CSC3423 Bio-Computing

Coursework 1: Biologically-inspired computing for optimisation

Callum Simpson (UG) – b6030326

In this report I will be discussing how I modified a GA and PSO algorithm to find an optimal value for the Rastringin function.

Description

GA

Genetic algorithm is an algorithm inspired by natural selection that refines a pool of solutions to gets its most optimal fitness. It works by performing an initial step then 4 cycle phases to modify a population.

initial population.

In this step a random population is created. A population are possible solutions to a problem, normally represented as a vector of elements called a chromosome. A chromosome is made up by individuals called genes.

1 Evaluation (Fitness function)

Each particle in the population is evaluated and given a fitness scores which is used to determine the probability that the individual will be selected for reproduction.

2 Selection.

This represents natural selection where the best (most fit) individual will survive and be passed onto the next generation and the weak individual will be removed. There are many different types of selection methods. For example, roulette wheel and torment size

3 Crossover.

In this phase 2 parent induvial are chosen from the population and based on a certain probability will create two offspring. There are many different types of crossover.

> 1 – point crossover -- Select a point in with the chromosome and swap the genes on either side of that crossover point with the other parent to create two children.
>
> 2-point crossover – Similar to 1 – point but take 2 points instead.
>
> Uniform crossover – decide gene by gene which parent we get it from.

4 is Mutation

In this stage we make changes to a gene in a chromosome by flipping it (changing it from a zero to a one). This is done based on a low random probability (probability mutation). This done to provide some diversity within the population.

After this the population will be passed back to the evaluation for a new cycle.

PSO

A particle swarm algorithm is based on co-operation. Each particle will exchange information about what they have found (places visited) to particles in the same neighbourhood. Every practical in a neighbourhood knows the fitness of the overs in its neighbourhood and the position of the one with the nest fitness. In each iteration a particle will move towards the “best position” that it knows. To do this it uses velocity which is its current velocity + A weighted random portion in the direction of its personal best + A weighted random portion in the direction of the neighbourhood best. Once it has worked out this velocity the new position is its old position plus the new velocity. (information based on lecture notes).

Note *For GA I went through the 10 thousand first and then the 1 million*

1 Million algorithm calibration process – GA

I wanted to work out the best value for each parameter so I could do a large sweep fitness. I did this by creating an array of different values I wanted to test for each parameter. In the end I had a system of 4 for loop. For each combination I value I ran it through 4 different builds of the engine (the 4 combinations of Mutator or Gaussian Mutator and Uniform Crossover or Single Point Crossover) 30 times to create an average. Unless I have specified assume, I have kept the default values in for a certain check. I had systems in place to write all a range of different checks to a CSV file.

In each of my sweeps I did personally a binary check after I got each of my results (I would do a wide sweep of values and then do a more precious check around the more optimal values.

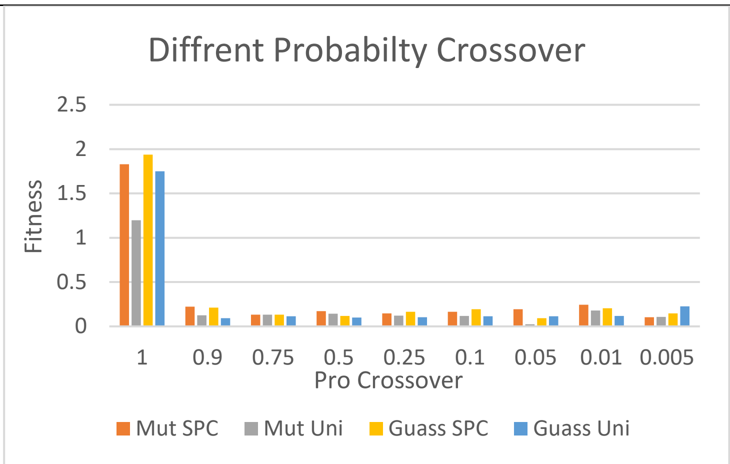
I decided to find the optimal values for each parameter by testing different options against the default. This is so that I could get values to run a parameter sweep at the end

1 Mill – GA probability crossover

I did a parameter sweep PC with the following values.
PC
1 ,0.9 , 0.75 , 0.5 , 0.25 , 0.1 , 0.05 , 0.01 , 0.005 ,0.001

diffrentPC	Mut SPC	Mut Uni	Guass SPC	Guass Uni
1	1.828639	1.1985	1.93672	1.750763
0.9	0.221879	0.1265	0.209988	0.093687
0.75	0.132849	0.133352	0.131786	0.114728
0.5	0.172312	0.143691	0.1163	0.099123
0.25	0.148187	0.121335	0.163654	0.104562
0.1	0.166247	0.116462	0.193182	0.115169
0.05	0.193302	0.022261	0.093373	0.113564
0.01	0.242346	0.178962	0.205714	0.116391
0.005	0.103696	0.105514	0.147916	0.227418
0.001	0.12602	0.148851	0.185377	0.183617

From my results it looks like the best value for this parameter the best value is 0.9 if using GuassUni and 0.05 if using MutUni.



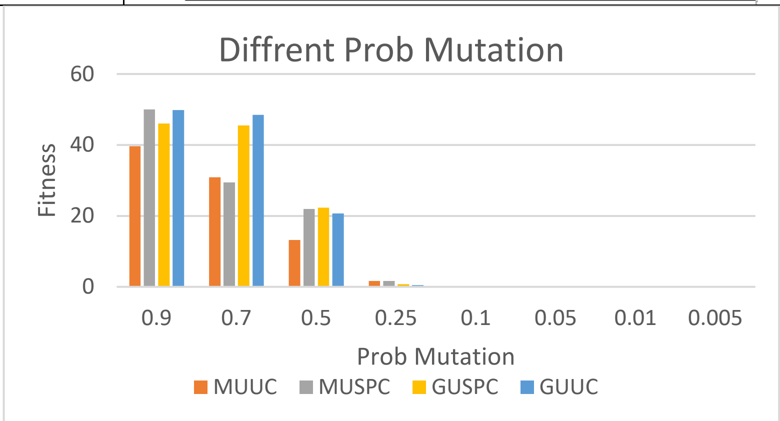
1 Mill - Probality Mutation

For this test I kept all the other values as the default, but I did a parameter sweep PC with the following values.

,0.9 , 0.7 , 0.5 , 0.25 , 0.1 , 0.05 , 0.01 , 0.005 ,0.001

ProbM	MUUC	MUSPC	GUSPC	GUUC
0.9	39.67365	50.05405	46.02366	49.83987
0.7	30.92008	29.43241	45.4988	48.4431
0.5	13.24358	21.99593	22.30239	20.66401
0.25	1.673805	1.626574	0.76148	0.470562
0.1	0.003048	0.007531	0.003416	0.001483
0.05	0.00375	0.003343	0.001468	0.001019
0.01	0.013915	0.030308	0.006646	0.002318
0.005	0.025845	0.019897	0.005765	0.040774
0.001	0.080459	0.548302	0.094328	0.107264

From my results it looks like the best value for this parameter the best value is between 0.1 and 0.05

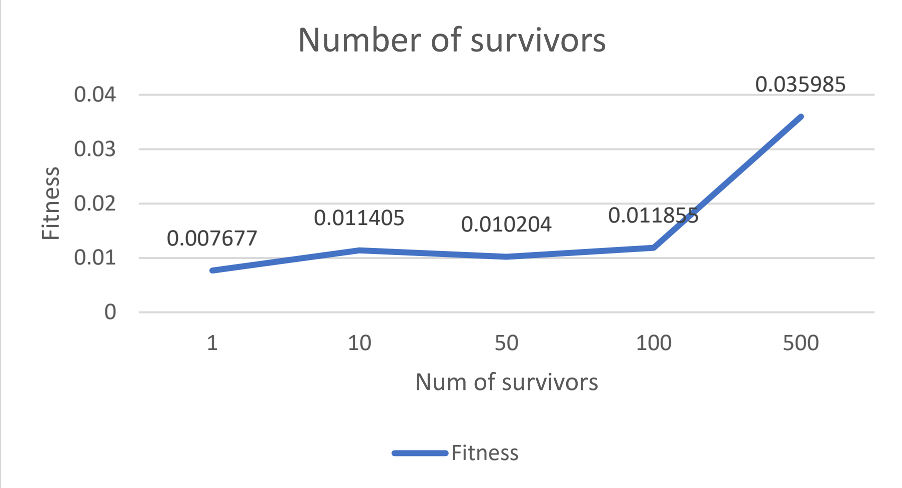


1 Mill - Num Survivors

sweep with the following values.
1, 10 , 50 ,100 ,500

NumSurvivors	Engine	Fitness
1	GuassMutUni	0.007677
10	GuassMutUni	0.011405
50	GuassMutSPC	0.010204
100	GuassMutUni	0.011855
500	GuassMutUni	0.035985

From my results it looks like the best value for this parameter the best value is 1



Tournament size

For this I wanted to test is 2 as it the smallest possible value that can be used) 5% of population , 10% , 25% , 50% , 100%. However, I couldn't find a standout case. To further test this I ran it along side Number of survivors. From this I discovered that in general Tournament size didn't really affect the fitness all too much.

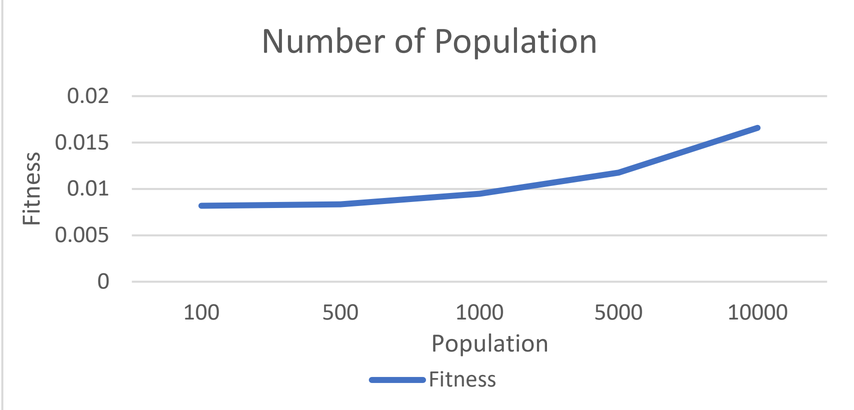
1 Mill - Num and Iterator parameters

First I did a sweep with the following combinations .

PopSize	NumIters
100	10000
500	2000
1000	1000
5000	200
10000	100

PopSize	NumIters	Engine	Fitness
100	10000	GuassMutSPC	0.008185
500	2000	GuassMutUni	0.00833
1000	1000	GuassMutSPC	0.009474
5000	200	GuassMutUni	0.01174
10000	100	GuassMutUni	0.016581

From my results it looks like the best value is around 100 population (10000 iterators) and 500 population (2000 iterators)



Final parameter sweep.

PopSize	{20000, 10000 , 8000 , 6660 , 5000 , 2000}
NumIters	{50 , 100 ,125 , 200, 500}
ProbMutation	{0.1 , 0.07 , 0.05 , 0.3}
probCrossover	{0.9, 0.8};
Numberofsurvivors	{1 , 5% ,10%}
Tornsize	{2 , 5% ,10% , 25% , 100%}

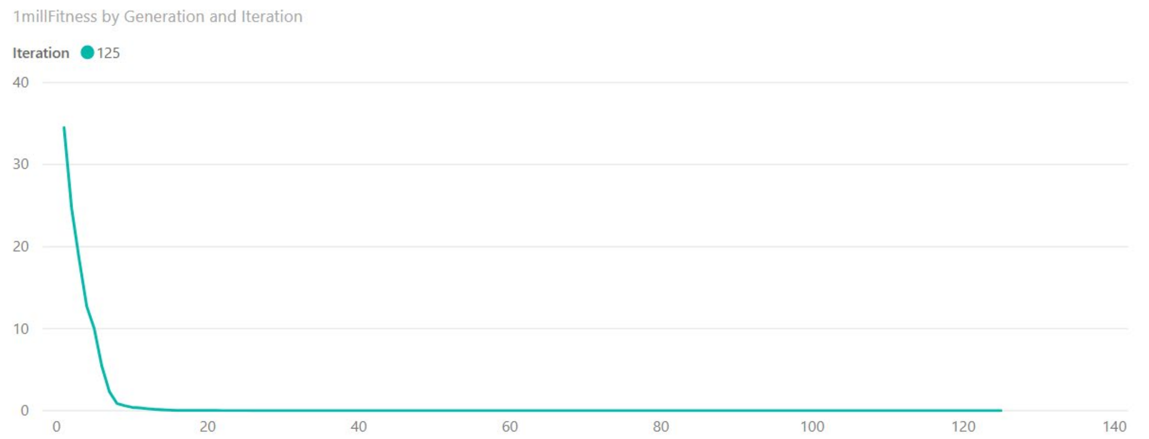
This is the best value that I got

The best value I got when the parameter sweep completed was.

PopSize	NumIters	NumSurvivors	TournamentSize	ProbM	probC	Engine	Fitness
8000	125	1	8000	0.1	0.9	GUUC	1.52E-04

I am saying that this is my most optimised.

1 Million fitness graphs




```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=470.679550145000 s; mean=3.765436401160 s      |
|              Altering: sum=2.452638095000 s; mean=0.019621104760 s        |
|   Fitness calculation: sum=0.379254767000 s; mean=0.003034038136 s        |
|     Overall execution: sum=473.312512605000 s; mean=3.786500100840 s      |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 125                                                |
|               Altered: sum=2,794,546; mean=22356.368000000                |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=28; mean=0.026538; var=0.037882                |
|               Fitness:                                                    |
|                      min  = 0.000187659281                                |
|                      max  = 225.936152080270                              |
|                      mean = 17.024064526354                               |
|                      var  = 792.461163072705                              |
|                      std  = 28.150686724709                               |
+---------------------------------------------------------------------------+
```

After refining I got the following results for 1 million. The minimum value that I achieved from this run was 0.00018 with the mean being 17. The overall execution took 473 seconds with mean of each step taking 3.7 seconds. A large portion of the time was taken by selection. The age of the oldest particle was 28 and the average lifespan of each particle. The generation in which I got the best value was 124 and I got a good solution at the 96st generation (longest life span).

1 million -PSO Calibration

I modified the class to allow me to do a large sweep for a wide range of parameters of each values for example I could pass in 5 different values of Neigh Weight and 4 different Global Weight and it would run 30 time for each combination and get an average best fitness for each of them.

For the optimisation I made sure the parameters (neigh Weight, inertia Weight, personal Weight and max MinVelocity) where between 0 and 4 following the guidance Michael Meissner [2006] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1464136/

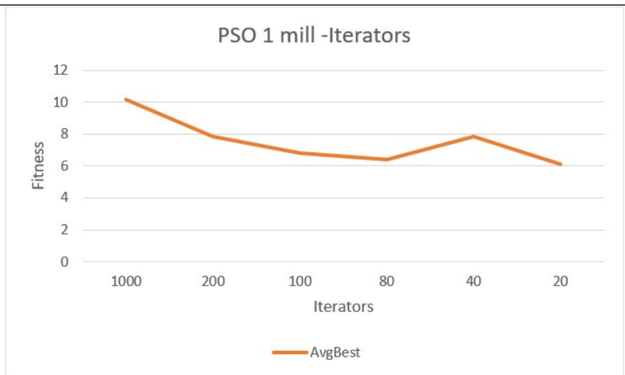
First, I wanted to test the different number of particles and iterations.

NumParticles = {1000, 5000, 10000, 12500, 25000, 50000}

NumIters = {1000, 200, 100, 80 , 40 , 20};

and got these results.

numParticles	numIters	AvgBest
1000	1000	10.13535
5000	200	7.817406
10000	100	6.793177
12500	80	6.380299
25000	40	7.831152
50000	20	6.081828



This suggests that a low number of Iterator high Particle will probably get the best results.

I also wanted to test low number of particles and got the following.

NumParticles = {10, 50, 100, 250, 500, 1000}
NumIters = {100000, 25000, 10000, 4000 , 2000 , 1000}

numParticles	numIters	AvgBest
10	100000	28.02459
50	25000	16.02412
100	10000	17.35762
250	4000	15.10056
500	2000	12.7435
1000	1000	11.30052

This shows that a low iterators count is probably best.

Next, I wanted to get an understanding of the engine. So, I did a parameter sweep of the following. I did this without modifying the number particles / iterators as I wanted to get just developed a understanding of how everything related to one another.

NumParticles	1000
NumIters	1000
InertiaWeight	2 ,1, 0.5
PersonalWeigh	2 ,1, 0.5
GlobalWeight	2 ,1, 0.5
MaxMinVelocity	1 , 0.01, 0.001 , 0.0001 , 0.00001

Here are the best results as I found them.

numParticles	numIters	neighWeight	inertiaWeight	personalWeight	globalWeight	maxMinVelocity	AvgBest
1000	1000	2	2	2	2	1	63.95944
1000	1000	2	2	2	1	1	44.2998
1000	1000	2	2	2	1	0.01	12.46332
1000	1000	2	2	2	0.5	0.01	2.593492
1000	1000	2	2	2	0.5	0.001	1.296799
1000	1000	2	2	1	1	0.01	0.284134
1000	1000	2	0.5	2	0.5	1	0.23221

This suggested a high MaxMin velocity, low global and inertia but a high neigh and personal weight.

To refine further I tested the following values

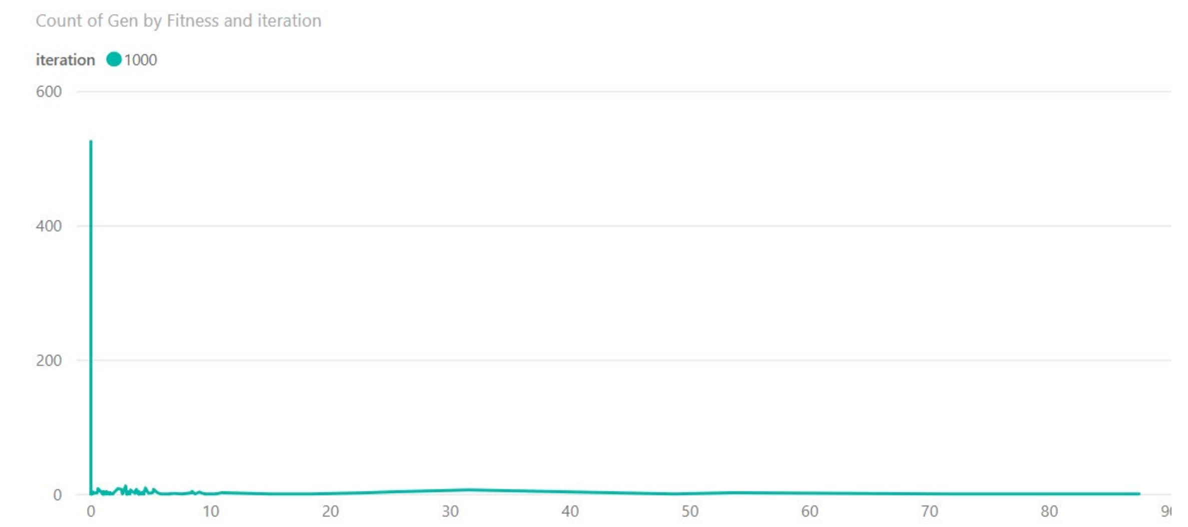
NumParticles	1000
NumIters	1000
InertiaWeight	2, 1.5 ,1, 0.5, 0.1
PersonalWeigh	2 ,1.5, 1 ,0.5,0.1
GlobalWeight	2, 1.5, 1, 0.5, 0.1
MaxMinVelocity	1 , 0.01 , 0.001 , 0.0001 , 0.00001

Num Particles	Num Iters	Neigh Weight	Inertia Weight	Personal Weight	Global Weight	maxMin Velocity	AvgBest
1000	1000	2	1.5	2	0.1	0.01	0.206319
1000	1000	2	1.5	1.5	0.5	0.01	0.15725
1000	1000	2	1	2	0.1	0.01	0.110004
1000	1000	1	0.5	2.5	0.1	1	0.0

As you can see, I came across 0 which is the best possible fitness so I decided to stop modifying as it I couldn't get a better value.

PSO 1 Million - Performance

Graph



Status
Time : 1.24 Seconds
Best : 400th generation
Greatest : 400th generation
Best fitness : 0.0

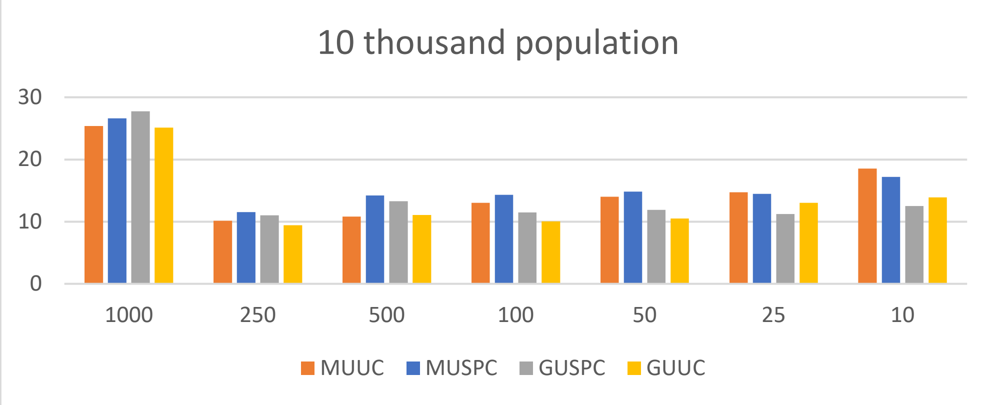
10 thousand GA calibration

10 thousand GA - Population * Num iterators

I tested the following combination in such a way that PopSize[i] * DefaltNumIters[i] = 10,000.

PopSize	NumIters
1000	10
250	40
500	20
100	100
50	200
25	400
10	1000

PopSize	NumIters	MUUC	MUSPC	GUSPC	GUUC
1000	10	25.38873	26.63173	27.75914	25.13373
250	40	10.11783	11.54679	10.99282	9.399862
500	20	10.81382	14.22926	13.27525	11.04561
100	100	13.04567	14.31774	11.48807	10.05193
50	200	14.00745	14.82532	11.87347	10.48926
25	400	14.70733	14.46194	11.24793	13.00195
10	1000	18.50926	17.21341	12.4892	13.90171



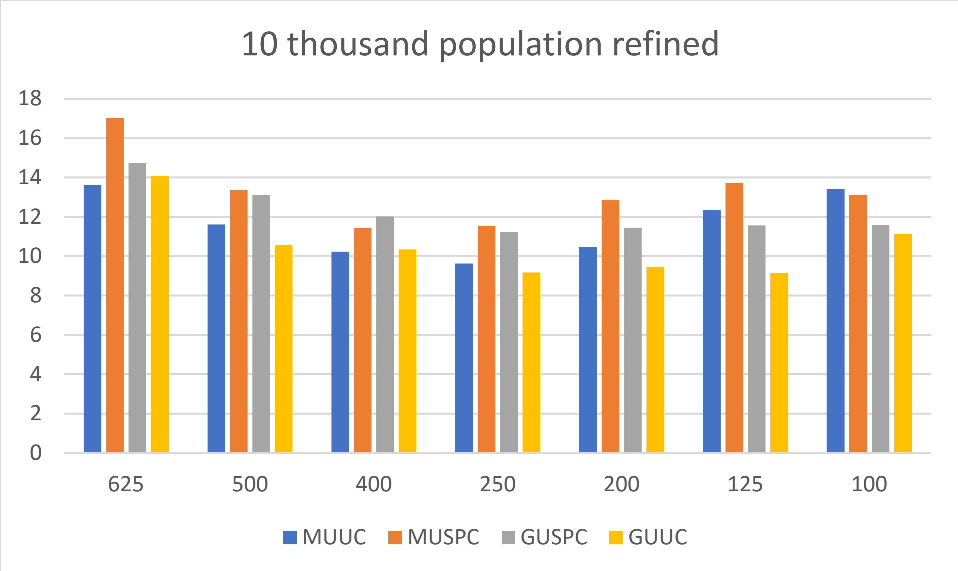
10 thousand GA - Population * Num iterators - Refined

Following the last test I refined it to

PopSize	NumIters
625	16
500	20
400	25
250	40
200	50
125	80
100	100

PopSize	NumIters	MUUC	MUSPC	GUSPC	GUUC
625	16	13.63442	17.03293	14.73483	14.08752
500	20	11.60983	13.35875	13.10081	10.55413
400	25	10.22979	11.43544	12.02693	10.34647
250	40	9.623056	11.55357	11.2362	9.169481
200	50	10.46475	12.87136	11.45058	9.462305
125	80	12.35784	13.72396	11.56021	9.136695
100	100	13.39771	13.12967	11.58207	11.14728

From this it seems that the most optimal lies between (400 ,25) and (125,80).



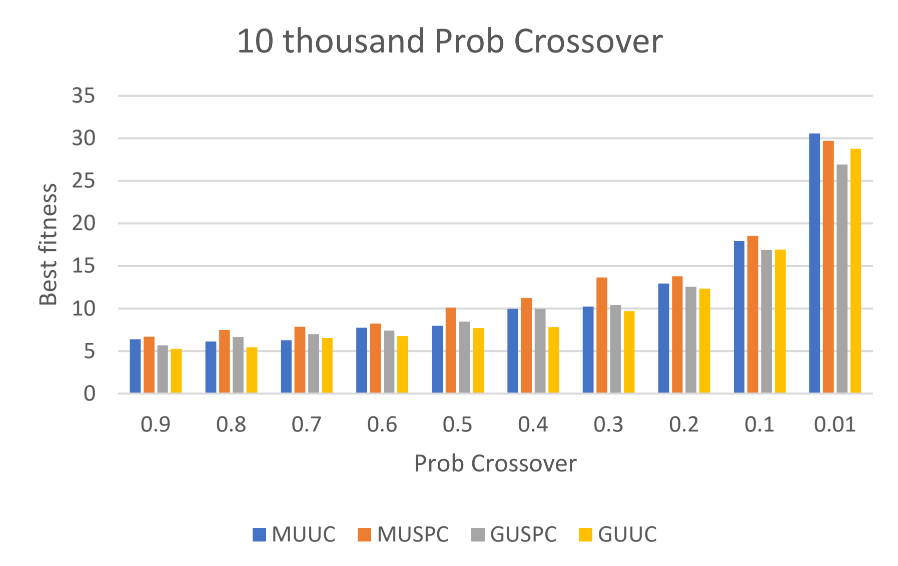
I also tested for every combination of PopSize * DefaltNumIters where it equal less than 10000 but found the best values where the ones where the total equals 10000

10 thousand GA – Probabilty of crossover

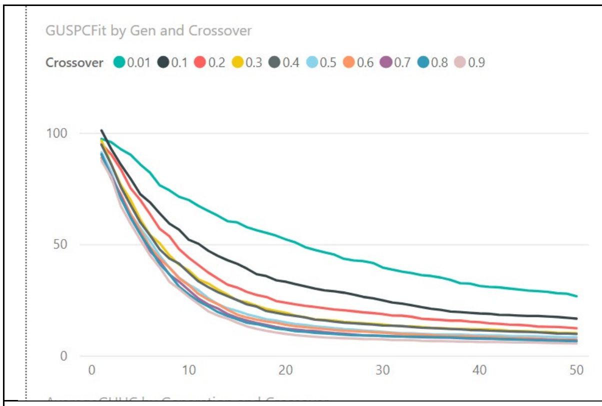
double[] DefaultprobCrossover = {0.9 , 0.8 ,0.7 ,0.6 , 0.5 , 0.4 , 0.3 , 0.2 ,0.1 , 0.01};

probC	MUUC	MUSPC	GUSPC	GUUC
0.9	6.36842	6.691014	5.6894	5.246498
0.8	6.131937	7.469896	6.645829	5.430267
0.7	6.287361	7.856091	7.006588	6.539426
0.6	7.739002	8.228144	7.403253	6.765123
0.5	7.95442	10.1019	8.450004	7.70421
0.4	9.944731	11.23563	9.957061	7.8229
0.3	10.22059	13.65317	10.39574	9.692443
0.2	12.94757	13.81573	12.54812	12.33343
0.1	17.94334	18.55103	16.87892	16.91071
0.01	30.57236	29.73084	26.91514	28.78235

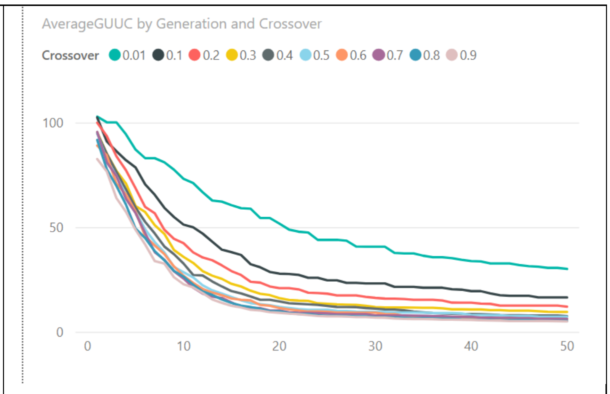
From these results It looks like the optimal crossover is 0.9 (I didn't set it to 1 as that could cause errors). I tested this with the default values as well and found 0.9 was the best in that case as well.



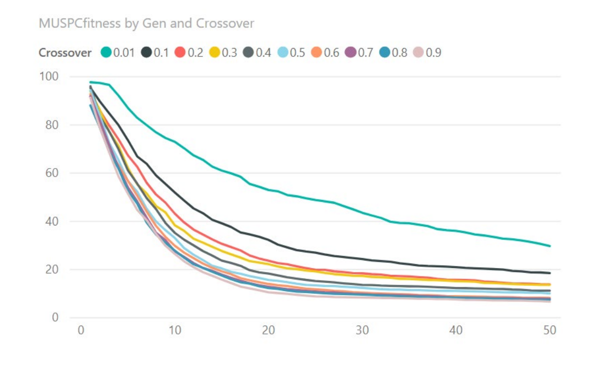
Graphs gotten from results



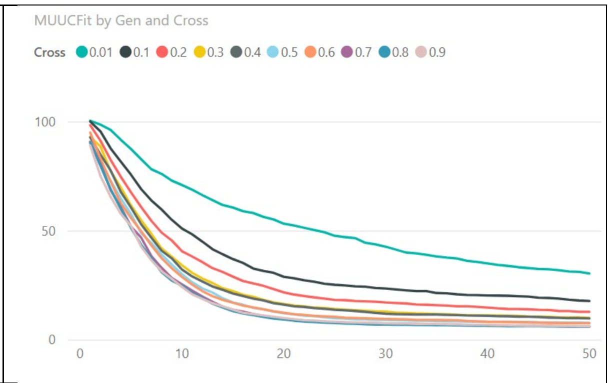
GUSPC



GUUC



MUSPC



MUUC

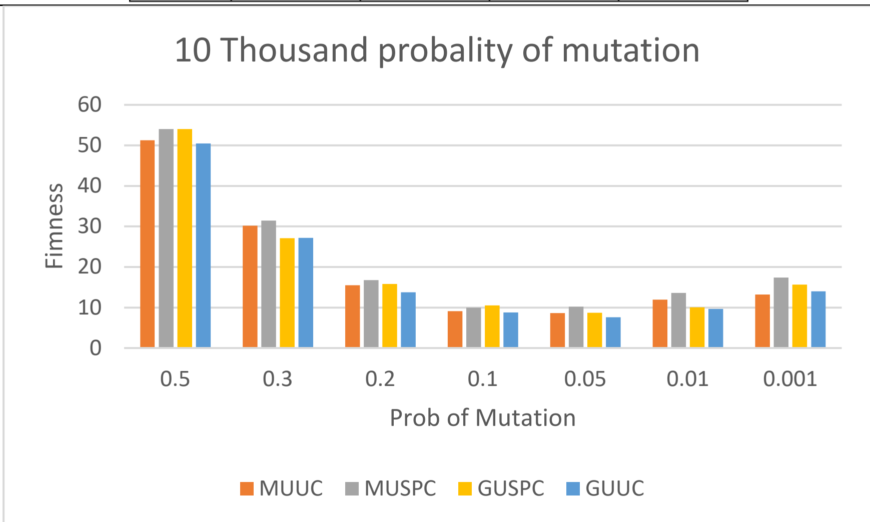
I decided to keep the default crossover for now (0.3). I than ran the following

10 thousand GA – Probabilty of Mutation

```
double[] DefaultprobMutator = {0.9 , 0.8 ,0.7 ,0.6 , 0.5 , 0.4 , 0.3 , 0.2 ,0.1 , 0.01};
```

ProbM	MUUC	MUSPC	GUSPC	GUUC
0.5	51.26296	54.0086	54.02856	50.4763
0.3	30.20946	31.45769	27.11482	27.17214
0.2	15.46833	16.72411	15.77896	13.77703
0.1	9.090659	9.969239	10.53073	8.759928
0.05	8.646037	10.21508	8.728491	7.604414
0.01	11.97154	13.59765	10.07876	9.664906
0.001	13.17211	17.38319	15.65329	14.02

Based on the results it seem the best value is between 0.1 and 005. I decided to further refine these

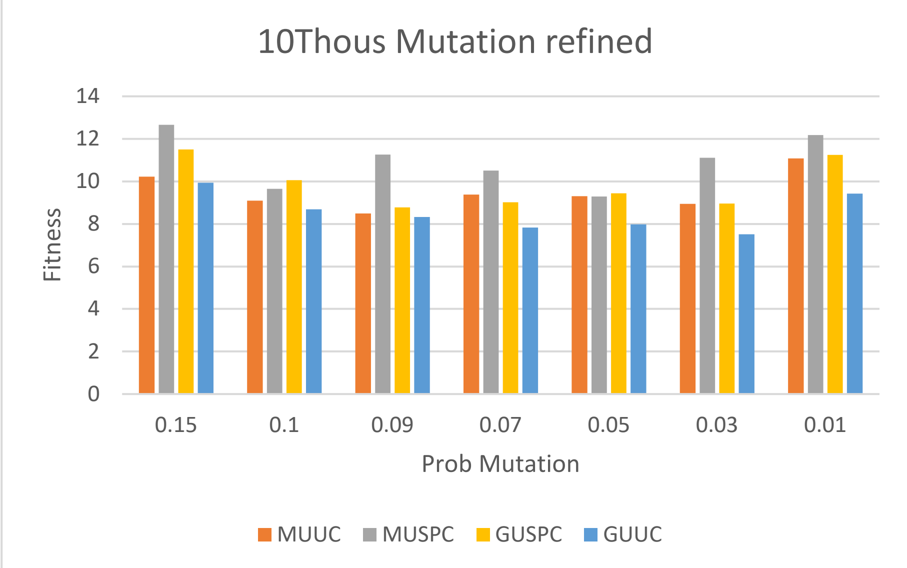


10 thousand GA – Probabilty of Mutation refined

double[] DefaultprobMutator = 0.15, 0.1 , 0.09 , 0.07 , 0.05 ,0.03 ,0.01

ProbM	MUUC	MUSPC	GUSPC	GUUC
0.15	10.21837	12.65936	11.49398	9.93726
0.1	9.100582	9.652729	10.05435	8.684422
0.09	8.485323	11.25616	8.775445	8.324116
0.07	9.380381	10.50514	9.014876	7.827887
0.05	9.302436	9.288215	9.432448	7.976002
0.03	8.944953	11.10448	8.964803	7.510412
0.01	11.07398	12.18367	11.23824	9.431405

This shows that between 0.05 and 0.003 would yield the best result.

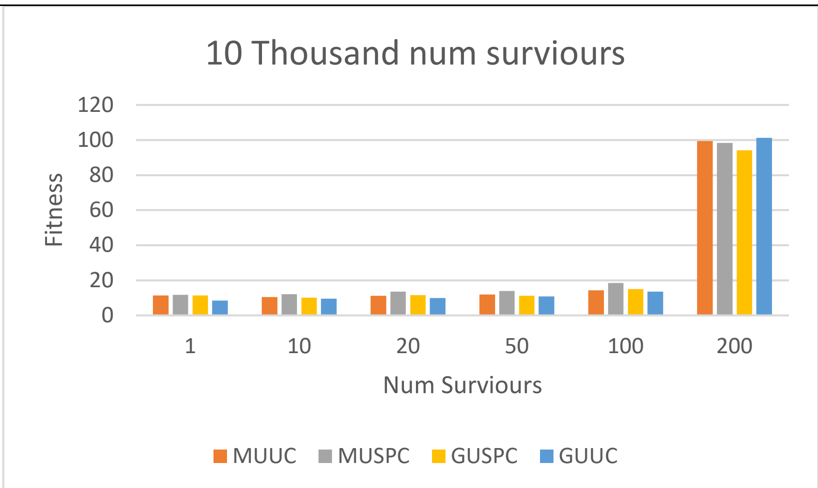


10 thousand GA – Num survivor

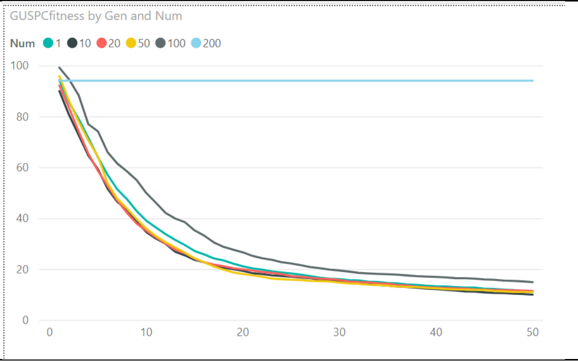
I did a sweep with the following Numberofsurvivors = { 1 , 10 , 20 , 50 , 100 , 200}; I chose these values as they are 1% , 5% , 10%, 25% , 50% and 100%.

NumSurvivors	MUUC	MUSPC	GUSPC	GUUC
1	11.39106	11.68055	11.35173	8.489048
10	10.49693	12.18297	10.12713	9.580678
20	11.15606	13.55513	11.5316	9.996146
50	11.93654	13.98675	11.10386	10.81793
100	14.36451	18.51638	15.04286	13.46568
200	99.47846	98.37598	94.22134	101.2581

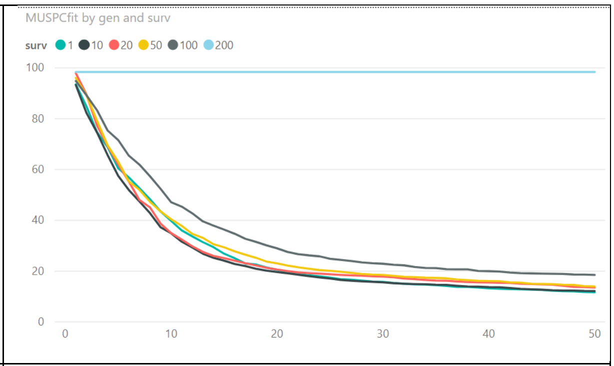
This suggest that between 1 and 5% would give the best results.



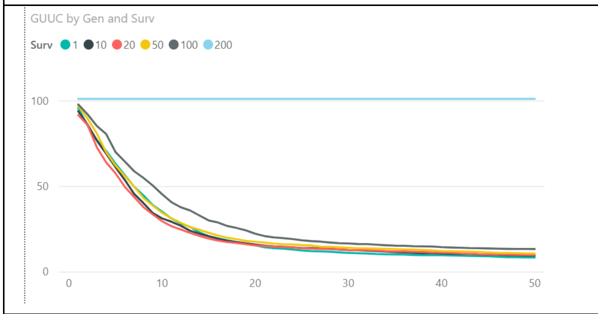
Graphs gotten from results



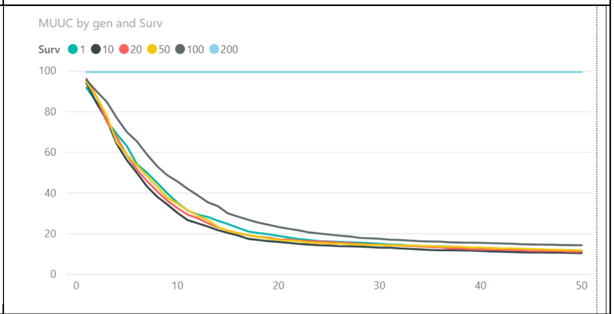
GUSPC



MUSCP



GUUC



MUUC

From these results it looks like the best value is 1 to 10% percent of survivors

Tournament Size

DefaultTournamentSize = { 2 , 10 , 20 , 50 , 100 , 200 };

I chose these values as they are 2(the smallest possible), 5% , 10%, 25% , 50% and 100%

TournamentSize	MUUC	MUSPC	GUSPC	GUUC
2	10.68594	12.44531	11.22277	10.20898
10	11.62545	11.30191	10.70092	9.531194
20	12.65065	12.29825	11.58441	10.36154
50	13.23482	15.76979	11.70191	11.04174
100	12.37876	14.63526	10.26844	10.60268
200	13.44339	14.74644	11.34611	11.00533

I haven't produced the graph for this test as its difficult to tell the results

I couldn't really find a standout value for what tournament size should be. I decided to run a test comparing looping different parameters values of Numberofsurvivors = { 1 , 10 , 20 , 50 , 100 , 200} with DefaultTournamentSize = { 2 , 10 , 20 , 50 , 100 , 200 }; From this test I found for any given tournament size then best number of survivors will always be 1 or 5%.

10 Thousand refined.

The results from the previous test I have worked out to get the optimal value then

- Population Size should be between 400 and 125
- NumIters should be between 25 and 80;
- Probability of mutation should be around 0.9
- Probability of crossover should be between 0.1 and 0.03
- Numberofsurvivors should be between 1 and 10%
- Tornsize should be between 2 and 50%

I ran the following parameter sweep.

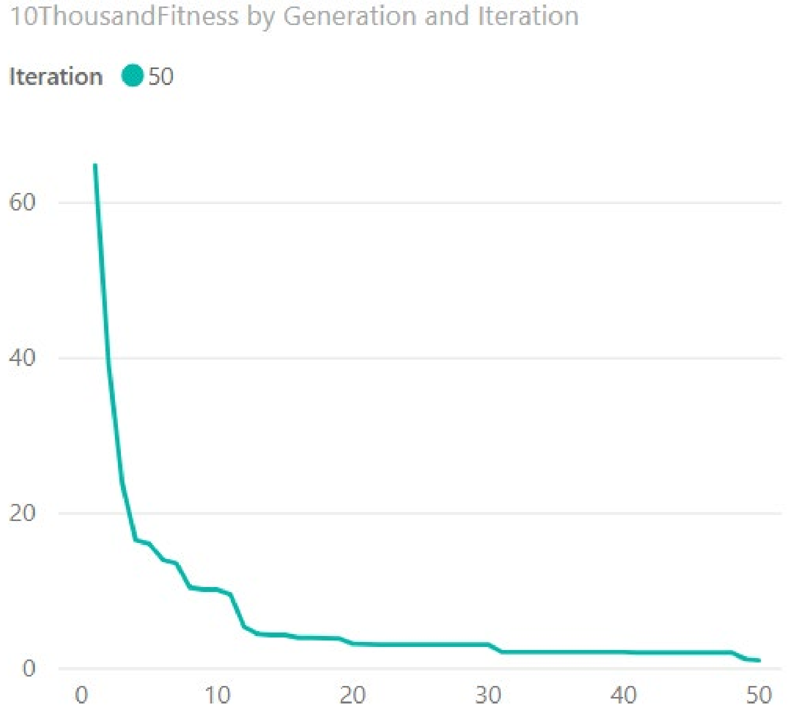
Population Size	400,250,200,150
NumIters	25,40,50,60,80
Probability of mutation	0.1 , 0.07 , 0.05 , 00.3
Probability of crossover	0.9, 0.8
Number of survivors	1 , 2% ,5% ,10% , 20%
Torn size	2 , 5% ,10% , 20% ,25% , 100%

And got the following result

PopSize	NumIters	NumSurvivors	TournamentSize	ProbM	probC	Engine	Fitness
200	50	5	200	0.1	0.9	GUUC	1.347525

This was the best value that I found for ten thousand.

Graph of ten thousand



System stats

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|              Selection: sum=0.082814302000 s; mean=0.001656286040 s       |
|               Altering: sum=0.078490757000 s; mean=0.001569815140 s       |
|    Fitness calculation: sum=0.038484947000 s; mean=0.000769698940 s       |
|      Overall execution: sum=0.207671262000 s; mean=0.004153425240 s       |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|            Generations: 50                                                |
|                Altered: sum=27,032; mean=540.640000000                    |
|                 Killed: sum=0; mean=0.000000000                          |
|               Invalids: sum=0; mean=0.000000000                          |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                    Age: max=11; mean=0.111600; var=0.512197               |
|                Fitness:                                                   |
|                       min  = 1.077559457635                               |
|                       max  = 206.036989939528                             |
|                       mean = 23.404613724419                              |
|                       var  = 967.178393180899                             |
|                       std  = 31.099491847632                              |
+---------------------------------------------------------------------------+
```

The minimum value that I achieved from this run. The overall execution took 0.208 seconds with mean of each step taking 0.004 seconds. Selection and Altering took roughly the most time. The age of the oldest particle was 11 and the average lifespan of each particle as 0.1 . The best possible value was gotten at the 50th generation and the point I got a good solution was at the 31st generation (longest life span). The engine set up which produced the best value was GUUC.

PSO 10 thousand calibration

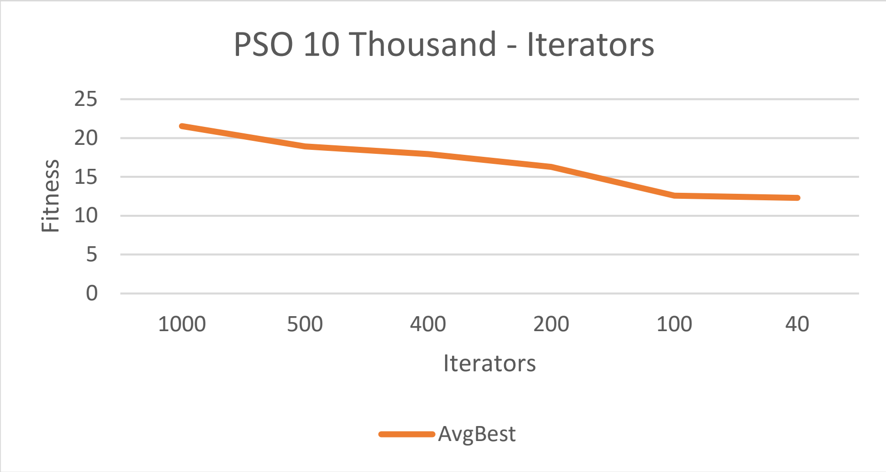
First, I decided to see what a good balance of number of Particales would be and number of Iters

I tested the following. So that NumParticles[i] * NumIters[i] = 10000

NumParticles	10 , 20 ,25, 50 ,100 ,250, 500
NumIters	1000 , 500, 400 , 200 , 100 ,40,20

These are my results

numParticles	numIters	AvgBest
10	1000	29.51845
20	500	21.53345
25	400	18.9256
50	200	17.92269
100	100	16.30127
250	40	12.60258
500	20	12.30201



The results suggest that high particles low iterators would be best

Next, I investigated how changing the max min velocity effects the fitness of each particles / iters pairs. I ran these against the different numParticales and numIters I tested in the last step to see if I could find any changes.

diffrentMaxMin	10 , 20 ,25, 50 ,100 ,250, 500

Here are the top 5 best results

numParticles	numIters	maxMinVelocity	AvgBest
25	400	0.1	18.92295
25	400	0.001	17.79691
100	100	0	15.48459
100	100	0.1	12.63062
500	20	0.001	12.07045

PSO refined

Finally, I decided to do a large parameter sweep including everything and testing every single combination of the following. I wanted to do this as PSO uses some randomness.

diffrentNumParticles	500, 250 ,200 ,100 , 50 , 40, 25 , 20
diffrentNumIters	20, 40 ,50 , 100 , 200 , 250 , 400 , 500
diffrentNumParticles	3 ,2.5 ,2 ,1.5 , 0.75 ,1 , 0.5
diffrentInertiaWeight	3 , 2, 1.5 ,1 ,0.75, 0.5 , 0.01
diffrentPersonalWeigh	3 , 2, 1.5 ,1 ,0.75, 0.5 , 0.01
diffrentGlobalWeight	3 , 2, 1.5 ,1 ,0.75, 0.5 , 0.01
diffrentPersonalWeigh	1 , 0.1 , 0.01, 0.001, 0.0001 ,0.00001

Here are my top 5 values gotten from my sweep.

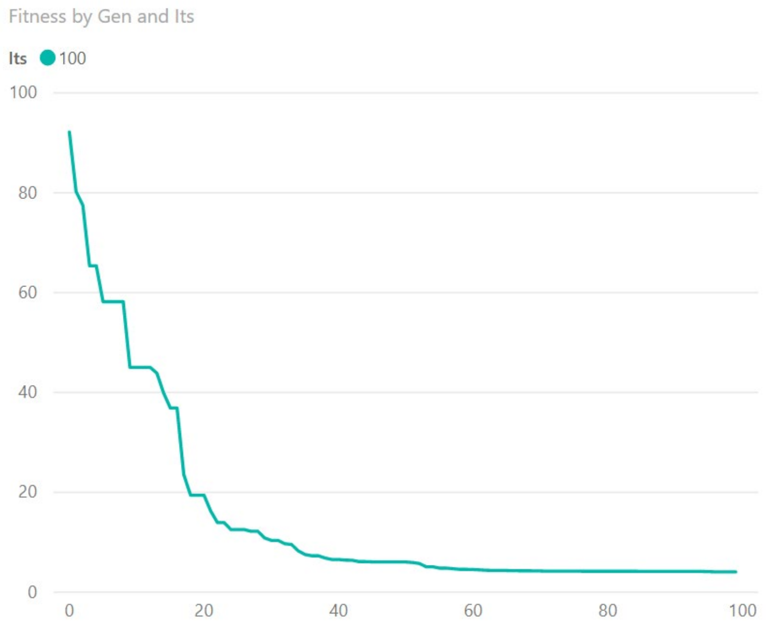
numParticles	numIters	neighWeight	inertiaWeight	personalWeight	globalWeight	maxMinVelocity	AvgBest
250	40	0.75	0.75	0.75	0.01	0.1	5.550738
200	50	0.5	0.5	2.5	0.5	0.1	5.544215
100	100	0.5	0.5	2.5	0.5	0.001	5.283335
100	100	0.75	0.75	2	0.01	0.1	5.259183
100	100	0.75	0.75	1.5	0.01	0.1	4.753735

My results suggest that the optimal values for PSO are an identical value of Particles and Iters, a low neighWeight , a low inertia weight and higher personal weight , an extremely low global weight and a high max min velocity.

I ran the best value I got again and got the following: I have decided this is my best result.

numParticles	numIters	neighWeight	inertiaWeight	personalWeight	globalWeight	maxMinVelocity	AvgBest
100	100	0.75	0.75	1.5	0.01	0.1	4.045685

Graph



Time : 0.46 Seconds

Best : 100th generation

Greatest : 76th generation

Best fitness : 4.045685

Algorithm comparisons

Both these algorithms have similar properties, the initial population is randomly generated, and both use fitness values to evaluate the population and search for an optimal value using random techniques. However, PSO doesn't use genetic operators (mutation and crossover). PSO particles have memory and use internal velocity. They way information is shared between the two is also very different. In GA the chromosomes share information with each other so that the whole population moves towards an optimal value. Where as in PSO the best particle gives its information to the other and all particles will start and move towards that value.

Unknown [http://www.swarmintelligence.org/tutorials.php]

Personally, GA was much easier to modify to find the optimal value as all the genetic operators where mostly detached from one another so could quite find the optimal parameter value easier. In PSO all the weight where linked so modifying one value could also affect the others.

1 Million Comparison	GA	PSO
Best solution	1.54E-4	0.0
Time to optimal	473	1.24 Seconds
number of iterations for best	124th	400th generation
number of iterations for greatest	96st	400th generation
Iterations	125	1000

When compared to one another my run of PSO is the best for working out fitness with a 1 million budget. It got the best solution; it was a lot faster even though it had almost 10* the iteration of GA. This is because GA had to spend a lot of time altering whereas once a PSO particles found 0 it alerted all the other particles to come to towards it then sat at zero.

10 Thousand Comparison	GA	PSO
Best solution	1.347525	4.045685
Time to optimal	0.208	0.46
number of iterations for best	50th	100th generation
number of iterations for greatest	31st	76th generation
Number of iterations	50	100

When compared to one another my run of GA got the best for working out fitness with a 10 thousand budget. It got the best solution by a factor of 4, it had shorter time to optimise, it required a low number of iterations and the best fitness at an earlier iteration.

Comparing the two to one another shows that for higher budgets then PSO would probably be the best algorithm to run but for low budgets then GA would probably be the better option. What should also be mentioned is that based on the graphs PSO would get hung up at particular values or vary a bit at the start whilst it's trying to find an optimal point whereas GA was always moving towards a more optimal value.