ECM3412 Nature Inspired Computing: Evolutionary Algorithms

Student ID: 670028137, Candidate Number: 020891

1 Experiments

In total, there were six experiments to run. Each of these experiments ran with a different permutation of parameter values. The parameters that could be tuned are as follows:

- M - The maximum number of genes that can be randomly changed by a mutation. Increasing this increases the impact of mutation. Setting this to 0 removes any mutation.
- P - The size of the population to maintain. Increasing this can make change slower as new mutations and offspring make a smaller proportion of the whole population.
- crossover - Determines whether any crossover of parents happens or not. If this is set to false, the algorithm effectively becomes a local search.

The algorithm described in the coursework specification is a Steady State evolutionary algorithm. This means that, unlike a generational algorithm, we maintain the population. We only add new children if they match the replacement criteria; in this case, being stronger than the weakest member of the population. Steady State algorithms are cheaper in terms of iterations, but make slower progress.

I have structured the code such that it is in its own Python module. I have then used Jupyter Notebooks to perform the experiments.

The table below represents the results of each experiment. I have included the average fitnesses of each experiment after the initial population is created. This gives a good indication of how well the algorithm has performed. Where the value of 'crossover' is not stated, it is true.

	Params	**M=1, p=10**	**M=1, p=100**	**M=5, p=10**	**M=5, p=100**	**M=5, p=10, No crossover**	**M=0, p=10**
BPP1:	Initial Avg	12,744.2	12,734.8	12,911	12,626	12,327.2	12,234
	Best	24	152	494	656	3,836	4,606
	Average	37.6	235.6	577.2	832.4	4,770.4	6,800
BPP2:	Initial Avg	2,419,050.2	2,384,575.5	2,373,918.6	2,383,927.1	2,379,126.6	2,332,741.7
	Best	670,878	925,664	531,232	928,890	1,613,086	1,809,324
	Average	732,194.8	1,097,719.2	604,266.4	1,041,460.4	1,630,180.8	1,868,654.0

In this specific instance of the Evolutionary Algorithm, we have used a fitness function where a lower score is better.

2 Questions

2.1 What combination of parameters produces the best results for BPP1 and BPP2?

For BPP1, the best set of parameters was M=1, p=10, crossover=true.

For BPP2, the best set of parameters was M=5, p=10, crossover=true.

2.2 Why do you think this was the case?

Both algorithms suffered when increasing the size of the population. I put this down to the combination of the fitness evaluation limit combined with the method of replacement chosen. Weakest Replacement requires that we have calculated the fitness of each member of the population so that we can (potentially) remove the weakest candidate. Increasing the size of the population dramatically increases the number of times that their fitness is evaluated as a result. Fewer overall iterations of the algorithm can take place. If the number of iterations allowed increased, or a different metric was used for counting iterations, increasing population would not result in a drop-off.

In general, increasing population size should increase the quality of a solution. This has the cost of increasing the computational resources required. Eventually a point is reached where increasing the population size has diminishing returns.

I ran the BPP1 with M=1, p=100 five times as before, increasing the number of iterations to 50,000. The program took considerably longer to run as expected, but resulted a better average of 32 (compared to 37.6 previously). We have almost certainly hit the point of diminishing returns of increasing the population size. This does explain the reduction in performance when increasing the population size without adjusting the maximum number of iterations, however.

BPP1 did best when the mutation rate was 1, whilst BPP2 did best when the mutation rate was raised to 5. We are using K-ary encoding to represent our chromosomes. K is, in this case, the number of bins in the problem. Where BPP2 has a higher value of bins, but the same number of items, it has far more possible solutions. Setting mutation rates is a trade-off between exploitation and exploration. We want to be able to keep the good parts of good solutions. At the same time, we also want to explore the whole solution space. BPP2's larger solution space requires a higher mutation value to traverse. In BPP1's comparably smaller solution space, a mutation value of 5 is too high and loses good solutions.

In order to verify this, I adjusted the problems as follows. BPP1 now has 100 bins instead of 10. BPP2 now has 10 bins instead of 100. The count of items has remained the same. I then tested both problems with M=1 and M=5. The results are in the table below, showing the average results after 5 trials:

	M=1	**M=5**
Adjusted BPP1 (bin-size=100)	1,502.4	1,644.4
Adjusted BPP2 (bin-size=10)	5,686.0	108,658.0

From this, we can see that to an even greater extent than before, increasing the value of M with a smaller bin-count results in significantly worse performance. The difference in a big bin of 100 seems to be marginal in this case.

2.3 What was the effect when you removed mutation? What about crossover?

Removing crossover and mutation both resulted in worse algorithm performance. In the case of removing crossover, the algorithm can only improve from mutation. This results in a luck-based result and does not use the strength of existing solutions to build better solutions.

In the case of removing mutations, the algorithm is reliant on luck of creating good solutions in the initial population. New strategies will not be generated through mutation.

This section serves to demonstrate that both mutation and crossover are essential parts of the Evolutionary algorithm. Without both working in conjunction, the algorithm will perform significantly worse.

2.4 What other nature-inspired algorithms might be effective in optimising the BPP problems? Explain your choice(s)

An algorithm that could be used is Ant Colony Optimisation. In order to do this, we need to represent the Bin-Packing problem as a graph traversal. We can do this by creating a construction graph that maps items to bins. We can then strengthen paths based on the fitness of the solution. Ant Colony optimisation is better for large problems. A bin-packing algorithm with more items, then, would be better approached by ACO.