**	**7**	**0**	**3**	**5**	**8**
Parent$_2$	**2**	**7**	**0**	**3**	**5**	**8**	**1**	**4**
Child after first crossover	**2**	**4**	**1**	**7**	**0**	**3**	**5**	**8**

This is repeated for all indices from 0 to X in the first parent, resulting in the new child having the first X indices identical to that of the second parent.

The **poolCrossOver** function randomly selects two arrangements from the elite pool to be crossed over until the population size is reached. The two selected arrangements are passed onto the **crossover** function which performs the above explained procedure.

5.4 Mutation

As was mentioned in the introduction to this paper, mutation is a key part of genetic algorithms, as it allows for new search spaces to be explored in the hope of escaping local maxima. The **poolMutate** function considers the newly formed population of crossed over arrangements and survivors from the previous generation together with the mutation probability decided by the user.

A simple probabilistic test is performed, so as to decide whether a given arrangement should be mutated or not. In the case of mutation being decided, the selected arrangement is passed onto the **mutate** function, which will actually perform the mutation. The mutate function randomly selects two difference indices in the queen arrangement, and performs a swap between them, rendering the arrangement mutated. This newly mutated arrangement is then reintroduced to the population.

5.5 Termination

At the end of each iteration, a test within the population is made to see whether any of the arrangements is a solution to the problem. If this is the case, the algorithm is terminated, returning the solution. Otherwise, the algorithm repeats the whole cycle again. This is repeated until a solution is found or 10,000,000 generations have been considered, in which case the algorithm terminates.

6. Expectations

In this section we shall discuss the expectations of the different search and optimisation algorithms under different conditions.

6.1 Creeping Random Search

The search space being considered is one which contains a great number of local maxima which do not lead to a solution by only a handful of violations, especially for large values of N. Therefore, although this approach is the simplest and most static of the three, I expect to see this algorithm find a solution efficiently, given its sporadic fashion in traversing the search space. With a decently high ratio of global searches, I believe that this algorithm should be able to find a successful arrangement for problem sizes all the way up to 20x20 boards.

6.2 Simulated Annealing

I believe that this algorithm should perform decently well given the right set of parameters. For a decent performance I believe that a decently high starting temperature and a slight decrease rate would allow this algorithm to overcome local maxima and jump over to a global solution. However, for large values of N, experimentation may be required to better tune the algorithm with the right parameters, as if the decrease rate is too subtle, then the algorithm will just randomly jump in and back out of global maxima.

6.3 Genetic Algorithms

I expect the parameter value for population size to have a great effect on the efficiency of the system, and I believe that the right value would allow the system to handle the problem with N values reaching around 20. Given the type of search space involved, I believe that a decently high mutation rate is required to escape local maxima, and a moderate crossover rate would allow to explore the neighbourhood of the given maxima. Also, while small populations would increase efficiency of system run-time, I expect a smaller population to return unsatisfactory results for large sized problem spaces.

7. Results

In this section we shall showcase the results of the three different algorithms running on the same sized problems with different parameters. We shall also chose a handful of different sized problems so as to compare these algorithms under different levels of strain. The following is an example of one of the command line outputs from running the program, the first being CRS, the second being SA and the final being GA:

```
Please enter the number of Queens you would like to include in this
problem:
8
Please enter the percentage of local searches (whole number):
70
Solution found in 399 cycles
 - - - - - Q - -
 - Q - - - - - -
 - - - - - - Q -
 Q - - - - - - -
 - - - Q - - - -
 - - - - - - - Q
 - - - - Q - - -
 - - Q - - - - -
Please enter the number of Queens you would like to include in this
problem:
8
Please enter the starting temperature: 80000
Please enter the decrease rate of the temperature: 0.998
Solution found in 784 cycles
 - - - - Q - - -
 - - - - - - Q -
 Q - - - - - - -
 - - - Q - - - -
 - Q - - - - - -
 - - - - - - - Q
 - - - - - Q - -
 - - Q - - - - -
Please enter the number of Queens you would like to include in this
problem:
8
Please enter the population size to maintain: 20
Please enter the cross over percentage (1.0 - 99.0): 80
Please enter the mutation probability percentage (0.1 - 99.0): 2
Completed in 2451 generations.
 - - - - - Q - -
 - - Q - - - - -
 Q - - - - - - -
 - - - - - - Q -
 - - - - Q - - -
 - - - - - - - Q
 - Q - - - - - -

 - - - Q - - - -
```

The remainder of the results shall be presented in a table format. Given that the randomly generated seed function may provide advantages/disadvantages to algorithms when testing, all cycle results displayed are the average of two runs with those parameters:

7.1 Creeping Random Search

Size of Problem	Percentage of Local Searches	Average number of cycles until correct solution found
8	70%	214
8	90%	701.5
12	70%	43,683.5
12	90%	41,432
15	70%	229,856
15	90%	1,014,697
20	70%	53,187,636
20	90%	69,405,353
25	70%	N/A
25	90%	N/A

7.2 Simulated Annealing

Size of Problem	Starting Temperature	Temperature Decrease Rate	Average number of cycles until correct solution found
8	80000	0.998	128.5
8	80000	0.98	117.5
12	80000	0.998	959
12	80000	0.99	N/A
15	8000000	0.998	N/A
15	8000000	0.99	N/A
20	100000000	0.998	N/A
20	100000000	0.99	N/A
25	10000000000	0.998	N/A
25	10000000000	0.99	N/A

7.3 Genetic Algorithms

Size of Problem	**Population Size**	**Crossover Rate**	**Mutation Rate**	**Average number of cycles until correct solution found**
8	20	80%	3%	57.5
8	100	95%	1%	88
12	20	80%	3%	44,595
12	100	95%	1%	31,117.5
15	20	80%	3%	549,488.5
15	100	95%	1%	494,105.5
20	20	80%	3%	N/A
20	100	95%	1%	N/A
25	20	80%	3%	N/A
25	100	95%	1%	N/A

8. Conclusion

Following observations of the results in the previous section. It may be concluded that the algorithm that performed best with the given problem sizes and method of implementation is the Creeping Random Search. While all other algorithms failed to compute solutions for a 20 x 20 board, the Creeping Random Search algorithm managed to compute a solution in roughly 53 million steps. Given 2.78×10^{33} possible arrangements exist, this algorithm managed to provide a solution while only probing 1.91×10^{-24} percent of all arrangements. However, for smaller problem sizes, this algorithm suffered by a slight margin of inefficiency. This, however, has negligible effects on the run-time of the program.

9. Discussion

In this section we shall discuss the outcomes of this assignment. We shall individually assess the performance of each algorithm together with their shortcomings. We shall then talk about and compare the algorithms as a whole.

9.1 Creeping Random Search

Observing the results of the creeping random search, we may compare the effects of the different parameters on the efficiency of the algorithm. We may observe that in almost all cases, a higher global search ratio of 30% achieved better results. This meets the expectations presented earlier, which reinforce the idea that the search space is made up of a number of local maxima which are not global. Therefore, by increasing the frequency of random jumps, we are more likely to jump into a global maxima and find a solution. I believe that this algorithm is the most suited for this sort of problem, given that it will always jump around and test a totally new area every interval.

9.2 Simulated Annealing

Following observations of the results for the Simulated Annealing algorithm, it is apparent that the algorithm did not run efficiently for the given problem, even unable to search for solutions on a 15x15 board or a 12x12 board with slightly adjusted parameters. This is due to the fact that simulated annealing is suited for situations where one (or very few) solutions (global maxima) exist. This is due to the fact that the global maxima is hopefully found at the start of execution and is then refined to find the solution as the temperature decreases. However, the search space in hand contains multiple regions which appear to be global maxima but contain no solution. Therefore, the algorithm probes around, jumping from maxima to maxima, but is unable to home in on a maxima with a solution. Attempts to introduce a greater decrease rate had no noticeable effects on the final result.

A possible shortcoming is the fact that the differences in fitness levels may hinder the acceptance of sub-optimal arrangements. Since the fitness of an arrangement may range from 0 to N (where N is the size of the arrangement), then the probability of selecting a sub-optimal arrangement, however bad it is, is rather high considering that the numerator of the exponential may only be as large as –N. Whereas if the fitness ratings ranged between 0 and M (where M is much larger than N), then very sub-optimal arrangements would have a very limited probability of being accepted.

9.3 Genetic Algorithms

Observing the results for genetic algorithms, it is noticeable that the algorithm yielded decent results for problem sizes up to 15. This is expected, given the computationally heavy functions involved in this algorithm, together with the fact that each function is applied to each member of a population. It is also noticeable that a bigger population allows for more efficient searches for a solution, as a rich gene pool could allow for more areas in the search space to be probed at the same time. We also see that greater crossover rates yielded better results, due to the fact that each maxima was more thoroughly searched before it was dismissed or exited.

Possible shortcomings for this algorithm include the fact that crossover is performed on those arrangements which shall live on to the next generation, possibly resulting in a lack of diversity within the gene pool. This could heavily rely on mutations to jump out of local maxima. Different possible crossover methods may also yield better results, with different ways of altering values to avoid violations.

9.4 Comparison

Observing the results of the three different algorithms as a whole, it is clear that Creeping Random Search is the most effective of them all in finding solutions for large problem sizes. As previously explained, this is due to the fact that a greater number of random jumps occur. For smaller sizes of N, such as 12 or less, we see that Genetic Algorithms work better, as fewer iterations are required to search for the best solution. However given the minute number of iterations involved, this will not have a noticeable effect on efficiency. This experimentation has also proved that Simulated Annealing is not a suitable algorithm for such a search space with multiple maxima/minima which may not be solutions.

I believe that the representation of queen arrangements and the manner in which errors and fitnesses were tested was highly efficient and easy to work upon. Other possible implementations may include the generation of a queen class whereby a queen is assigned a row and column, and these values may be changed to find a solution. However, the method implemented implicitly avoids row and column violation, which I believe, in itself, is already a great positive.

References

Figure 1: http://articles.leetcode.com/wp-content/uploads/2012/03/8-queens.png

Figure 2: http://www.turingfinance.com/wp-content/uploads/2014/03/Simulated-Annealing-for-Portfolio-Optimization1.gif

Figure 3: http://orig01.deviantart.net/6684/f/2009/195/3/f/sword_smithing_cutlace_pic_1_by_animeghost66.jpg

Figure 4: http://a4.files.biography.com/image/upload/c_fit,cs_srgb,dpr_1.0,h_1200,q_80,w_1200/MTE5NDg0MDU0OTM4NjE3MzU5.jpg