|  |  |  |
| --- | --- | --- |
| **Logo_2deckBW70mm** |  | **ASSESSMENT COVER SHEET** |

**DATE RECEIVED:**

**Postmark:**

**Please complete section below:**

**Course Code:**2802ICT

**Course Name:**Intelligent Systems

**Due Date:**   25/03/2018 **Assessment Item #:**1

**Course Tutor:**   Alan Liew

**Course Convenor:**   Alan Liew

**Study Period or Session 1**

**Please provide your STUDENT NUMBER here:**

s5084150

**Student Name:**   Carl Humphries

**Address:**

**ACADEMIC INTEGRITY DECLARATION**

Breaches of academic integrity (cheating, plagiarism, falsification of data, collusion) seriously compromise student learning, as well as the University’s assessment of the effectiveness of that learning and the academic quality of the University’s awards. All breaches of academic integrity are taken seriously and could result in penalties including failure in the course and exclusion from the University.

Students should be aware that the University uses text-matching software to safeguard the quality of student learning and that your assignment will be checked using this software.

I acknowledge and agree that the examiner of this assessment item may, for the purpose of marking this assessment item:

reproduce this assessment item and provide a copy to another Griffith staff member; and/

submit this assessment item to a text-matching service. This web-based service will retain a copy of this assessment item for checking the work of other students, but will not reproduce it in any form.

Examiners will only award marks for work within this assignment that is your own original work.

I, hereby certify that:

except where I have indicated, this assignment is my own work, based on my personal study and/or research.

I have acknowledged all materials and sources used in the preparation of this assignment whether they be books, articles, reports, lecture notes, or any other kind of document or personal communication.

I have not colluded with another student or person in the production of this assessment item unless group work and collaboration is an expectation of the assessment item.

this assignment has not been submitted for assessment in any other course at Griifith, or at any other University or at any other time in the same course without the permission of the relevant Course Convenor.

I have not copied in part or in whole or otherwise plagiarised the work of other students and/or other persons.

I haven’t made this piece of work available to another student without the permission of the Course Convenor.

Providing this declaration falsely is considered a breach of academic integrity.

**I have retained a copy of this assessment item for my own records.**

Acknowledged by:   Carl Humphries Date:25/03/2018

(insert name here)

Where the item is submitted electronically Clicking “I Agree” constitutes an electronic signature for the purpose of assignment declaration compliance.

**STUDENT CONSENT**

*(to be completed by the student before their essay, assignment or other work is uploaded to an internal/online learning University website or used for the purpose of moderation (not to be used if there is to be public access to the work)*

At Griffith the use of assessment exemplars by academic staff is encouraged to inform students’ understanding of the performance standards associated with learning and achievement in the course. An assessment exemplar is an authentic example, actual sample or excerpt, of student work that has been annotated to illustrate the ways in which it demonstrates learning, achievement and quality in relation to the intended learning outcomes (including graduate outcomes) for the course. Assessment exemplars may be made available in a range of ways. In order to collect assessment exemplars students are asked to consent, on every assessment item submitted, for their work, without disclosure of the contributor’s identity, to be used, and reproduced as an assessment exemplar for standard setting and moderation activities.

I acknowledge that for the purpose of standard setting and moderation activities the examiner of this assessment item may wish to store, reproduce, annotate, and communicate my work to others, including future students, without disclosure of my identity.

I consent to my Work, N-Queens without disclosure of my personal details, being stored, reproduced annotated and communicated within the University’s secure online learning environment.

I do not consent to my Work, being stored, reproduced annotated and communicated within the University’s secure online learning environment.

Acknowledged by: Carl Humphries Date:25/03/2018

(insert name here)

Where the item is submitted electronically Clicking “I Agree” or “I do Not Agree” constitutes an electronic signature for the purpose of student consent.

**Extension Requests:**

Assessment Item Number:      Due Date:

Extension Granted:  Y  N Amended Due Date:

Extension Approval Number:  

Extension Approved by:       Approval Date:

**Examiner’s Use Only: Tick Where Appropriate:**

Name:          Submitted late without extension – returned to student to seek extension /special assessment.

Mark Given:          Suspected plagiarism: referred to the Course Convenor.

Second Examiner:          Returned to student to be resubmitted by:  
(if required)       (date) for the following reasons:

Mark Given:     

**Examiner’s Comments:**

N-Queens

# Abstract

The aim of this report was to test and compare four different search optimisation algorithms, Hill Climbing, Random Restart Hill Climbing, Simulated Annealing and Genetic Algorithm. The algorithms were tested on a range of different sizes (N=4, 8, 20, 50 and 100) excluding 50 and 100 for Random Restart due to time constraints (took too long to get valid data). All tests were averaged over 30 separated tests to give more accurate results and show if there was any randomness present (the variation of the results).

The algorithms were compared on efficiency, effectiveness and convergence speed. Simulated Annealing seemed to outperform the others due to it wasn’t as reliant on the cost function. It also appeared to be less random with the results showing the standard deviation was quite low.

The one recommendation would be to use a constrained state space where only one queen per row and per column. And all moves were defined by swapping two queens (row positions) but keeping their own column.

## Terminology

* Eval(): the cost function used to count the clashes with queens. Sometimes called cost.
* Move: A move is just moving one of the queens up or down to a new valid position.
* State: A configuration of the Board where each column has only one queen.
* Neighbour: A state that can be reached by only one move.
* N,n: the size of the problem. The board is NxN and N queens needed to be placed.
* Peak: local maximums (extremum), ridges or plateaux.
* Pool size: the size of the limited population for genetic algorithm.

Table of Contents

[Abstract 4](#_Toc509761407)

[Terminology 4](#_Toc509761408)

[Introduction 6](#_Toc509761409)

[The N-Queens Problem 6](#_Toc509761410)

[Analysis 7](#_Toc509761411)

[Data Structures 7](#_Toc509761412)

[Hill Climbing 8](#_Toc509761413)

[Random Restart Hill Climbing 9](#_Toc509761414)

[Simulated Annealing 11](#_Toc509761415)

[Genetic Algorithm 13](#_Toc509761416)

[Comparison 16](#_Toc509761417)

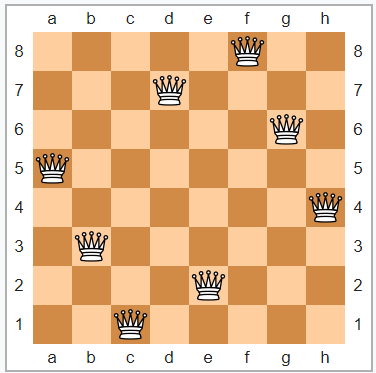
[Conclusion 18](#_Toc509761418)

[Recommendations 19](#_Toc509761419)

[Reference List 20](#_Toc509761420)

# Introduction

## The N-Queens Problem

The N-Queens problem is a widely covered topic that is used quite frequently for testing the efficiency and effectiveness of a search optimisation algorithm. The N-Queens problem is defined as; a chessboard state of N rows and N columns, where the aim is to put N number of queens (Queens from a chess set) so that they do not lie on the same diagonal, the same row or the same column (non-attacking). 

N-Queens solution of size 8, by unknown, Wikipedia, Public Domain.

The queen’s problem from a technical perspective is can be defined by

* **initial state**, (a configuration of the board where queens are placed randomly or pre-determined, in their own column)
* **action** or move (moving one queen up or down in its own column),
* result of the action **successor** or neighbour, (a configuration of the board that is one action away)
* **cost**, (total number of queens under attack, on the same diagonal, the same row or the same column)
* **goal state**,(if the cost is equal to 0 in the current state the current state is the goal state)

While there is no exact pattern for how many solutions, there seems to be an exponentially high increase in solutions as you increase N. From this pool of solutions there is lot less unique (fundamental) solutions. Unique solutions are the same if they are flipped transposed or rotated.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***n*:** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **...** | **24** | **25** | **26** | **27** |
| **fundamental:** | 1 | 0 | 0 | 1 | 2 | 1 | 6 | 12 | 46 | 92 | ... | 28,439,272,956,934 | 275,986,683,743,434 | 2,789,712,466,510,289 | 29,363,791,967,678,199 |
| **all:** | 1 | 0 | 0 | 2 | 10 | 4 | 40 | 92 | 352 | 724 | ... | 227,514,171,973,736 | 2,207,893,435,808,352 | 22,317,699,616,364,044 | 234,907,967,154,122,528 |

Table showing solutions, by unknown, Wikipedia.

# Analysis

The range of N tested was (4, 8, 20, 50, 100). Most experimental testing was done on N=20 because smaller boards were too quick to see any change when altering different parameters and factors, and N>20 was too slow to get results quick enough. All test results were averaged over 30 tests and stored in an Excel spreadsheet. Most graphs were generated with SPSS graphing tools from this data.

## Data Structures

The program uses a very basic layout to make things easier, it is written in python using 3.6 standard packages. Every queens’ position is repressed as [x,y] where x is the index of a list and y is the value, this if often referred as a complete-state formulation (Russell & Norvig 2009, p. 122) forcing a smaller state space where no two queens can be in the same column.

### Board Representaion

The Boards are setup in classes so that they store both the board data and the cost. This allows the addition/alteration of “\_\_str\_\_” to print the board and cost in an intelligible manner. As well as the “\_\_lt\_\_” so it can perform .sort() on a list of Boards. On initialisation of a Board object, the cost function is called (Eval()) and the result is stored in the object.

### Cost Function (Eval())

The cost function is one of the most important functions in the program. It is used in all the algorithms, and is how each algorithm determines if they have solved the problem, or if they are getting closer to the goal state. The cost function is quite basic but has time complexity of O(n2).   
The basic explanation of this is for every column get the queen in that column then compare it to every other queen checking the relative diagonals and the rows. Basically, checking everything to the right of the queen in question. Below is the pseudocode:

cost <- 0  
FOR i<-[0 to N-1]   
 get relative diagonals  
 FOR j<-[i+1 to N]  
 if queen in i column and j column is on same diagonal  
 cost<-cost+1  
 if queen in I column and j column is on the same row  
 cost<-cost+1  
END FOR  
return cost

### Other

There are other functions inside the program, that are used for each algorithm but the explanations are documented in the code. And because they are only used for specific algorithms they are not going to be described here.

## Hill Climbing

### Explantion

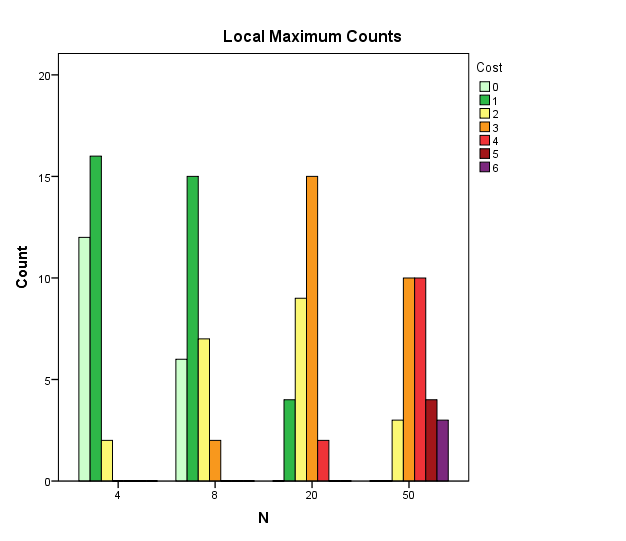
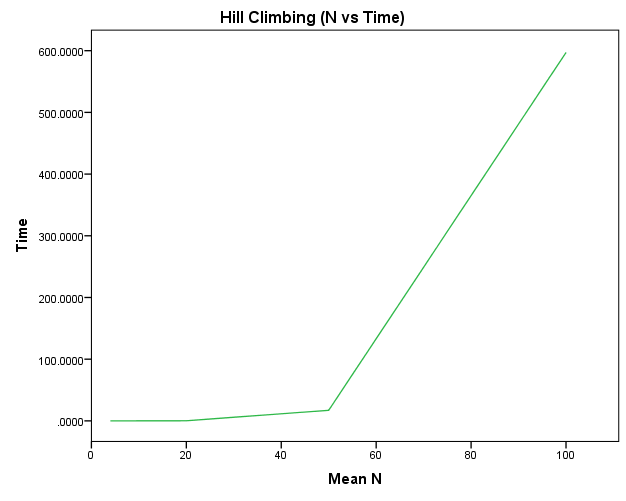
Hill climbing is one of the most basic approaches to solving the N-queen’s problem. It is done by looking at all the neighbouring states, and taking the one (randomly pick is there is multiple) that produces the best Eval(). A way to do this is to get the cost of all the neighbouring states and just pick the best. But doing this is exponentially more intensive the bigger the problem is. Because there is n(n-1) neighbours and each one needs to be evaluated at O(n2). Below is the pseudocode that was implemented:

X<-random initial state  
Neighbours<-get\_best\_neighbours(X)   
WHILE Eval(X) > goal\_state AND Neighbours is not empty:  
 X<-random neighbour from Neighbours  
 Neighbours<-get\_best\_neighbours(X)  
END WHILE

### Testing Results

#### Efficiency

The first thing tested was to see how fast the algorithm was. The algorithm was tested on the range of N, and the average times were recorded over 30 runs. As N rose, the results seemed to be exponentially larger. The testing was stopped at 100 due to it taking ~5 hours to get results. (See figure on the left)



#### Effectiveness

The other thing that was tested was to see if it could solve the problem to the goal state “0”. But using this algorithm often causes it to hit local maximums (extremum), ridges or plateaux. This is when there is no neighbour that is better than the current state and it has not yet reached the goal state. From testing it was apparent that the larger the N, the further it was away from the goal state (See figure on the right).

## Random Restart Hill Climbing

### Explantion

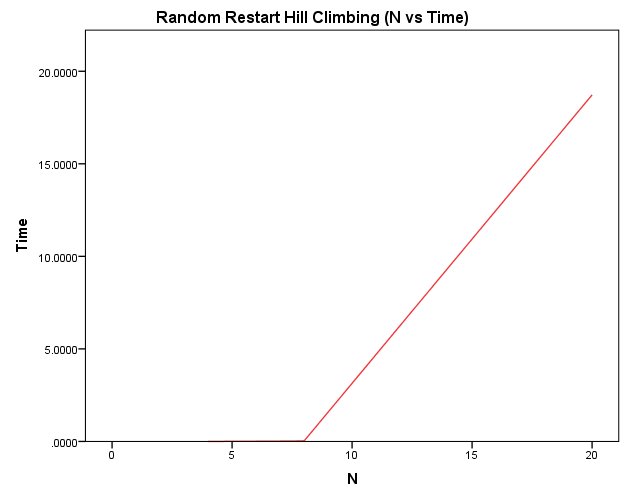
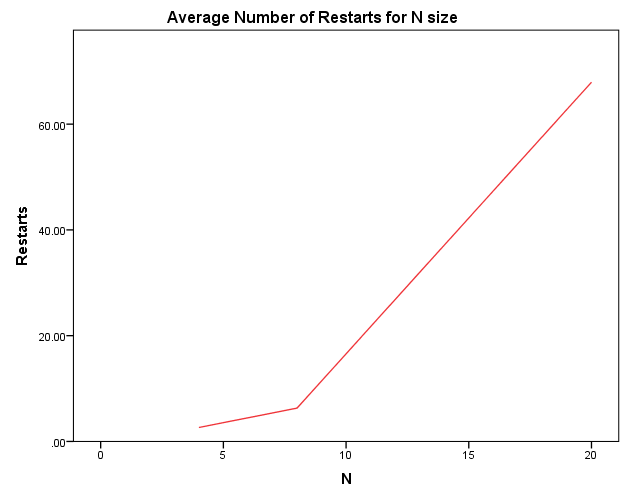
While hill climbing is generally fast, there is a large chance of hitting a “peak” around 86% for N=8. Because of this, a method was created to basically follow “If at first you don’t succeed, try, try again.” (Russell & Norvig 2009, p. 124). This is done by, once the Hill Climbing algorithm hits a peak restart from a random state. Do this until the goal state has been found. Pseudocode below, which is a slight altered Hill Climbing algorithm:

X<-random initial state  
Neighbours<-get\_best\_neighbours(X)   
WHILE Eval(X) > goal state:  
 WHILE Neighbours is not empty:  
 X<-random neighbour from Neighbours  
 Neighbours<-get\_best\_neighbours(X)  
 END WHILE  
 IF Eval(current state) > goal\_state:  
 X<-randomise new current state  
 Neighbours<-get\_best\_neighbours(X)  
END WHILE

### Testing Results

#### Efficiency

While this method is quite valid. It was found in testing that it took too long to find a solution with N>20 (shown in the figure on the right). This may be due to the problem stated in Hill Climbing, where it takes too long to evaluate the cost of all the neighbours. Because of this slow speed It wasn’t practical to use Random Restart on larger problems so they were not included. This was apparent from the number of restarts it took the larger the problem was. (See figure on left) The range of N was 4, 8, 20.

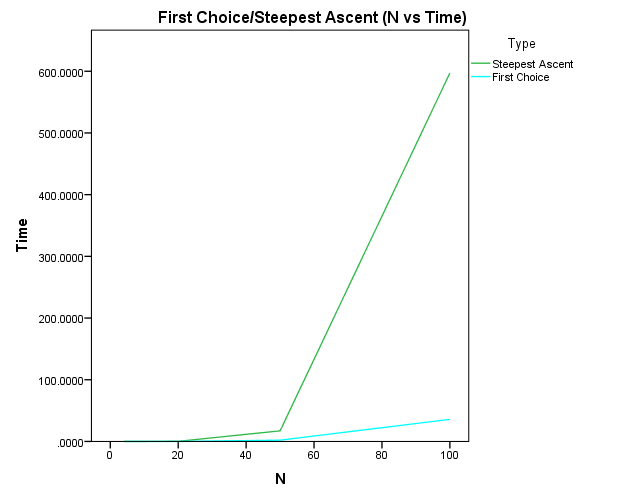


#### Effectiveness

Because Random Restart is just using Hill Climbing to solve the solution over and over until goal state is found. The effectiveness was 100% of the time it got to the goal state since it was the ending condition.

### Experimentation (Both Random Restart and Hill Climbing)

The largest problem that was found in testing (both Hill Climbing and Random Restart) was that the evaluation of the queens took too long to check all neighbours. The first method that was tested was to try combat this was First-choice hill climbing. This is where the algorithm generates a random neighbour and checks if it is better. This lowered processing times quite drastically (See figure below). This method wasn’t used in comparison testing since the aim of this report was to focus on Steepest Ascent.



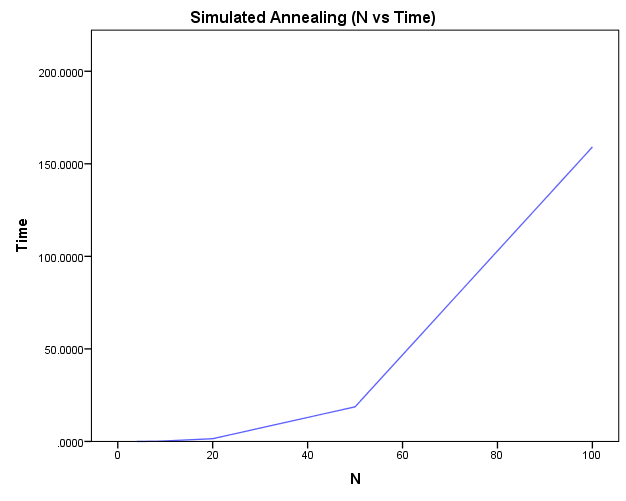
## Simulated Annealing

### Explantion

Simulated Annealing was derived from the engineering technique Annealing. Where metal is heated up to a high temperature then slowly cooled to give better results. It is sort of a combination between randomly moving queens and First Choice Hill climbing. The basic process is pick a random neighbour, if the Eval() of that neighbour is better, take that as new current state. Otherwise accept with probability using the formula given by (P=e-(Eval(X’)-Eval(X))/T). Without going into the math behind this, as T approaches infinity regardless of the change in cost the move will be accepted (random move). But the closer T is to 0 the more “greedy” (Only taking better moves) it becomes. The difference in the Eval() also determines if the move will be accepted. The worst the move is the less chance it gets of being accepted.

Using this formula, the algorithm can slowly decrease T to converge at a constant speed, in aims to converge to a solution. This is done by, after K times of checking moves and accepting some. T is multiplied by a number (Alpha) between 0.0-1.0 (exclusively). This is then repeated until T is too small. If Alpha is closer to 1 the slower T will decrease. This is also the case if K is larger. Pseudocode below, showing the algorithm with variable parameters:

X<-random initial state  
WHILE Eval(X) > goal\_state:  
 FOR K times:  
 X’<-random valid neighbour  
 IF Eval(X’) < Eval(X):  
 X<-X’  
 ELSE IF e-(Eval(X’)-Eval(X))/T > random number 0.0 to 1.0:  
 X<-X’  
 END FOR  
 T<-T\*alpha  
END WHILE



### Testing Results

The parameters used in these tests where what was found to be most effective with the algorithm created are:

* K=100
* T=1.0
* Alpha=0.99

#### Efficiency

This algorithm was very efficient having an almost linear graph (see figure on previous page). This was probably due to the same reason the Hill Climbing slowed down. This is due to it picking a random neighbour, so it doesn’t always have to evaluate every possible neighbour. It was easily manageable to collect data for the range that was used. Taking only 4800 seconds to get 30 tests at N=100.

#### Effectiveness

The effectiveness for Simulated Annealing was quite amazing given it was so quick. There was 8/30 times on N=100 it didn’t get to goal state. But this was due to it stopping because T was too small (it was coded this way to avoid large calculation errors). If T was given more time it would have been able to find a solution without too much overtime.

### Experimentation

The parameters were altered, below are the results in tables. The main thing that affected the speed was how quickly the temperature reached the min threshold. Either K was small (checks before T is cooled by alpha) or Alpha was closer to 0. There is a fine line for efficiency and effectiveness therefore the end parameters that were chosen were K=100 T=1.0 and Alpha=0.99 for all the other tests.  
These values were constant and only thing that changed were the parameters specified in the tables.

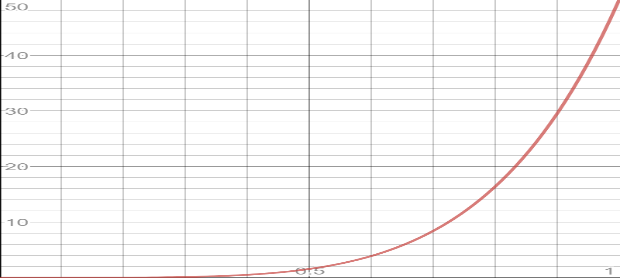
|  |  |  |
| --- | --- | --- |
| K | Time | Notes |
| 1 | 0.088827 | Failed to reach goal state average cost of 1.07 |
| 10 | 0.329425 | Failed to reach goal state average cost of 0.03 |
| 100 | 1.420315 | Didn't fail |
| N\*2 | 4.706395 | Didn't fail |

|  |  |  |
| --- | --- | --- |
| T | Time | Notes |
| 1 | 1.420315 | Didn't fail |
| 10 | 4.486413 | Didn't fail |
| 100 | 7.996348 | Didn't fail |
| N | 5.639654 | Didn't fail |

|  |  |  |
| --- | --- | --- |
| Alpha | Time | Notes |
| 0.8 | 0.269838389 | Failed to reach goal state average of 0.2 |
| 0.9 | 0.310860856 | Didn't fail |
| 0.99 | 1.42031521 | Didn't fail |
| 0.999 | 10.54973811 | Didn't fail |

## Genetic Algorithm

### Explantion

The Genetic Algorithm was created to simulate evolution and genetics. In its most simples form it boils down to survival of the fittest, where individuals who have better properties (better fitness/cost) pass down genes to the next generation. The algorithm does this by sorting a random population (initial population) by lowest cost first. Then using a selection formula, it picks two semi random parents (where the better parents are more likely to be picked). The formula is simple, it is just pool size times a random number 0.0 to 1.0 with the exponent of the selection factor. The higher the selection factor the more likely better parents are chosen (int(pool\_size\*xselection\_factor) Where x is a random number between 0.0 and 1.0). This gives an index to access the parent from the population. (see figure below showing the selection curve)   


Shows chance (X-axis) of selecting parent at index (Y-axis). (pool size of 50 and selection factor 5)

Once two parents are picked they need to reproduce. This is done algorithmically by picking a cross over point and cutting parents in half at that point. Then from there two children are created by swaping the halves of the parents. Now that there is two new children, both those children have a chance of being mutated. A mutation is just taking a random queen and moving it randomly up or down.

This process is done until enough children are created to fill a new population of the same size. This new population then becomes the main population and then repeated until one child is born with the goal fitness. Pseudocode below showing how the populations are changed with each generation:

Population<-generate\_initial\_population(pool\_size, N)  
sort population  
WHILE population[0] > 0 and generations < max\_generations:  
 FOR pool\_size/2 times:  
 X, Y<-get two semi random parents (using selection formula)  
 pick random cut point  
 child1, child2<-reproduce(X,Y)  
 IF random number 0.0 to 1.0 < mutate\_chance:  
 randomly mutate child1   
 IF random number 0.0 to 1.0 < mutate\_chance:  
 randomly mutate child2  
 add child1, child2 to new\_population  
 END FOR  
 population<-new\_population  
 sort population  
END WHILE

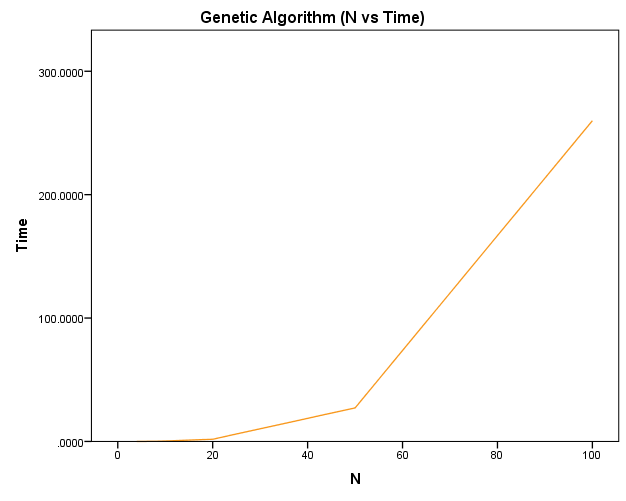
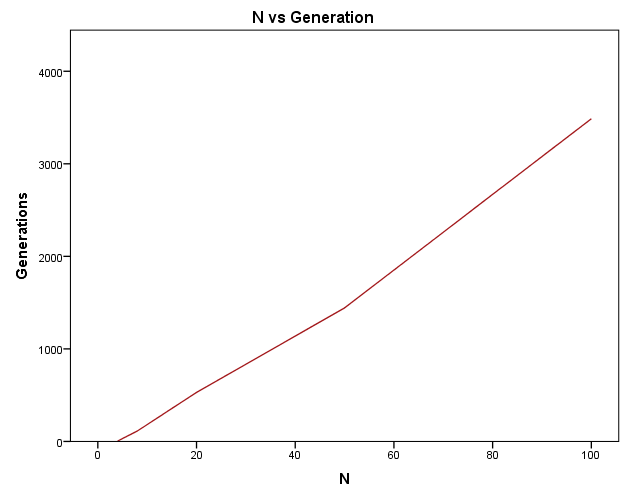
### Testing Results

The parameters used in these tests where what was found to be most effective with the algorithm created are:

* Mutation chance=0.8
* Pool size=50
* Selection facto=5

#### Efficiency

The Genetic Algorithm gave decent testing results. As N grew it seemed to get drastically slower. (see figure on left) Running anything bigger than N=100 would be way too long to wait for. As N increased there was almost a linear relation with the increase of the generations. (see figure on right) This seemed to be the reason it took longer the larger the problem was. But it could also could have been the cost function takes longer the larger N is.

#### Effectiveness

Genetic was 100% effect there was no point where it could not solve it and it seems to be its strong point. The algorithm did have a cap of 10,000 generations and a N that was >250 may have had some inconsistencies because of the trend in generation size. Time was more the limiting factor shown from the testing.

### Experimentation

The selection factor, pool size and mutation chance were altered to see how they effected the algorithm. The same parameters were used as in the testing section. Results are in tables below showing averages for 30 tests with altered parameters. The main thing that was found from these results was that:

* If pool size was bigger it just took longer to finish, while the average generations decreased. This seemed to slow down though with a larger pool size (approaching ~400).
* There was an increase in performance the higher the selection factor was but also decreasing after 4. (5 was still used throughout testing because at larger problems it seemed to do better)
* The use of Genetic algorithm on this specific problem (N-Queens), seemed to need a higher mutation chance as it seemed to rely on it to get to the goal state. If the mutation chance was too small it would have trouble getting to the goal state.

|  |  |  |
| --- | --- | --- |
| Pool Size | Time (sec) | Notes |
| 20 | 0.92688 | 708.7 average generations |
| 50 | 1.597479 | 492.5 average generations |
| 100 | 2.966867 | 456.3 average generations |
| N2 | 10.81236 | 413.6 average generations |

|  |  |  |
| --- | --- | --- |
| Selection Factor | Time (sec) | Notes |
| 2 | 2.115309 | 638.3 average generations |
| 3 | 1.622928 | 492.2 average generations |
| 4 | 1.337468 | 403.4 average generations |
| 5 | 1.555264 | 478.1 average generations |

|  |  |  |
| --- | --- | --- |
| Mutation chance | Time (sec) | Notes |
| 10% | 24.13689 | ~53% fail rate at 7698.5 average generations |
| 50% | 4.86313 | 1357.5 average generations |
| 80% | 1.331962 | 409.5 average generations |
| 90% | 1.239717 | 327.6 average generations |

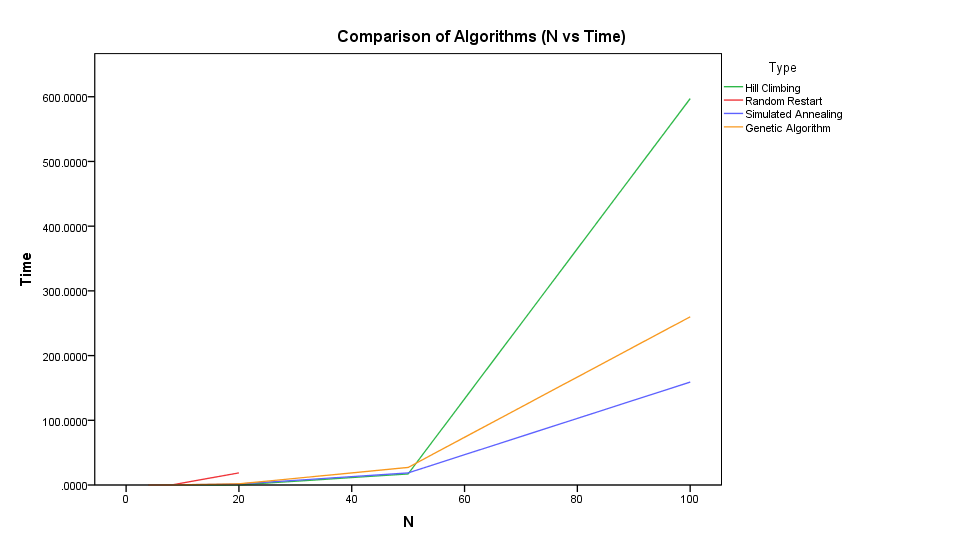
## Comparison

Because the algorithms were quite different and had their own strong points. They were compared over a few different key areas. These areas included:

* Efficiency: How long did it take to finish?
* Effectiveness: Did it get to the goal state and if not how far from it?
* Converging speed: How fast does it decrease the cost?

### Efficiency

All the algorithms were tested on the same computer using only one python interpreter to allow similar environmental factors. The table below shows N vs Time for all the algorithms. Note that Random Restart did not include tests for N=50 or 100. Due to, both the number of restarts needed and time to finish one Hill Climb increase drastically as N increased. Making it take too long to get any data.

It is quite apparent that Simulated Annealing was better as N increased. Genetic was close but started branching out in larger N’s. It seemed that they were all really close with no noticeable difference with N=4 or 8. But as the graph shows Random Restart quickly broke off at N=20. It is hard to see but for N<=20, but Hill Climbing was the fastest by a small amount. 

The table below shows how consistent the times were for a given N. Both the Random Restart and Genetic Algorithms were quite random due to standard deviation and mean being so close. So, the given times weren’t always accurate for these algorithms hence why average of 30 tests were used.

Table showing mean and standard deviation for each algorithm over 30 tests.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | Hill Climbing (sec) | | Random Restart (sec) | | Simulated Annealing (sec) | | Genetic Algorithm | |
| Mean | Std. Dev | Mean | Std. Dev | Mean | Std. Dev | Mean | Std. Dev |
| 4 | 0.000133 | 0.0003458 | 0.000401 | 0.0004994 | 0.000718 | 0.0005531 | 0.001736 | 0.0025435 |
| 8 | 0.002101 | 0.0006741 | 0.025071 | 0.0266090 | 0.057840 | 0.0314991 | 0.100113 | 0.1096118 |
| 20 | 0.174353 | 0.0352897 | 18.726449 | 18.3209268 | 1.483057 | 0.2465537 | 1.783861 | 2.0652446 |
| 50 | 17.1365 | 2.35657 | N/A | N/A | 18.710125 | 7.5127565 | 27.187464 | 17.9617912 |
| 100 | 597.1689 | 66.91822 | N/A | N/A | 159.076205 | 52.7818018 | 259.946073 | 127.6198648 |

### Effectiveness

It is quite apparent that Hill Climbing by itself is not really a viable option because it has such a low chance of getting the goal state. Only at a small N it found a solution and even then, it was a fairly low chance. Both Genetic and Random Restart always found a solution due to it being the terminating condition. If Random Restart was quicker it would still have had 100% effectiveness because of this.

Simulated Annealing failed at high N due to the temperature cooling too fast and hitting the threshold (T\_min) too early. This was tested to be true but an error catch method was used for the small calculations. Therefore, it wasn’t included in the testing data.

Overall all methods apart from Hill Climbing were valid in effectiveness to this problem. With small parameter tweaks Simulated Annealing would have been 100% effective.

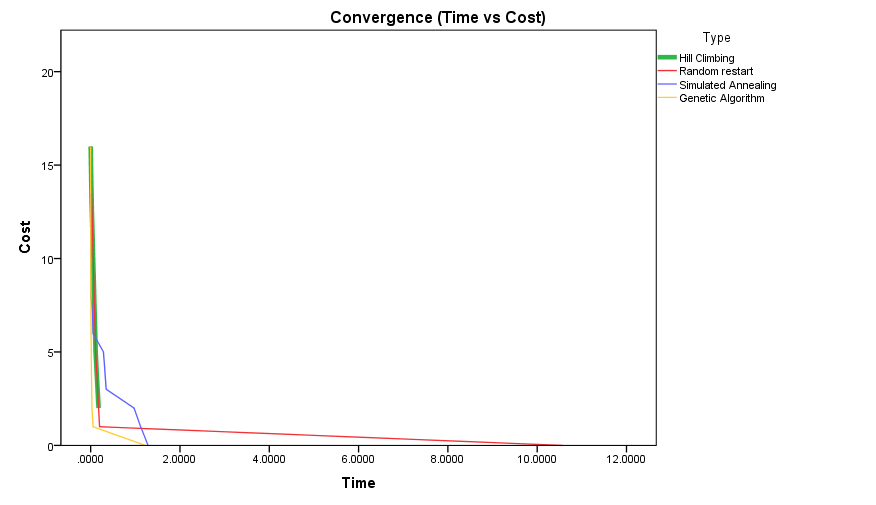
Table showing percentage of the algorithm reaching goal state.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Hill Climb (%) | Random Restart (%) | Simulated Annealing (%) | Genetic Algorithm (%) |
| 4 | 40 | 100 | 100 | 100 |
| 8 | 20 | 100 | 100 | 100 |
| 20 | 0 | 100 | 100 | 100 |
| 50 | 0 | N/A | 100 | 100 |
| 100 | 0 | N/A | 73.3 | 100 |

### Converge speed

All the algorithms were run on the same N=20 Board. Every time the cost was lowered the time was taken. This produced this graph showing the convergence speeds. Because Hill Climbing didn’t reach the goal it rarely fully converges. Because of this Random restart was serval times slower than Simulated Annealing and Genetic Algorithm.

Because of Genetic Algorithms nature of randomness, it converged quicker but took longer to completely converge to the goal state. If it’s compared to Simulated Annealing it is apparent that Simulated Annealing is much more of a constant drop. Allowing it to converge nicely on larger N’s.



# Conclusion

While these algorithms had their strengths, it was quite clear Simulated Annealing was more effective, efficient and had a decent converge speed. Because of the nature of the cost function being O(n2) it halted performance for the Hill Climbing techniques. If the cost function was faster or instant Hill Climbing and Random Restart would have been probably more efficient than the other algorithms. It should be noted that Simulated Annealing and the results from testing were very reliable and they give a good indication on the performance, because the data wasn’t so randomly spread.

# Recommendations

One major thing that was tested and could improve the performance of the different algorithms. Is to shrink the state space more than just limiting the queens to one column each. It could be altered so queens cannot lie in the same row or column, drastically decreasing the amount of states the problem has. One problem with this though is that the moves needed to be altered so they don’t break this new constraint. This can be achieved by swapping two queens row positions inside their column, rather than just moving one up or down in its row. By swapping, the algorithm maintains the constraint while removing a lot on invalid states. With the small testing done, it almost increased performance by 10-fold and seemed to be more effective. One drawback with this was that is difficult to use this with the Genetic Algorithm because of the nature of mutation and reproducing.

# Reference List

-Unknown 2018, *Eight queens puzzle*, viewed 20 March 2018, <<https://en.wikipedia.org/wiki/Eight_queens_puzzle>>

-Russell, S & Norvig, P 2009, *Artificial Intelligence, A Modern Approach: Third Edition*, Prentice Hall, USA.