FINAL PROJECT REPORT

N Queens with Poison cells

Problem Overview:

My problem “N Queens problem with poison” is the variation of the classical n queens’ problem. Here the numbers of valid cells where the queen can be placed are reduced by marking them as poison or invalid.

P		Q		
P			P	Q
	Q			
	P	P	Q	P
Q	P			

Figure 1: A solution for 5 queen problem with poison cell
(Q = position of queen on the board, P = Position of the poison cell)

For example in figure 1 we have been provided a solution for 5 queens with poison cell on a chess board of 5x5 where none of the queens can capture each other by moving horizontally, vertically and diagonally.

The problem is complicated by the inclusion of the poison cell which are marked P in the figure 1. In these cells the queen cannot be placed as they are marked invalid randomly before we start solving the problem. Having a few cells invalid means you have to search for a solution that does place the queen on a poison cell. And since we have randomly marked the cells as poison. There could be instances where you may not find the best solution but an approximation of the best solution. For example if we randomly initialize 10% of the cells as poison then it could be possible that an entire row and column could be covered in poison. At that point the best solution will give at least two queens who can capture each other either vertically or horizontally.

Why I chose this problem?

The classical N queen problem itself is considered a tough problem for large ‘N’ since most of the algorithms concentrate on back tracking. Having poison cells will complicate the situation with backtracking algorithms will take more computation time and with increasing the percentage of poisoned cells and queens, the algorithm is more likely to break down.

It is equally difficult with a deterministic algorithm to specify what you consider as a good and bad solution. Whether you accept queens not attacking each other but placed on poison cells

or the vice verse. Whichever option you choose your algorithm will tend to have bias towards that sort of placement and so the solution provided by run of algorithm may not be the best approximation.

How did Genetic Algorithm (GA) solved my problem?

Genetic algorithms have no such worries. You can define a fitness function as per your bias or without any bias and let algorithm take over. The algorithm creates a bunch of solution randomly and checks the extent of validity of each. The solutions from the bunch hen are chosen to be combined by any of the methods for GA crossover to define another bunch solution instance.
The above process is repeated till we get a best solution as per the fitness function. In between the process you can add random values to some extent in a solution to extend the search space. The same process has been described in detail in this report about how I went about it.

Parameters and solution instances:

There are only two parameters to the problem as per my algorithm. For how many queens do you want to solve the problem for and what percentage of cells you want to be marked as poison.

The output as per the algorithm coded is a permutation (more on that later) showing the best fit solution for position of queens, a spreadsheet with two tabs one showing the places which cells on the board are free and poison. And the other sheet showing where the queens have been placed on the board. The spreadsheet rows and columns can be modeled on lines of a chess board. The position of the queen is shown with a Q and a queen placed on poison cell is shown as Q/P.

Throughout the GA implementation the position of the poison cells is taken to be constant that is initialized at the beginning of the algorithm.

GA GENETIC PARAMETERS

Genotype:

The genotype is the abstraction of the solution at a very primitive level that is randomly initialized and checked for completeness to the solution in the GA.

- For solving the “n queens problem with poison” or even “n queens” we need a abstraction of the chess board which will be the environment. For this an ideal choice would be a 2D array.
- However with a 2D array the number of computations would increase greatly along with the complexity of initializing the population and a complex fitness calculation.
- Thus have chosen a better alternative of substituting the 2D array with a permutation.

What is a permutation?

A permutation is basically a 1D array which acts as a place holder for the position of the queens. Instead of marking position of the queen on a 2D array with a special variable and then taking the 2 array parameters that uniquely define a cell; the permutation itself stores the position of the queen.

How?

The position index of the array is considered as the column number of the 2D chess board and the value at that position is the row in which the queen is placed.

Example 1:

A permutation	[	4	2	0	3	1	]
		⇓	⇓	⇓	⇓	⇓	
Position	{	0	1	2	3	4	}

In the above example the first element of a 1D array of length 5 is 4. So that means the queen in the first column is placed in the fourth row that is in a 2D array the first queen position will be in row [4] and column [0] and so on.

If you are placing K queens on a chess board of K x K. Then you will need a permutation or 1D array of length K to represent a genotype.

Mapping:

Thus the genotype in example 1 will be represented as the following in mapping process.

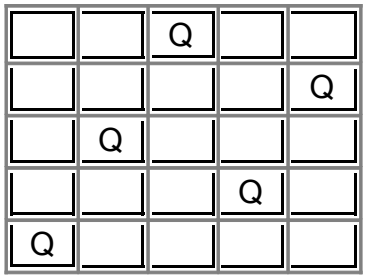


Figure 2: The genotype mapping to position of the queens

Phenotype:

The phenotype stays same as the genotype. That is the permutation. Since the fitness will be evaluated on the position of the queen; which we get itself get from the genotype. So the after processing the genotypes the traits on which the fitness will be evaluated will be the same permutation representation.
Thus it can be said that phenotype and genotype not only uses the same data structure (1D array in this case) but also the same representation. Thus in this problem both the genotype and the phenotype are same. Or both the terms can be applied to the same structure being used.

Environment:

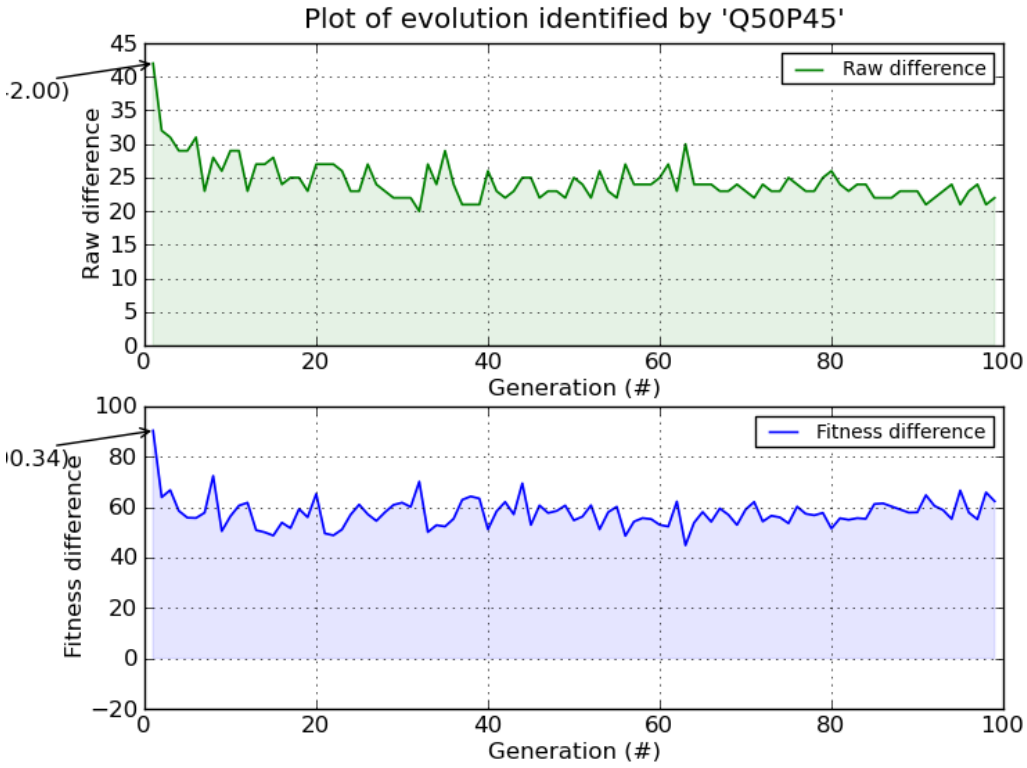
A word about the environment in which the genetic algorithm (GA) will run. Here environment is not the computer platform but one of the condition on which the fitness will be evaluated. Poison cells form a part of the problem environment. Before we run the GA we randomly populate a certain percentage of cells on the chess board with invalid poison marker. That is we take a 2D binary array (abstraction of chess board) and then mark the poison cells with 1 and the free cells as zero.
This binary array or the environment remains static throughout the process of evolution. Each generation of population compares against this environment to check the validity of the queen placement on a free cell as against a poison cell. For the purpose of the meaningful name in the program the environment is called as poison matrix.

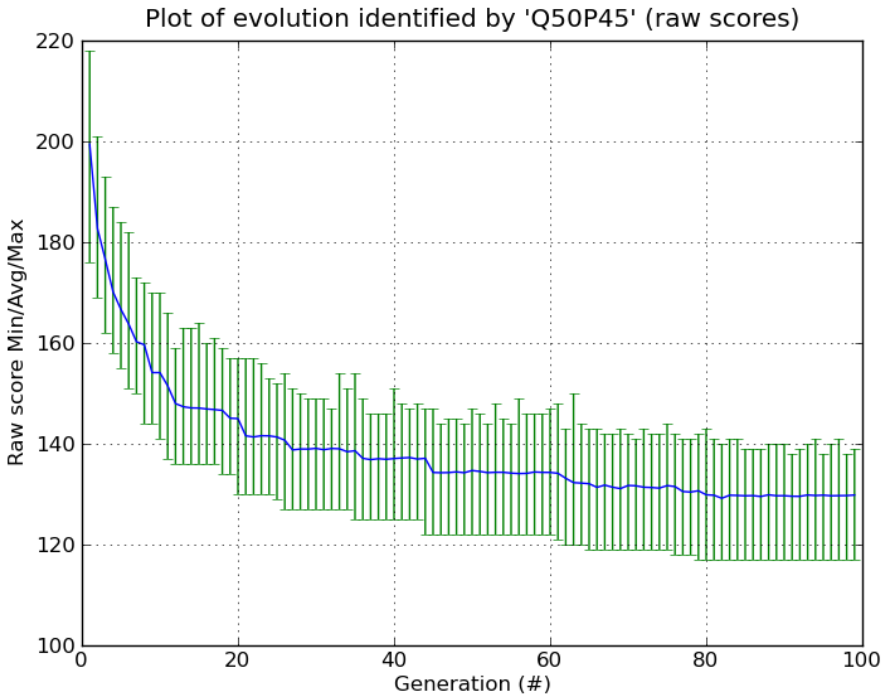
Individual:

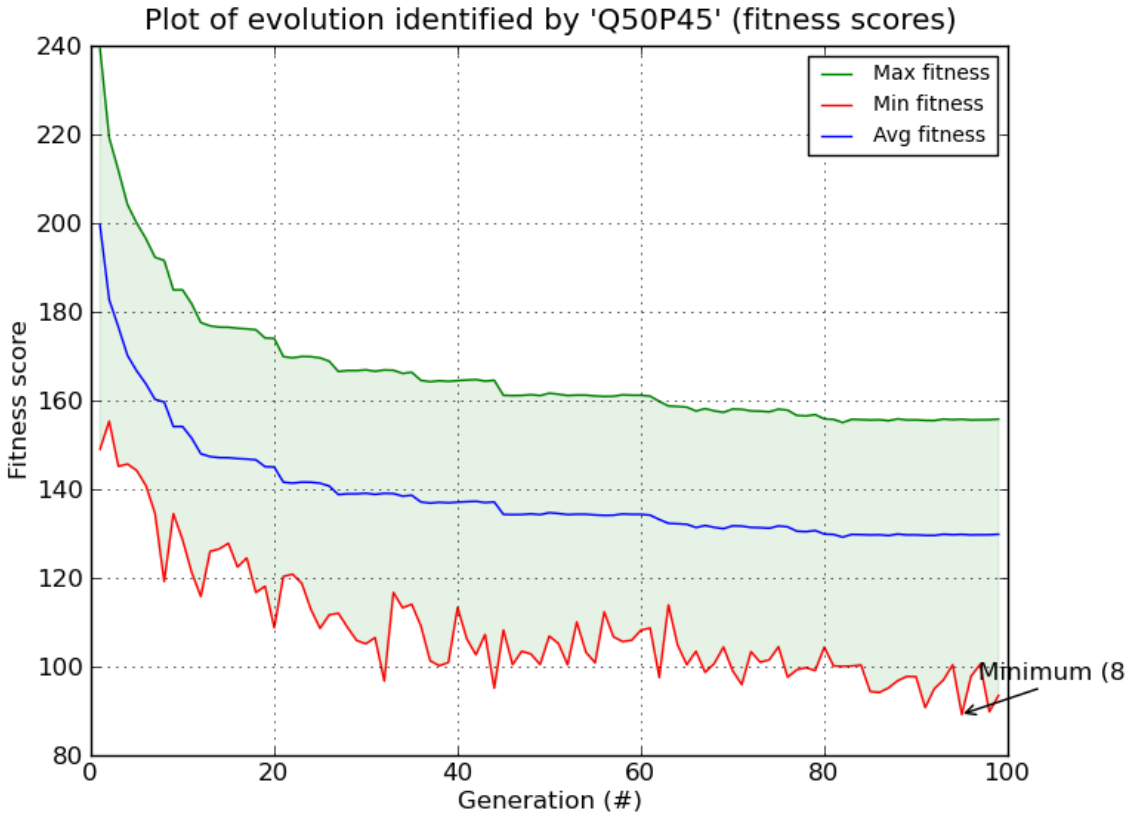
Since we have defined genotype and phenotype; so we can now define the individual. The individual or the solution also happens to be a permutation or rather the same genotype and phenotype. Since the permutation itself is the minimal requirement for us to map the solution output and will be evolved hence it can be treated as an individual.
Thus our permutations not only are genotypes but also represent individuals and the

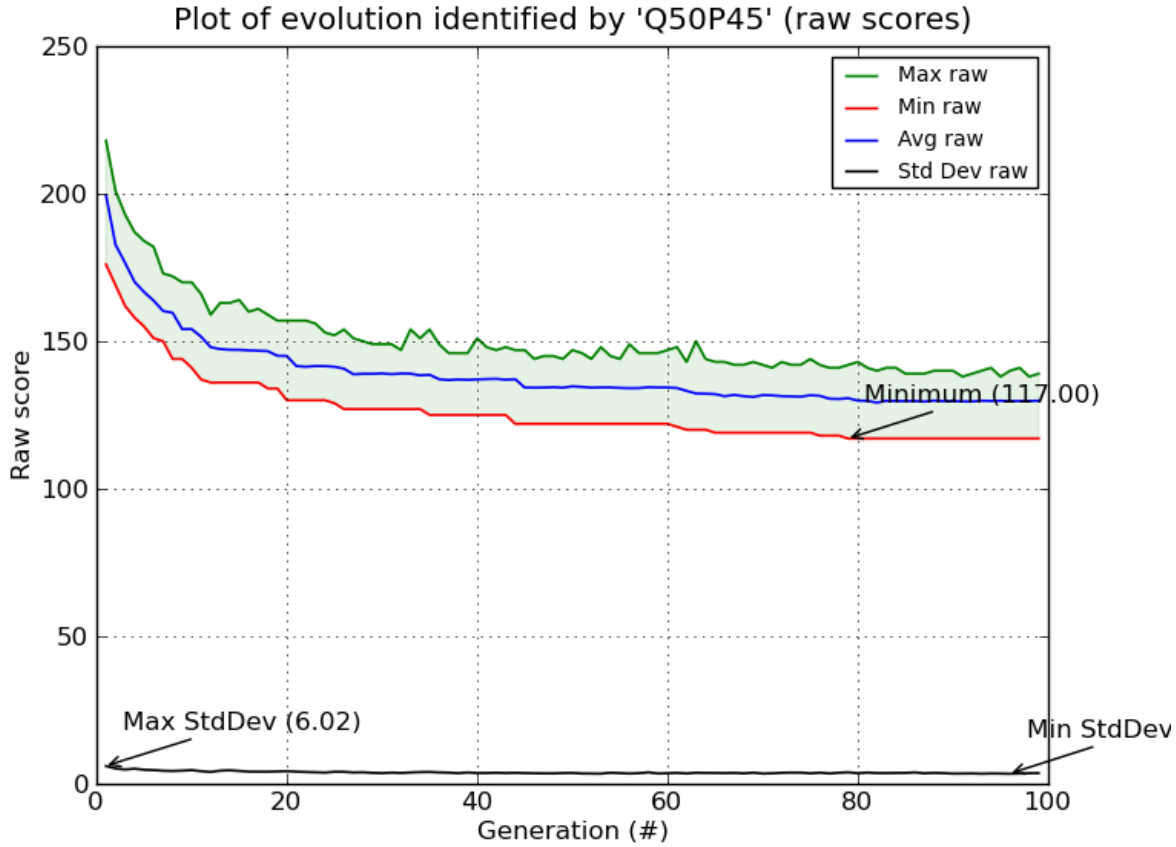
population.

Figure 3: Sample of graphs from run of 50 Queens and 45% poison.






Plot of evolution identified by 'Q50P45' (fitness scores)
Max fitness
Min fitness
Avg fitness
Fitness score
Generation (#)
Minimum (8
240
220
200
180
160
140
120
100
80
0
20
40
60
80
100


Plot of evolution identified by 'Q50P45' (raw scores)
Max raw
Min raw
Avg raw
Std Dev raw
Raw score
Generation (#)
Minimum (117.00)
Max StdDev (6.02)
Min StdDev
250
200
150
100
50
0
0
20
40
60
80
100

Fitness:

1. The fitness will be the best when no queen can capture each other and none of them are placed in a poison cell. Since each of the position in the genotype 1D array represents a unique column and no single position in the array can be occupied by more than one positional indicator number; which means that each queen is placed in separate column. Thus the fitness need not check for column capture.

2. This leaves only row and diagonal capture for fitness evaluation as per classical problem. For row capture it will be suffice to check if two elements in the 1D array have the same number. If yes, it means that they are in the same row and increment the fitness score by 1.

3. For diagonal capture we check for each chromosome or rather element in the genotype this condition absolute (genotype [i] – genotype [j]) = absolute (i – j).

4. If the above condition is true then it means that queen represented by genotype [i] can capture queen at position represented by genotype [j]. In that case we increment the fitness score by 1.

5. Similarly we check with the poison matrix to see if any of the queens are placed on a poison cell. A poison cell is one cell in poison matrix that is marked with the value 1. If it is on a poison cell then we increment the fitness score.

6. Thus each time we encounter a bad characteristic of a chromosome in a genotype we increase the fitness score. Thus a maximum fitness score means bad fitness and the reverse holds a minimum fitness score is the best fit.

7. So the fitness function is subjected to minimization function in the GA that is the best fitness is the minimum fitness score.

8. Fitness is minimisation so ideally the fitness score should be zero but due to the way the GA is coded; the minimisation value stands to be twice of the size of permutation.

9. In my fitness function the loops for checking diagonal and horizontal row capture are two nested for loops. Python has a for-each loop rather than classical for loop. Since the outer and inner for, loop through each element in each of the evaluation the element at the same place is compared once; which increments the score as per condition of matching.

10. For example: in diagonal check
    absolute (chromosome [i] – chromosome [j]) == absolute (i – j)
    will be true when i=j that is both sides become true and score increments. Similarly for horizontal check when same elements compared the score increments.

11. For each element the score is incremented twice once in the row check and then in diagonal check. Thus the minimum fitness score turns out to be twice the number of elements in the genotype or twice the number of queens.

Genetic Repair and Scaling:

Scaling need not be done for this problem. Because of the probability of getting a dominant individual is very unlikely at the start of the evolution. Secondly, I am using rank selection which means that even individuals with lower fitness do have a chance of crossover albeit only a limited.

Genetic repair arises in case where there are chances of the population being swarmed by invalid or bad genotypes. However the use of a permutation as a genotype prevents this situation. It depends on how much of fitness score qualifies as bad genotype. Since using permutation the level to which there will be horizontal or diagonal capture possible will be limited and hence will the existence of bad genomes.

The other problem being how do you define a bad genotype. A permutation may throw up a result where no queen captures the other in any way but each queen placed in a poison cell. In this case the genotype can be branded as a bad genotype but is also one of the probable solutions since a vast majority of free cells may force the queens to be placed in such a way that they can be captured. Thus to define a bad genotype we will have to give a bias towards either queens placed without attacking each other or queens not placed on poison cells. The best solution however lies somewhere in between.

In case we want to define a bad genotype we can handle it by tweaking the fitness function and giving bias towards certain cases in fitness function. For example if we want queens to be placed in poison cells but not attacking each other then we can increment the fitness score by 2 instead of 1 if queens capture each other but only increment fitness score by 1 if they are in poison. Remember the fitness function is minimization; so in case of queen capture the penalties are raised and poison cell placement penalties are low which can give a solution of queens not captured but some placed in poison cell. The reverse can also be done.

In any case the GA itself is a process of evolving the bad genotypes to good genotypes. So bad genotypes could be a part and since using a rank selector there will be cases where even the bad genotype will be able to participate albeit in a limited way. Also for my problem I have not biased the fitness to define a bad genotype and let it go with the flow to give me the best possible approximation.

Figure 4: Graph of 30 Queens and Poison cell percentage 30 for 120 generations

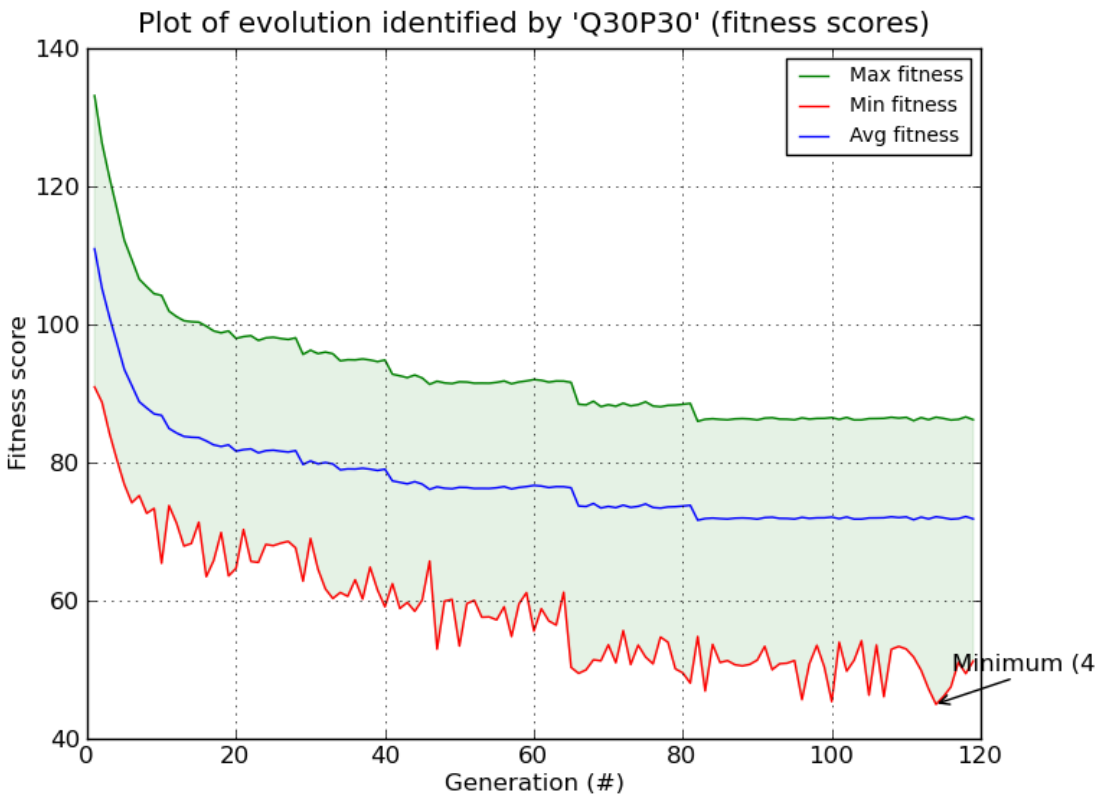
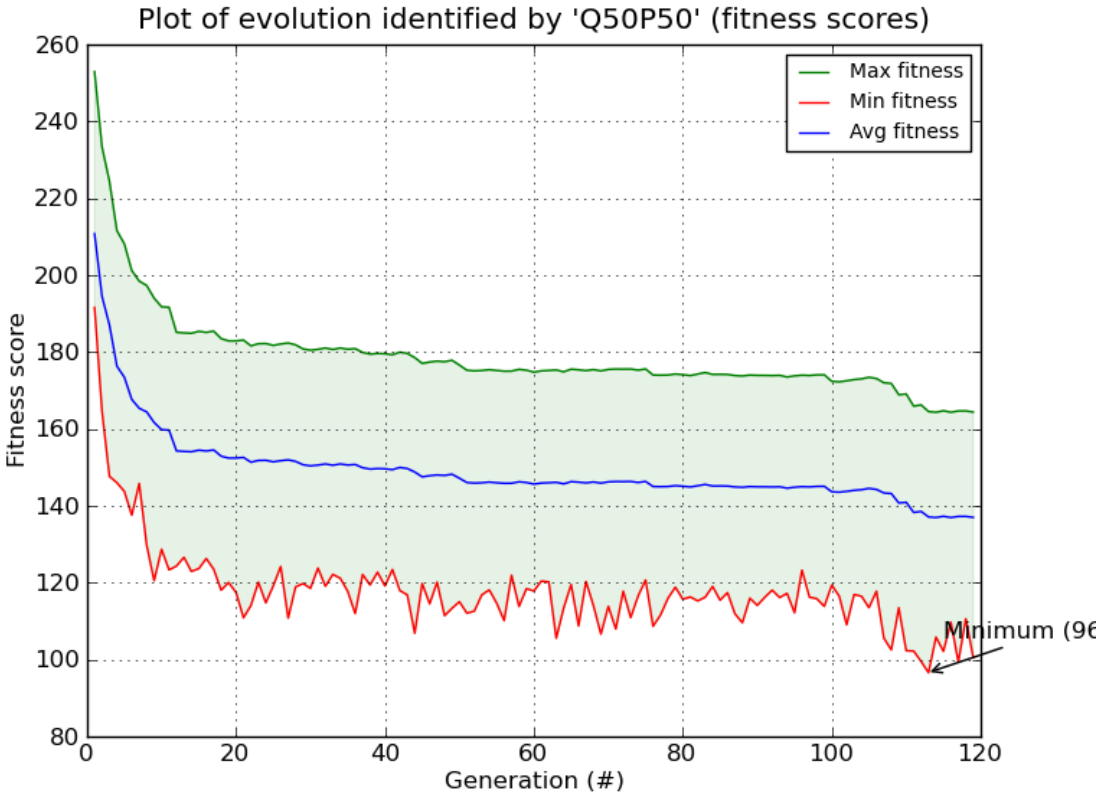


Figure 5: Graph of 50 Queens and 50% Poison cell for 120 generations



Crossover and Mutation:

The crossover is basically a method where genetic material between two participating individual genotypes is exchanged to produce new set of individuals.

Following are the crossover methods that are provided by Pyevolve to implement the genetic algorithm. My favourite pick of them is two point cross over. Although one point cross over works well but two point seems to better that and gives more flexibility.

Single Point Crossover:

In single point cross over for 1D array that is the genotype in question here. A random point is chosen for crossover. Then from that point the genetic material is divided into two parts. This is done for two individual participating for cross over. Each part of the split up genetic material pairs up with the split genetic part of the other individual.

For example parent 1= [3, 7, 2, 4, 8, 0, 5, 9, 1, 6] , parent 2 = [2, 6, 1, 7, 5, 0, 9, 4, 8, 3].
In both the parents the cross over point though randomly chosen is same for both the parents. In this case the crossover point is the one that divides the permutation in half.
So,
single point crossover (parent 1= [3, 7, 2, 4, 8, 0, 5, 9, 1, 6] + parent 2 = [2, 6, 1, 7, 5, 0, 9, 4, 8, 3]) =

child 1 [3, 7, 2, 4, 8, 0, 9, 4, 8, 3] + child 2 [2, 6, 1, 7, 5, 0, 5, 9, 1, 6]

Two point Crossover:

In two point cross over for 1D permutation instead of a single point two points are chosen for crossover. In short it is extension of single point crossover with two points for crossover. So instead of two parts the genetic material is split up into 3 parts. As with single point crossover here also the cross over points are kept fixed between two parents. After the split one part of one of the parent combines with two parts of the other parent.
For example:
parent 1= [3, 7, 2, 4, 8, 0, 5, 9, 1, 6] , parent 2 = [2, 6, 1, 7, 5, 0, 9, 4, 8, 3].

The portions highlighted in the genotype in different colours are different genetic materials divided. So after crossover the children will be.
Child 1 = [3, 7, 2, 7, 5, 0, 9, 9, 1, 6]
Child 2 = [2, 6, 1, 4, 8, 0, 5, 4, 8, 3]

Please note that as shown in the example above only the green middle portion need not be exchanged and portion may be exchanged.

Uniform Crossover:

In this crossover the chromosome are compared between the two parents and then are randomly swapped. For example parent 1 = [2, 6, 1, 7, 5, 0, 9, 4, 8, 3] and parent 2 = [3, 7, 2, 4, 8, 0, 5, 9, 1, 6]. So the child could be child 1 = [2, 7, 2, 7, 5, 0, 9, 1, 6]. This is however just

one of the possible child. Any combination can be applied.
Do note that there is inherent randomness in this crossover and properties of the parents may not persist in the child thus good parents can result in bad children through the uniform crossover.

Order Crossover:

In this cross over 2 arbitrary points are chosen for taking out a single part. That part is exchanged between the two parents to produce two new children. The elements that will be present in the swapped section are removed from the parent and then all are grouped together, so that they form contiguous elements without any gap in between. For example: parent 1= [3, 7, 2, 4, 8, 0, 5, 9, 1, 6] , parent 2 = [2, 6, 1, 7, 5, 0, 9, 4, 8, 3]. the n the children would be parent 1= [8, 1, 6, 7, 5, 0, 9, 3, 2, 4] , parent 2 = [7, 9, 3, 4, 8, 0, 5, 2, 6, 1].

Mutation:

Pyevolve provides only three mutation operators to work with - integer range, real range and list swap. Any other mutator that works on single dimensional array seems to give problem with the evolution algorithm.

The first two operators take a minimum and maximum value which is optional and by default it picks up the minimum and the maximum value of the 1D array or the permutation. The minimum and the maximum values if not provided are taken up randomly. Once the number of how many elements should be mutated in a genotype then those many elements are selected randomly and then those elements are randomly given a value from the permutation range.

The swap mutation operator just randomly takes a bunch of chromosome from genotype and then swaps the positions with the other chromosomes.

Conclusion and choices based on the checkpoint runs:

For my algorithm run I found that two point and single point crossovers worked the best. Especially the two point crossovers; though single point crossovers also gave results. But two point crossover had a better consistency in the solutions. Uniform and order crossover provided not so good results. The most lagging was uniform cross over as per my limited number of trials. It may be due to fact that though there are good solutions but there is an inherent more randomness in uniform crossover as compared to other crossover methods.

For mutation I had two choices integer and real range since other mutation methods were incompatible with my genotype and algorithm. Any one of the mutation operator could do and found no specific bias towards one. The swap mutation operator was found to be less popular.

Figure 6: Comparison graph raw score of different crossover rates.

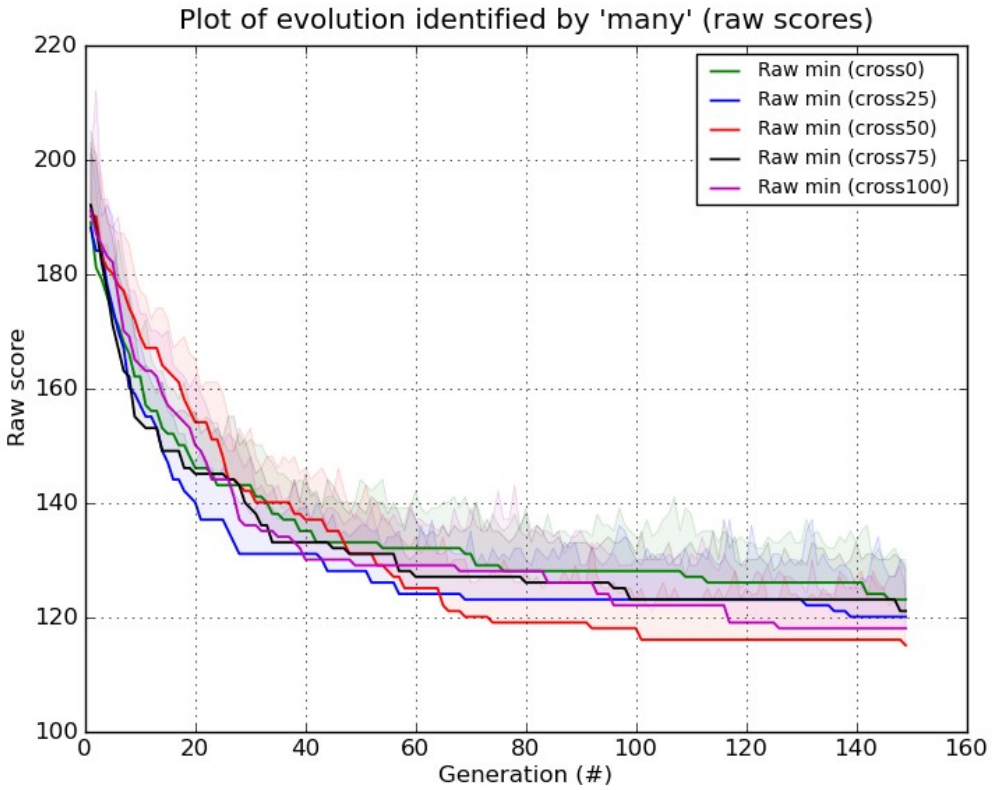
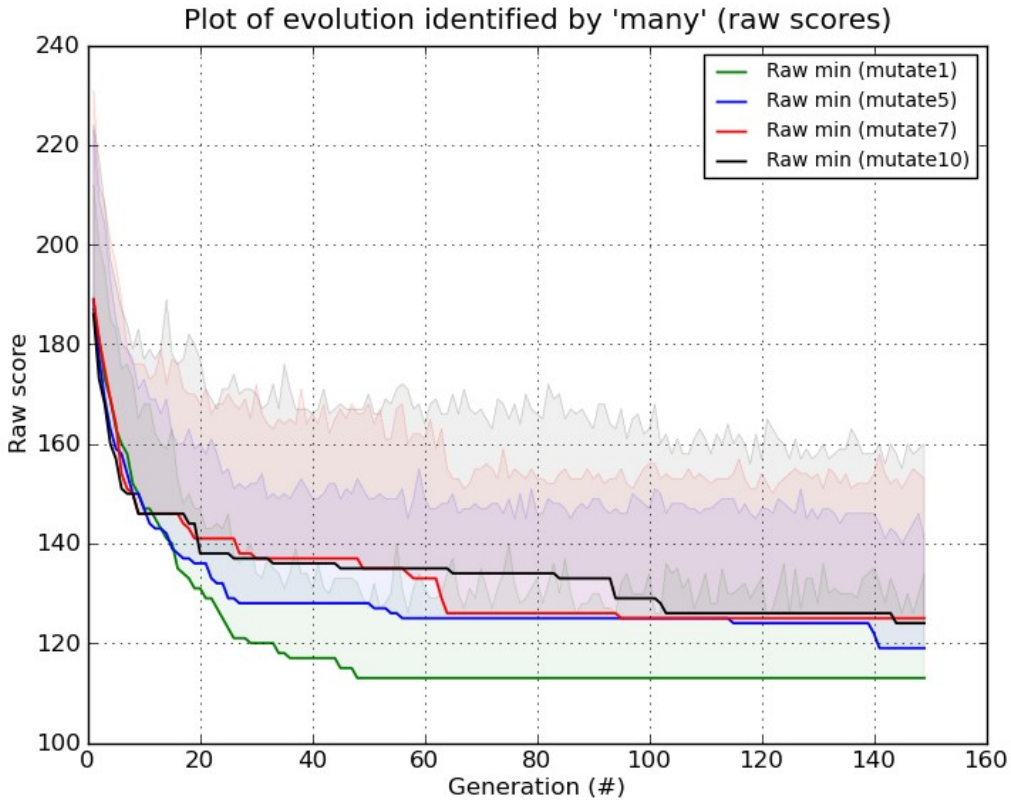


Figure 7: Comparison graph of raw fitness score of different mutation rates



Selection:

Pyevovle framework implements the algorithms as per David E Goldberg. So it has no mechanisms of overlapping generations. That is parents do not move to the next generation. So only experimentation can be done in selection scheme can be those in choosing the parents for cross over for next generation. There are four methods for selection – roulette wheel, tournament, uniform and rank selection.

Uniform selection

Individuals are selected randomly irrespective of fitness (uniformly) from the population to be the parents for crossover.

Rank Selection

All individuals are ranked as per fitness and then they are given a fitness function. Thus even the bad genomes get a chance to participate in crossover though in a limited way.

Roulette wheel:

Each time a individual is chosen from the population. Though the process may seem random but if the fittest individuals crowd the population, they will be chosen majority of the times.

Tournament selection:

Individuals are paired and as a tournament the fittest individual amongst the pair is chosen for the cross over.

Bad Genomes:

As already explained bad genomes will be taken care of by the fitness function. Also the problem lies in what you define as bad genome.

Conclusions:

1. For larger number of queens of order of 75 and highly poisoned chess board you need a larger number of generations to find the best solution. However it is not of much importance since at only certain generations does the fitness score drops a little. After a little drop in fitness around generation 100 it does drop a little bit around generation 140. But after generation 140 it stays the same till generation 200.

2. So, for larger number of queens and larger poison values. The longer the number of generations the better. However, at around for generation 100 or a bit above it around generation 120 would be suffice to find a approximate solution.

3. Although no bias was given but the algorithm concentrated on making sure that the queens do not attack each other; and then positioned the queens into the poison cells. A large number for poison cells almost 75 fell into the poison cells which was expected

but the interesting point to note is that most of the queens are not attacking each other.

4. A wild guesstimate for this is that around generation 140 and 100 the algorithm may have realised that poison cells are too many and hence cutting down on capture by queens may be able to bring the fitness score down.

5. It was clear that rank selection beats all other selection hands down; atleast for the checkpoint runs. The lowest fitness function of rank selection may be a due to probability of its poison cell placement being favourable. But that argument is turned down by the large gap in difference in the fitness function. The large number of poison cells and randomness of marking the cells means that there could be slight favourable variations in the fitness but not a large one.

6. Clearly for the entire population rank selection here comes out the best while tournament and uniform selection do not seem so favourable. Roulette selection performs somewhere in the middle. Rank selection converges faster than other algorithms in terms of generations.

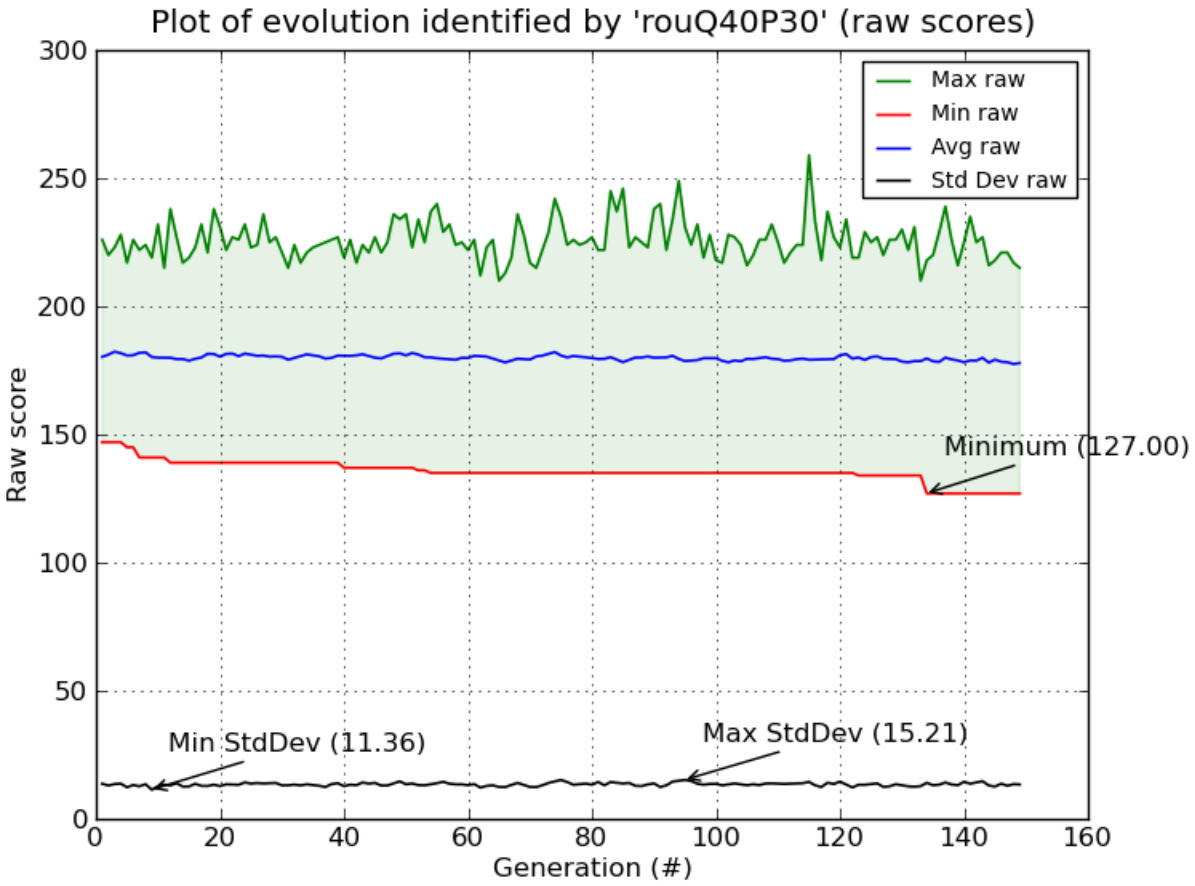


Figure 8: Graph of raw fitness score of Roulette wheel selection for 40 Queens and 30% poison

Fig 9: Graph of raw fitness score of Tournament Select for 40 Queens and 30% poison

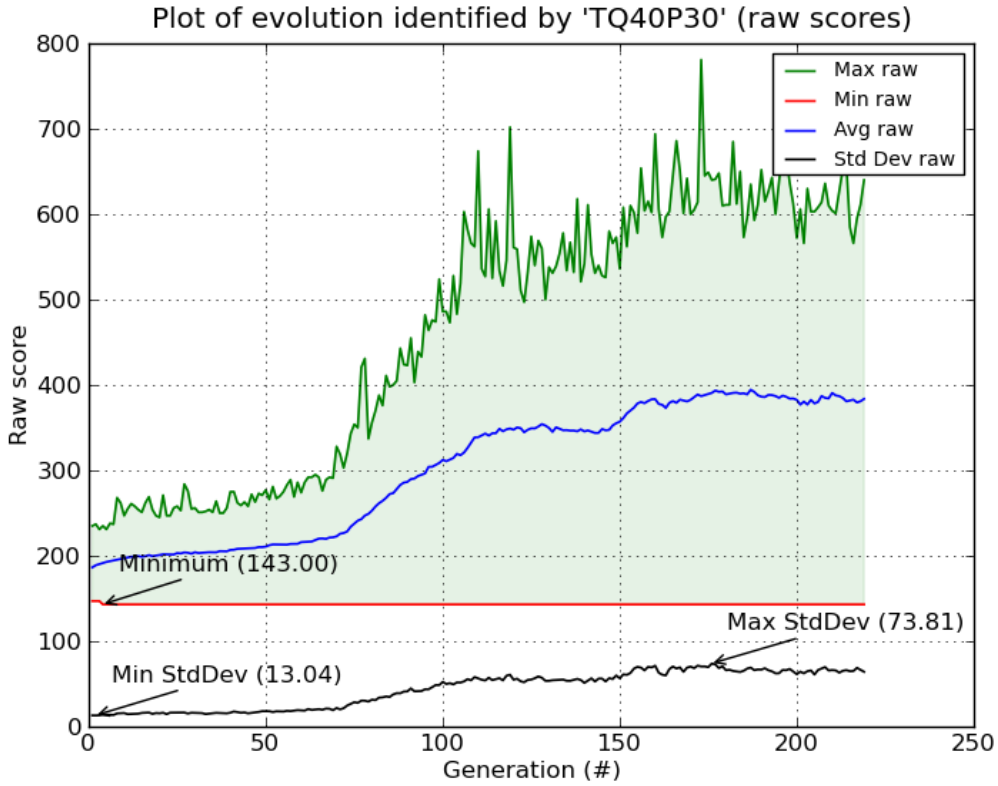


Fig 10: Graph of raw fitness score of Uniform Select for 40 Queens and 30% poison

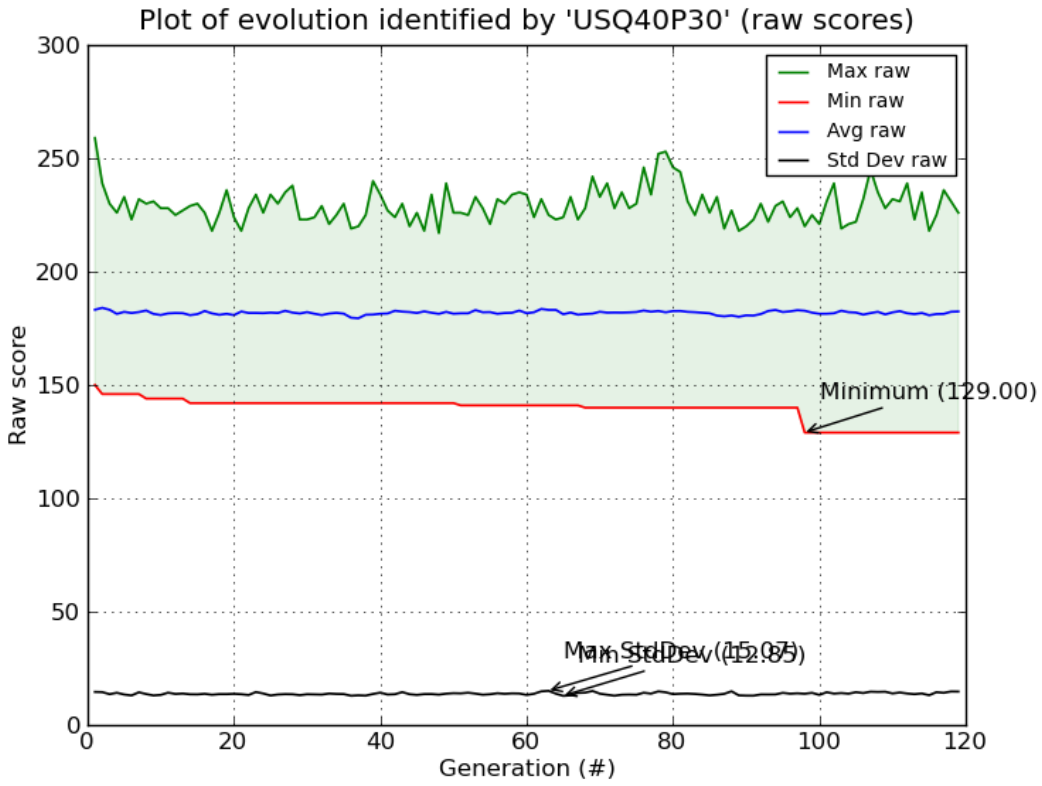
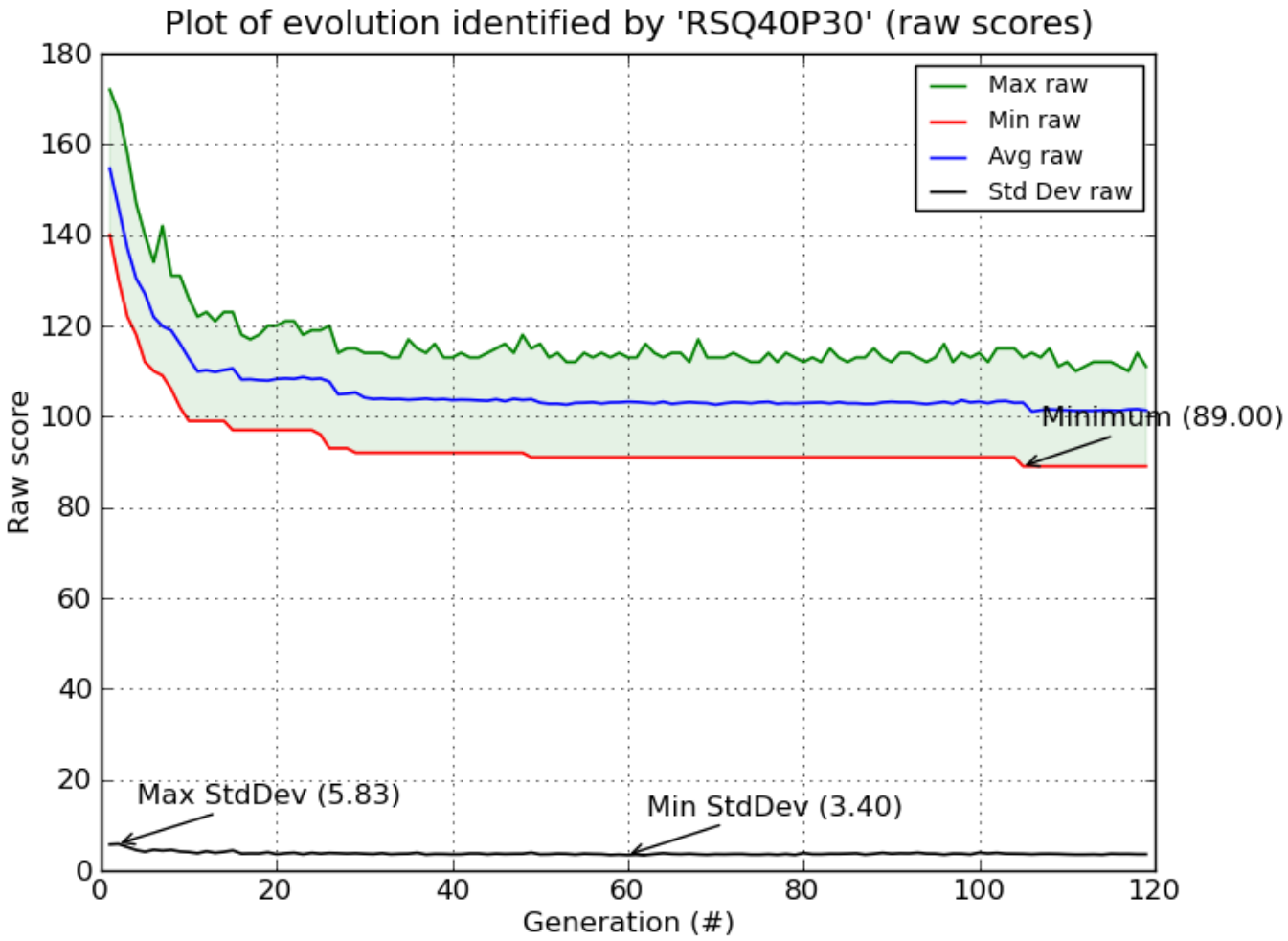


Fig 11: Graph of raw fitness score of Rank Select for 40 Queens and 30% poison



Halting Criteria:

Figure 12 and 13 given on the next page are subsets from the sets of graph that were run for different number of generations for different number of queens and different number of poison cells. The following conclusions have been drawn from those sets of graphs.

There was no particular halting criterion as such. For a lower number of queens (20) and lower levels of poison (20%) the terminating function provided by tool kit was around 100 generations where it seems to find a optimal solution.

Even for problems of higher poison rate (50%) and higher number of queens (greater than 50) around generation 100 you come to almost a optimal solution. However for the best solution the algorithm needs to run more generations.

Basically after generation 100 the fitness value seems to plateau and there are occasional dips after that but minimal in around generation 140. So for a higher poison value if you are keeping a medium population (number of queens x 5 to 7.5) you need to increase generations or wait more for the best approximate solution. Generation size up to 200 almost. But for most of the cases a generation size of 100 seems to work well.

Figure 12: Fitness Graph of 60 Queens and 60% poison cells for 150 generations

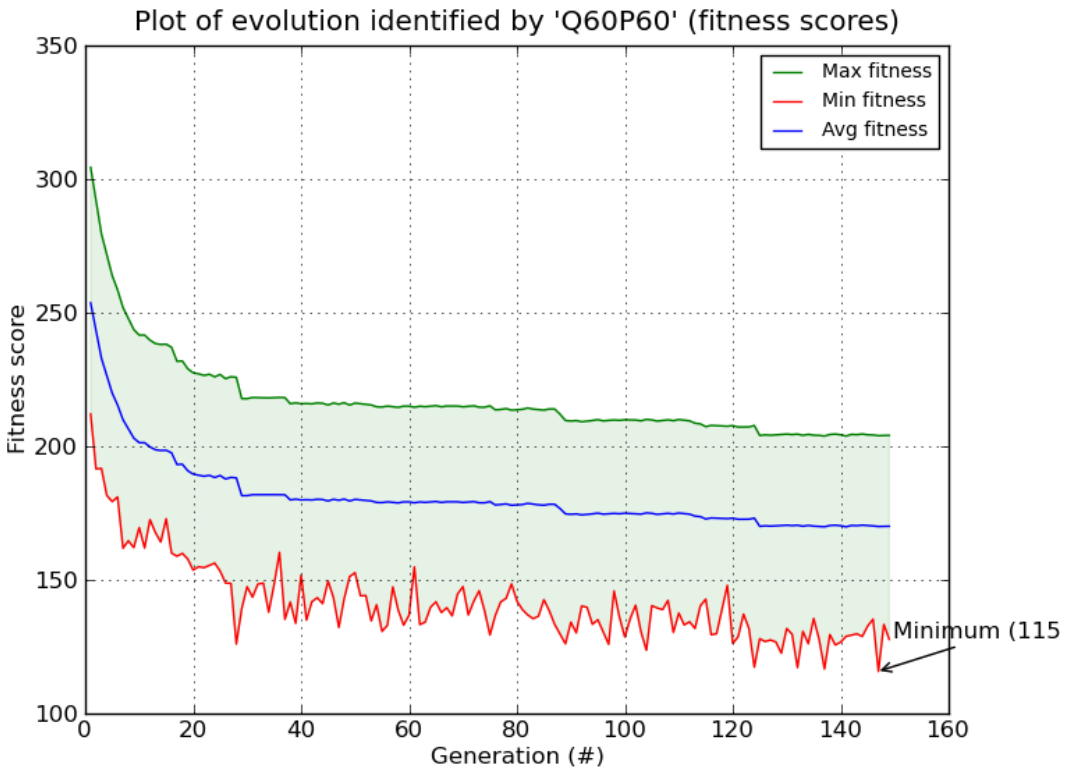
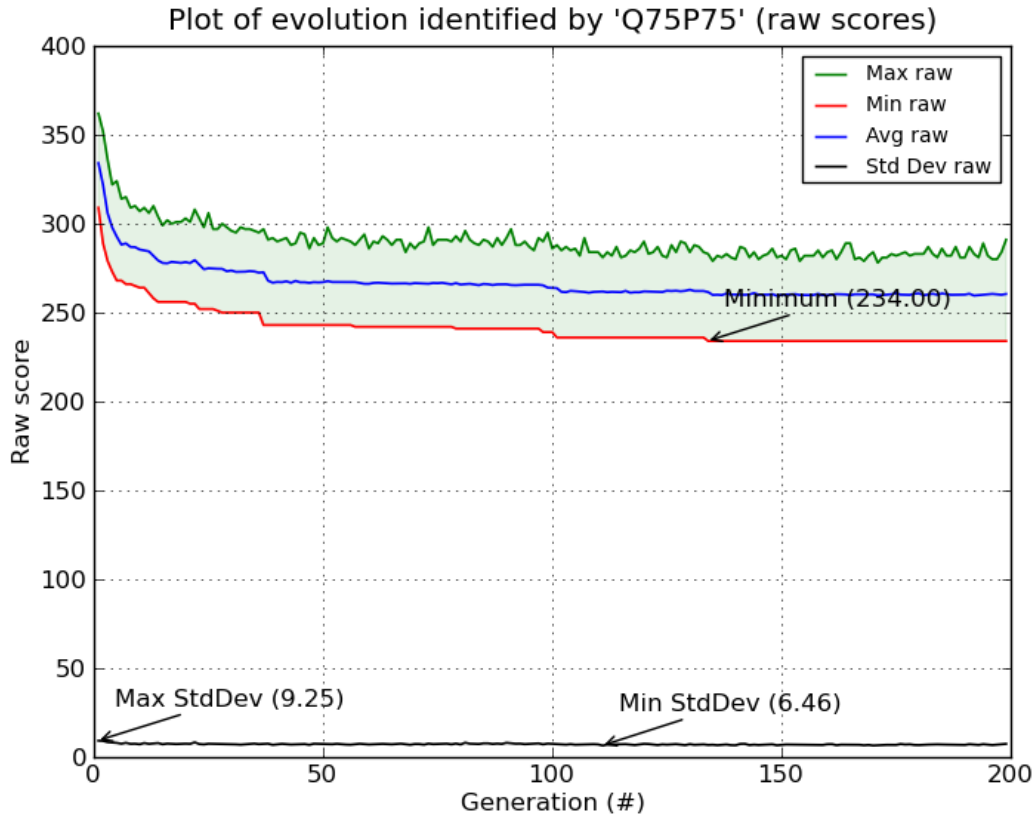


Figure 13: Fitness Graph of 75 Queens with 75% of cells poisoned, for 200 generations



Sample Run:

The following run was done for 100 Queens and 50% of cells poisoned and for 240 generations. Though it shows a runtime of around 8740 seconds it is a bit misleading since the computer was on hibernate mode for almost a hour during that run.

The placement of the queens on the board and the poison / free cell diagrams are attached in this folder in a .ods file (spreadsheet).

```
Please enter the number of the queens you want to find solution for: 100
Please enter the percentage of cells that you want poisoned: 50
Please enter the population size that you want for evolution: 800
Please enter the number of generations you want to evolve: 240
Please set the crossover percentage: 100
Please set the mutation percentage: 3
Set elitism to True or False: False
Enter the data base file name you want : Queen100
Please associate a id with this evolution run: Q100P50
Gen. 1 (0.42%): Max/Min/Avg Fitness(Raw) [503.96(440.00)/344.65(402.00)/419.97(419.96)]
Current generation: 10
- Statistics
                Minimum raw score               = 316.00
                Minimum fitness                 = 270.55
                Standard deviation of raw scores        = 4.60
                Maximum fitness                 = 396.29
                Maximum raw score               = 346.00
                Fitness average                 = 330.24
                Raw scores variance             = 21.20
                Average of raw scores           = 330.24

Current generation: 20
- Statistics
                Minimum raw score               = 290.00
                Minimum fitness                 = 240.08
                Standard deviation of raw scores        = 4.19
                Maximum fitness                 = 363.26
                Maximum raw score               = 315.00
                Fitness average                 = 302.71
                Raw scores variance             = 17.51
                Average of raw scores           = 302.71

Gen. 20 (8.33%): Max/Min/Avg Fitness(Raw) [363.26(315.00)/240.08(290.00)/302.71(302.71)]
Current generation: 30
- Statistics
                Minimum raw score               = 270.00
                Minimum fitness                 = 222.27
                Standard deviation of raw scores        = 4.05
                Maximum fitness                 = 338.18
```

Maximum raw score = 293.00
Fitness average = 281.82
Raw scores variance = 16.42
Average of raw scores = 281.82

Current generation: 40
- Statistics
Minimum raw score = 263.00
Minimum fitness = 214.13
Standard deviation of raw scores = 3.77
Maximum fitness = 330.11
Maximum raw score = 286.00
Fitness average = 275.09
Raw scores variance = 14.22
Average of raw scores = 275.09

Gen. 40 (16.67%): Max/Min/Avg Fitness(Raw)
[330.11(286.00)/214.13(263.00)/275.09(275.09)]
Current generation: 50
- Statistics
Minimum raw score = 256.00
Minimum fitness = 211.67
Standard deviation of raw scores = 3.76
Maximum fitness = 321.99
Maximum raw score = 280.00
Fitness average = 268.33
Raw scores variance = 14.12
Average of raw scores = 268.32

Current generation: 60
- Statistics
Minimum raw score = 250.00
Minimum fitness = 207.04
Standard deviation of raw scores = 3.70
Maximum fitness = 314.78
Maximum raw score = 274.00
Fitness average = 262.31
Raw scores variance = 13.66
Average of raw scores = 262.31

Gen. 60 (25.00%): Max/Min/Avg Fitness(Raw)
[314.78(274.00)/207.04(250.00)/262.31(262.31)]
Current generation: 70
- Statistics
Minimum raw score = 249.00
Minimum fitness = 198.87
Standard deviation of raw scores = 3.61
Maximum fitness = 313.84
Maximum raw score = 272.00

Fitness average = 261.54
Raw scores variance = 13.01
Average of raw scores = 261.54

Current generation: 80
- Statistics
Minimum raw score = 247.00
Minimum fitness = 198.00
Standard deviation of raw scores = 3.61
Maximum fitness = 311.37
Maximum raw score = 270.00
Fitness average = 259.47
Raw scores variance = 13.01
Average of raw scores = 259.47

Gen. 80 (33.33%): Max/Min/Avg Fitness(Raw)
[311.37(270.00)/198.00(247.00)/259.47(259.47)]
Current generation: 90
- Statistics
Minimum raw score = 247.00
Minimum fitness = 209.11
Standard deviation of raw scores = 3.74
Maximum fitness = 311.81
Maximum raw score = 273.00
Fitness average = 259.84
Raw scores variance = 13.98
Average of raw scores = 259.84

Current generation: 100
- Statistics
Minimum raw score = 245.00
Minimum fitness = 194.07
Standard deviation of raw scores = 3.47
Maximum fitness = 309.25
Maximum raw score = 268.00
Fitness average = 257.71
Raw scores variance = 12.03
Average of raw scores = 257.71

Gen. 100 (41.67%): Max/Min/Avg Fitness(Raw)
[309.25(268.00)/194.07(245.00)/257.71(257.71)]
Current generation: 110
- Statistics
Minimum raw score = 245.00
Minimum fitness = 204.74
Standard deviation of raw scores = 3.69
Maximum fitness = 309.20
Maximum raw score = 270.00
Fitness average = 257.67

Raw scores variance = 13.63
Average of raw scores = 257.67

Current generation: 120
- Statistics
Minimum raw score = 245.00
Minimum fitness = 192.06
Standard deviation of raw scores = 3.65
Maximum fitness = 309.48
Maximum raw score = 268.00
Fitness average = 257.90
Raw scores variance = 13.33
Average of raw scores = 257.90

Gen. 120 (50.00%): Max/Min/Avg Fitness(Raw)
[309.48(268.00)/192.06(245.00)/257.90(257.90)]
Current generation: 130
- Statistics
Minimum raw score = 243.00
Minimum fitness = 187.76
Standard deviation of raw scores = 3.44
Maximum fitness = 306.65
Maximum raw score = 265.00
Fitness average = 255.54
Raw scores variance = 11.85
Average of raw scores = 255.54

Current generation: 140
- Statistics
Minimum raw score = 243.00
Minimum fitness = 184.31
Standard deviation of raw scores = 3.41
Maximum fitness = 306.99
Maximum raw score = 265.00
Fitness average = 255.83
Raw scores variance = 11.62
Average of raw scores = 255.82

Gen. 140 (58.33%): Max/Min/Avg Fitness(Raw)
[306.99(265.00)/184.31(243.00)/255.83(255.82)]
Current generation: 150
- Statistics
Minimum raw score = 242.00
Minimum fitness = 199.78
Standard deviation of raw scores = 3.62
Maximum fitness = 306.00
Maximum raw score = 267.00
Fitness average = 255.00
Raw scores variance = 13.11

Average of raw scores = 255.00

Current generation: 160
- Statistics
Minimum raw score = 242.00
Minimum fitness = 196.26
Standard deviation of raw scores = 3.52
Maximum fitness = 305.80
Maximum raw score = 266.00
Fitness average = 254.83
Raw scores variance = 12.38
Average of raw scores = 254.83

Gen. 160 (66.67%): Max/Min/Avg Fitness(Raw)
[305.80(266.00)/196.26(242.00)/254.83(254.83)]
Current generation: 170
- Statistics
Minimum raw score = 242.00
Minimum fitness = 198.63
Standard deviation of raw scores = 3.55
Maximum fitness = 306.17
Maximum raw score = 267.00
Fitness average = 255.14
Raw scores variance = 12.58
Average of raw scores = 255.14

Current generation: 180
- Statistics
Minimum raw score = 239.00
Minimum fitness = 178.05
Standard deviation of raw scores = 3.43
Maximum fitness = 302.50
Maximum raw score = 261.00
Fitness average = 252.09
Raw scores variance = 11.74
Average of raw scores = 252.09

Gen. 180 (75.00%): Max/Min/Avg Fitness(Raw)
[302.50(261.00)/178.05(239.00)/252.09(252.09)]
Current generation: 190
- Statistics
Minimum raw score = 237.00
Minimum fitness = 177.16
Standard deviation of raw scores = 3.56
Maximum fitness = 300.06
Maximum raw score = 259.00
Fitness average = 250.05
Raw scores variance = 12.69
Average of raw scores = 250.05

Current generation: 200
- Statistics

Minimum raw score	= 236.00
Minimum fitness	= 187.63
Standard deviation of raw scores	= 3.37
Maximum fitness	= 299.12
Maximum raw score	= 260.00
Fitness average	= 249.27
Raw scores variance	= 11.34
Average of raw scores	= 249.27

Gen. 200 (83.33%): Max/Min/Avg Fitness(Raw)
[299.12(260.00)/187.63(236.00)/249.27(249.27)]
Current generation: 210
- Statistics

Minimum raw score	= 236.00
Minimum fitness	= 193.81
Standard deviation of raw scores	= 3.37
Maximum fitness	= 298.99
Maximum raw score	= 261.00
Fitness average	= 249.16
Raw scores variance	= 11.38
Average of raw scores	= 249.16

Current generation: 220
- Statistics

Minimum raw score	= 235.00
Minimum fitness	= 187.56
Standard deviation of raw scores	= 3.37
Maximum fitness	= 297.84
Maximum raw score	= 259.00
Fitness average	= 248.20
Raw scores variance	= 11.37
Average of raw scores	= 248.20

Gen. 220 (91.67%): Max/Min/Avg Fitness(Raw)
[297.84(259.00)/187.56(235.00)/248.20(248.20)]
Current generation: 230
- Statistics

Minimum raw score	= 235.00
Minimum fitness	= 194.00
Standard deviation of raw scores	= 3.49
Maximum fitness	= 297.64
Maximum raw score	= 260.00
Fitness average	= 248.03
Raw scores variance	= 12.20
Average of raw scores	= 248.03

Gen. 240 (100.00%): Max/Min/Avg Fitness(Raw)
[298.07(258.00)/179.17(235.00)/248.39(248.39)]
Total time elapsed: 8740.120 seconds.
- GenomeBase
Score: 235.000000
Fitness: 179.171642

Slot [Evaluator] (Count: 1)
Name: eval_func
Slot [Initializator] (Count: 1)
Name: G1DListInitializatorInteger
Doc: Integer initialization function of G1DList

This initializator accepts the *rangemin* and *rangemax* genome parameters.

Slot [Mutator] (Count: 1)
Name: G1DListMutatorIntegerRange
Doc: Simple integer range mutator for G1DList

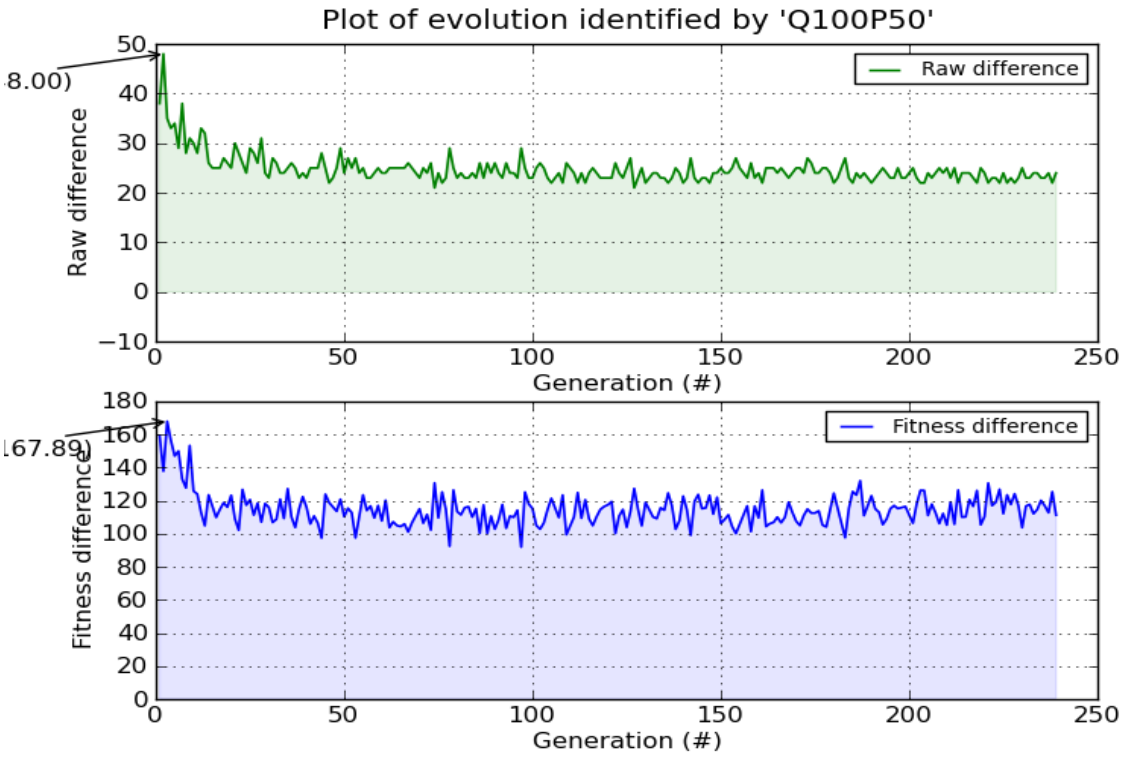
Accepts the *rangemin* and *rangemax* genome parameters, both optional.

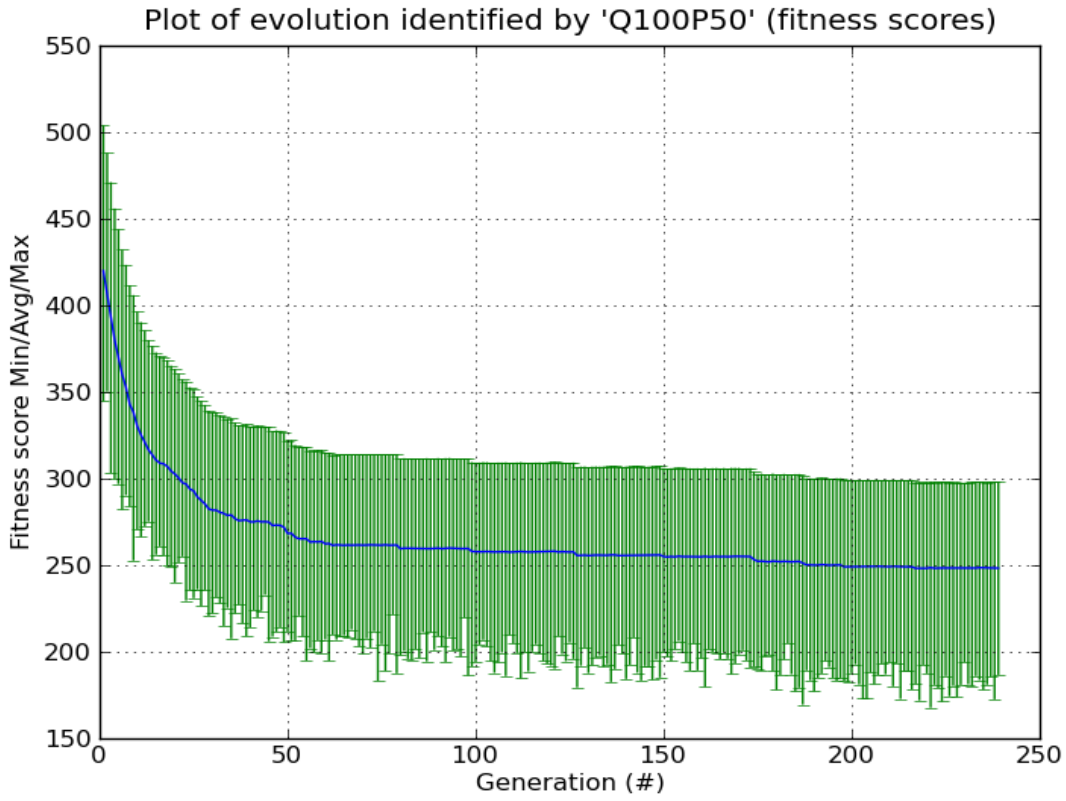
Slot [Crossover] (Count: 1)
Name: G1DListCrossoverTwoPoint
Doc: The G1DList crossover, Two Point

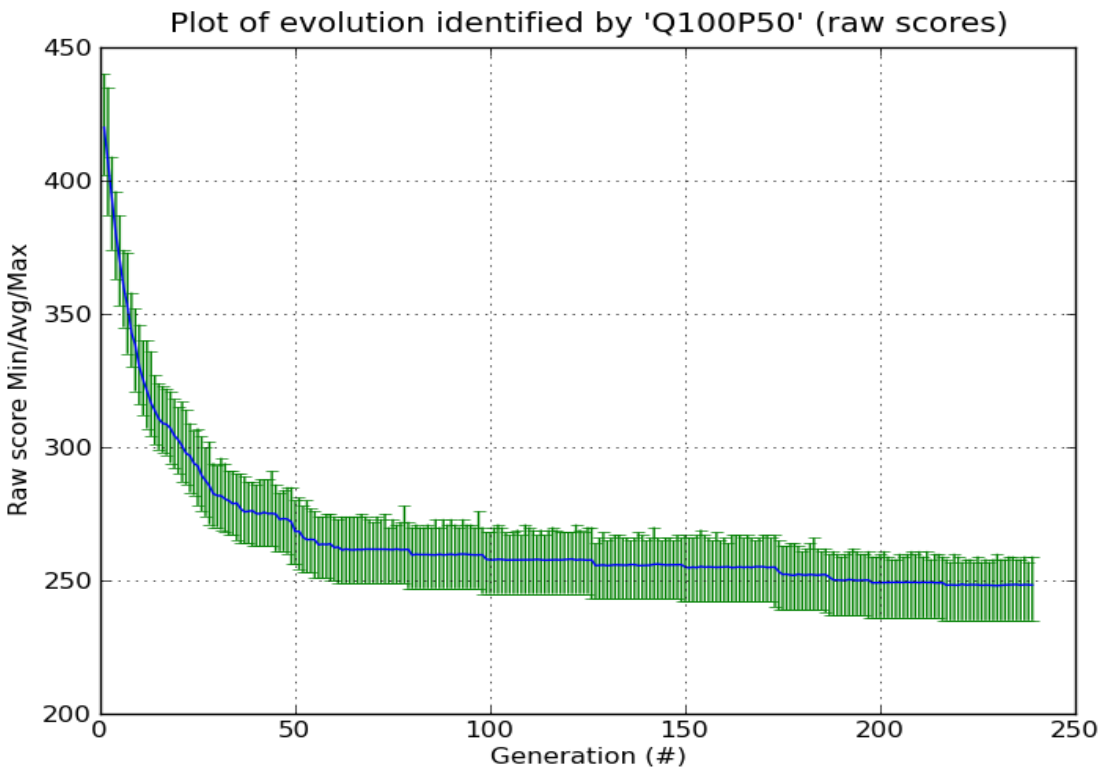
.. warning:: You can't use this crossover method for lists with just one element.

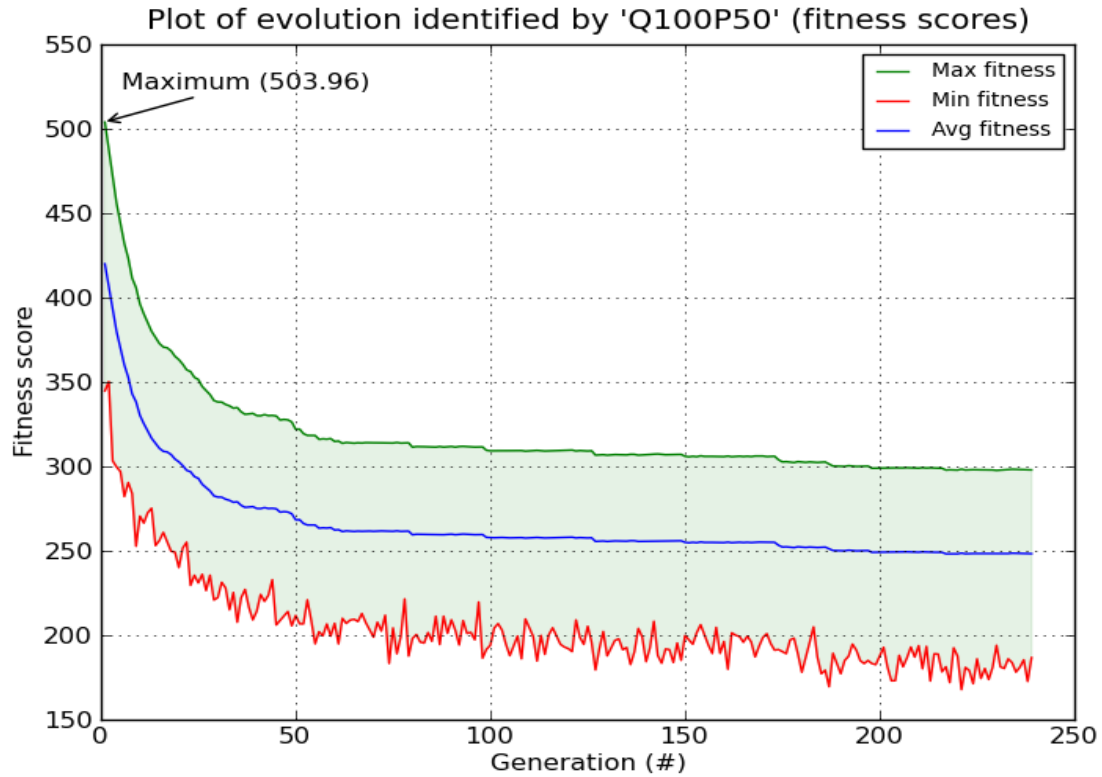
- G1DList
List size: 100
List: [29, 59, 36, 62, 86, 65, 27, 30, 12, 25, 76, 66, 95, 23, 57, 48, 67, 24, 40, 72, 90, 47, 31, 62, 7, 5, 95, 79, 33, 14, 26, 98, 60, 64, 74, 32, 13, 97, 87, 56, 88, 78, 80, 19, 10, 14, 11, 8, 61, 94, 0, 81, 41, 9, 45, 58, 46, 82, 29, 16, 71, 35, 43, 51, 34, 92, 68, 91, 69, 4, 18, 77, 17, 93, 0, 3, 70, 50, 2, 37, 75, 20, 44, 55, 63, 38, 85, 99, 72, 53, 83, 89, 84, 52, 21, 49, 28, 15, 42, 4]

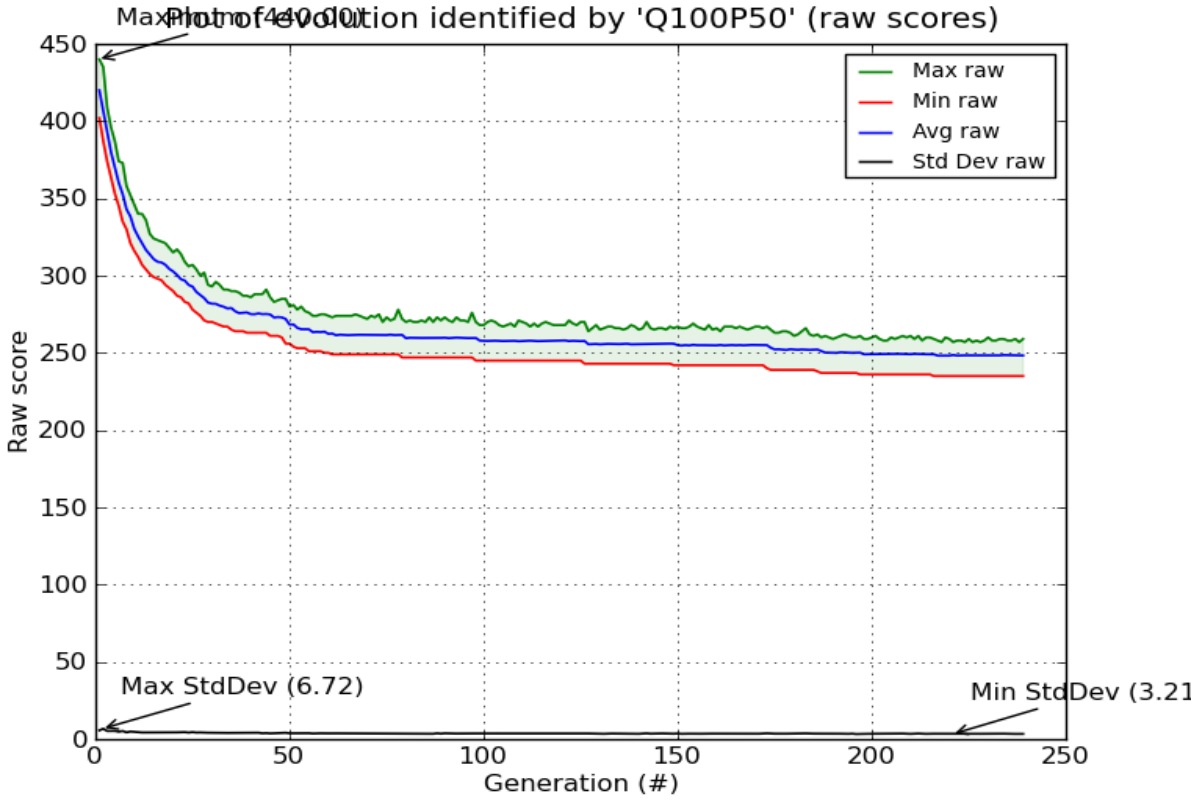
Figure 14: All the graphs related to the sample run

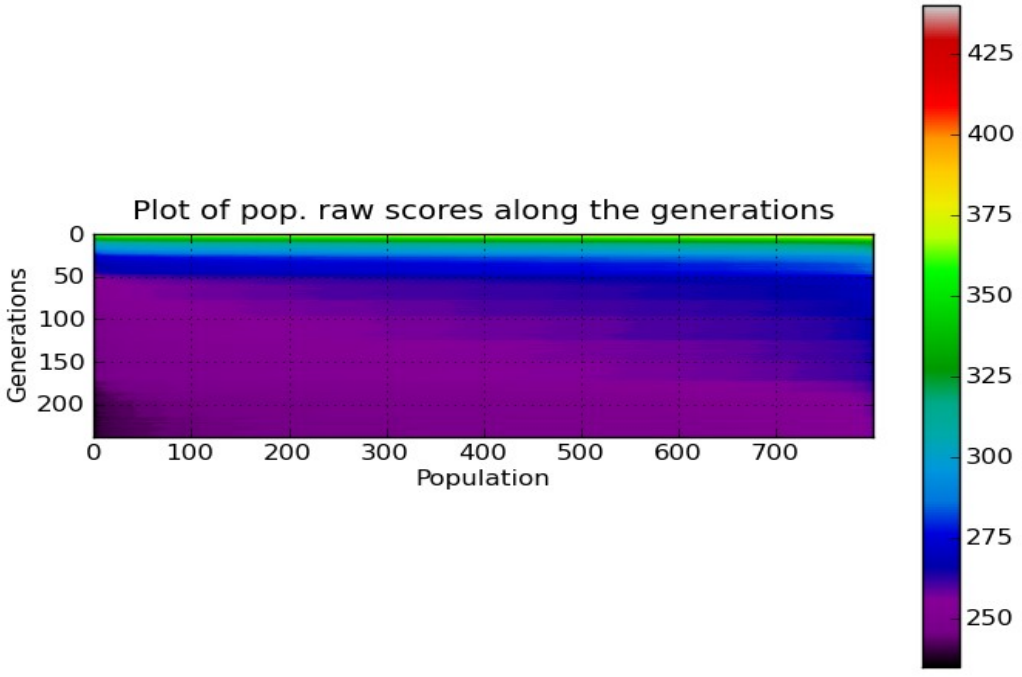





Plot of evolution identified by 'Q100P50' (raw scores)
Raw score Min/Avg/Max
Generation (#)
450
400
350
300
250
200
0
50
100
150
200
250


Plot of evolution identified by 'Q100P50' (fitness scores)
Maximum (503.96)
Max fitness
Min fitness
Avg fitness
Fitness score
Generation (#)
550
500
450
400
350
300
250
200
150
0
50
100
150
200
250


Plot of evolution identified by 'Q100P50' (raw scores)
Max raw
Min raw
Avg raw
Std Dev raw
Raw score
Generation (#)
Max StdDev (6.72)
Min StdDev (3.21
0
50
100
150
200
250
300
350
400
450


Plot of pop. raw scores along the generations
Generations
Population
0
50
100
150
200
0
100
200
300
400
500
600
700
425
400
375
350
325
300
275
250

Implementation:

The implementation details or the final code has been attached in the zip file folder of this report as GAFinalcode_nqueens. The result from the run has been included in the folder of this file in a spreadsheet N_queens.ods which has the position of the poison cells and the placement of the queens on two separate work sheets. A database was also generated during the run Queen100.db that has got all the details of individuals of all the runs for each generation.

The code has been implemented in python 2.5, using the tool kit pyevolve 0.5. Matplotlib and numpy was also required for plotting of the graphs. The graphs were plotted using the script file pyevolve_graph.py. More information on how to plot the graphs can be found at this location. http://pyevolve.sourceforge.net/graphs.html . The result spreadsheet requires another tool in Python called pyExcelerator which can be downloaded from here http://sourceforge.net/projects/pyexcelerator/ .
All the times the GA was run on my note book which is Core 2 Duo and runs Windows Vista 64 bit edtion.

Conclusions:

Crossovers and Mutation:

1. Found that two point and single point crossovers worked the best. Especially the two point crossovers; though single point crossovers also gave results. But two point crossover had a better consistency in the solutions.

2. Uniform and order crossover provided not so good results. The most lagging was uniform cross over as per my limited number of trials. It may be due to fact that though there are good solutions but there is an inherent more randomness in uniform crossover as compared to other crossover methods.

3. For mutation I had two choices integer and real range since other mutation methods were incompatible with my genotype and algorithm. Any one of the mutation operator could do and found no specific bias towards one. The swap mutation operator was found to be efficient.

Interestingly the best fit was crossover of 50% that seemed to converge fastest but that was the only exception and discarded as it did not fit the trend of the other crossover rates.

Population and Generations:

1. For larger number of queens of order of 75 and highly poisoned chess board you need a larger number of generations to find the best solution. However it is not of much importance since at only certain generations does the fitness score drops a little. After a little drop in fitness around generation 100 it does drop a little bit around generation 140. But after generation 140 it stays the same till generation 200.

2. So, for larger number of queens and larger poison values. The longer the number of generations the better. However, at around for generation 100 or a bit above it around generation 120 would be suffice to find a approximate solution. Also in the sample run for 100 queens and 50% poison it can be seen that at generation 100 it had already gained the substantive amount of fitness and after generation 220 the fitness is almost flat.

3. Although no bias was given but the algorithm concentrated on making sure that the queens do not attack each other; and then positioned the queens into the poison cells. For a large number for poison cells (75%) lot of queens were placed into the poison cells which was expected but the interesting point to note was that most of the queens were not attacking each other.

4. A wild guesstimate explanation for the above could be; that around generation 140 and 100 the algorithm may have realised that poison cells are too many and hence it started to cut down on capture by queens that may be able to bring the fitness score down.

Fitness and Selection

1. It was clear that rank selection beats all other selection hands down; at least for the checkpoint runs figure 8 to figure 12. The lowest fitness function of rank selection may be a due to probability of its poison cell placement being favourable. But that argument is turned down by the large gap in difference in the fitness function. The large number of poison cells and randomness of marking the cells means that there could be slight favourable variations in the fitness but not a large one.

2. Clearly for the entire population rank selection here comes out the best while tournament and uniform selection do not seem so favourable. Roulette selection performs somewhere in the middle. Rank selection converges faster than other algorithms in terms of generations.

Terminating condition:

1. There was no particular halting criterion as such. For a lower number of queens (20) and lower levels of poison (20%) the terminating function provided by tool kit was around 100 generations where it seems to find a optimal solution.

2. Even for problems of higher poison rate (50%) and higher number of queens (greater than 50) around generation 100 you come to almost a optimal solution. However for the best solution the algorithm needs to run more generations.

3. Basically after generation 100 the fitness value seems to plateau and there are occasional dips after that but minimal in around generation 140. So for a higher poison value if you are keeping a medium population (number of queens x 5 to 7.5) you need to increase generations or wait more for the best approximate solution. Generation size up to 200 almost. But for most of the cases a generation size of 100 seems to work well.

Last but not least a deterministic algorithm implemented in Java for n queen executed in almost same amount of time in fact less for a queen of larger size. The deterministic algorithm was solving the classical n queen; whereas the GA was solving the same number of queens but with 20% poison.

Future Work

1. First and foremost if possible in future would like to code from scratch. The tool kit though quite good had limitations like not allowing over lapping populations.

2. Would like to experiment with more figures and higher number of queens at least in the range of thousands. Most of my runs were queens in two digit figures and hence it does not seem to be a satisfactory number for making any conclusions.

3. Would like to experiment more on the population-generation relationship. It seems that the number of population and generations are inversely involved in a good solution. Although have presented a rough approximation in the conclusions but would like to fine tune that more. What combination of population and generation for a given poison percentage and queens is the best?

4. Would preferably like to test the same algorithm with different language implementations as a extension of point 1. Currently it takes a long time for number of queens in 3 digits, will like bring that computation number down.

5. Lastly didn't take the risk of having all the cells poisoned that is poison =100%, but guesstimating from runs of poison 90% can say that after 100 generations it will arrange queens in a manner that none can capture each other but will stay in poison cell. But do not have the evidence to show the same. The reader may attempt this with the code already provided.

Acknowledgments:

- Would like to thank Christian S Perone creator of Pyevovle toolkit for language python. All the wonderful graphs, databases, and run results have been possible due to this. Also for tolerating my stupid question emails; when something did not work on the frame work or the how to's.

- To Prof Peter Anderson for suggesting the problem and Prof Joe Geigel for suggesting changes in the genotype to get better results.

- And lastly my room mates who relieved me of my cooking duties, during completion of the report.