Homework 2: EA for constrained optimizations

Jennifer Za

1 Minimal Requirements (5 pts)

As per instructions, I have implemented the Stochastic Ranking method and the NSGA-II algorithm in tackling Evolutionary Algorithm problem applied to instances covered in labs. The following two tables show results my stochastic ranking algorithm and NSGA-II respectively achieved when applied to instances g06, g08, g11 and g24. Sourcecode of these implementations can be found in stochastic_ranking.py and nsgaii.py respectively. Values presented for this and all subsequent parts are arithmetic averages achieved after 5 runs. Each run worked with populations of size 100 over 100 generations. To run algorithms on different test instances it suffices to change the global "model" variable to a string corresponding either to "g06", "g08", "g11" or "g24". I have opted to separate code for each section into different files to facilitate as easy as possible for one to run each "part" of the homework separately without much fuss. I hope that redundancy in the amount of code reused at least to some extent offsets clarity in use.

Stochastic Ranking				
name	My values	My fitness	ref. values	reference fitness
g06	(14.86, 3.2)	-4615	(14.095,0.843)	-6961.81
g08	(1.23, 4.2)	-0.096	(1.23, 4.2)	-0.096
g11	(1.33, 4.23)	-0.049	(0.7, 0.5)	0.749
g24	(1.22, 4.24)	-0.096	(2.33,3.18)	-5.51

As can be viewed in the table, my algorithm for the most part has been able to reach values close to the reference ones. This is with a few exceptions. One test case standing out is the g11 one, which is particular because its constraint is not an inequality, but ties variables to a specific value, hence it is much harder to achieve precision with an algorithm such as mine which values vary from generation to generation and always have an idiosyncratic element making them merely "approximate" an optimal value.

NSGA-II				
name	My values	My fitness	ref. values	reference fitness
g06	(14.15, 0.96)	-6827.2	(14.095,0.843)	-6961.81
g08	(1.23, 4.24)	–0.096	(1.23, 4.2)	-0.096
g11	(-0.82, 0.66)	0.79	(0.7, 0.5)	0.749
g24	((2.3, 3.06)	-5.36	(2.33,3.18)	-5.51

The table above compares the NSGA-II algorithm with respect to the same test instances. As can be observed, it too achieved quite close values to the reference ones. When it comes to g06 or g24 it outperforms stochastic ranking, whilst it lags behind slightly in the other ones.

2 NSGA-II with constrained selection (1 pts)

As part of a bonus task, I have implemented a constrained selection method within the NSGA-II algorithm. Code for this can be found in the file constrained_selection_nsgaii.py. Parameters used were the same as with the minimal requirement algorithm and its parameters be altered in the same fashion.

NSGA-II with constrained selection				
name	My values	My fitness	ref. values	reference fitness
g06	N/A	N/A	(14.095,0.843)	-6961.81
g08	(1.23, 4.25)	-0.0957	(1.23, 4.2)	-0.096
g11	(-0.98, 0.41)	1.311	(0.7, 0.5)	0.749
g24	(2.42, 2.52)	-4.95	(2.33,3.18)	-5.51

As can be seen from the table above describing values obtained, when it came to cases such as g08, the algorithm was nearly able to match reference values. In other instances, it performed worse than with the binary tournament operator contained in the minimal requirements algorithm. It should be noted that for the instance g06 my algorithm for some reason at times did not reach a feasible solution. Whilst these lapses happened sparsely and the algorithm being identical to the one with binary tournament operator with the exception of the said operator, I did not feel results for this particular instance were solid enough to report.

3 Other constraint handling method: Death Penalty (1 pts)

When deciding on which other constraint handling method to use, I took into consideration both sophistication of an algorithm along with quality of approximation it could provide. Whilst the death penalty approach which consists in simply removing solutions which are outside the bounds of constraints is widely-used and effective

for the most part, it may lead to the program running too long. This is what happened to me when I've tried to implement it. Whilst I have been successful in applying it to NSGA-II and stochastic ranking (as can be seen in files death_penalty_stochastic_ranking.py and death_penalty_nsgaii.py respectively), I did not attain values which would be considerably better than those achieved in other parts of this homework. For illustration the table below shows the NSGA-II algorithm with death penalty implemented. Both Stochastic ranking and NSGA-II didn't prove to be more effective with death penalty. On the contrary, their runtimes increased substantially and did not provide significant improvements.

NSGA-II: Death Penalty				
name	My values	My fitness	ref. values	reference fitness
g06	(14.18, 1.01)	-6770.6	(14.095,0.843)	-6961.81
g08	(1.22, 4.25)	-0.096	(1.23, 4.2)	-0.096
g11	(0.57, 0.3)	0.813	(0.7, 0.5)	0.749
g24	(2.34, 2.95)	-5.3	(2.33,3.18)	-5.51

4 More complicated methods (2 pts)

I have also adjusted both methods to more complicated test instances, namely the g05 example. Codes for these can be found in files g05_stochastic_ranking.py and g05_nsgaii.py. With these, I struggled trying to find optimal values, or any values, which would satisfy all constraints. Whilst my algorithms succeeded in minimising g1 and g2 constraints, which I suspect is because these involved inequalities, it usually missed the other two constraints which functional value had to equal zero. I believe that attaining values closer to the reference ones missed the mark first, because my algorithm has some idiosyncratic sides to it in trying to mutate values but also perhaps because I have used sums of "breaches" of constraints as parameters in both stochastic ranking and nsga-ii and perchance the fact that these values were pooled and not separated, the algorithm did not have enough space to adjust each variable adequately.

NSGA-II: g05		
variable /function	My values	ref. values
x1	0.62	679.9
x2	2.97	1026.07
x3	-0.53	0.12
x4	-0.51	0.4
f()	7.795	5126.5
g1()	-0.58	0
g2()	-0.52	0
g3()	1430.9	0
g4()	-90.8	0

Stochastic Ranking: g05		
variable /function	My values	ref. values
x1	0.48	679.9
x2	2.35	1026.07
x3	-0.54	0.12
x4	-0.53	0.4
f()	6.15	5126.5
g1()	-0.56	0
g2()	-0.54	0
g3()	1455.6	0
g4()	-74.4	0

5 Alternative EA- Multi-Objective Genetic Algorithm

Whilst pondering on which algorithm to use in order to compare some different multi-objective results to those achieved via NSGA-II in previous sections, I stumbled upon the Multi-Objective Genetic Algorithm presented in (Genetic Algorithms for Multiobjective Optimization: Formulation Discussion and Generalization by M. Fonseca and P.J. Fleming, published in 1999), which seemed to be a nice counterpart to the multi-criterion version of the NSGA-II algorithm. Here, as opposed to ranking solutions by how many other solutions they dominate, it is the reverse. We count the number of solutions which dominate said solution. In their paper, Fonseca and Fleming describe two ways of implementing their algorithm. One uses a rank-based fitness assignment method and the other one a niche-formating method. Whilst I have implemented both, as can be seen in the code for this part (moga.py), the former outperformed my implementation of the latter. I attribute this to my inability to find a mistake I must have made whilst implementing the niche method. In spite of this, I have selected a costumised rank method where I have out fitness to be a linear function of result given by the objective and number of other solutions that dominate each solution. In the code, I have multiplied the number of "dominating other solutions" for each by two and the table below describes my findings.

Multi-Objective Genetic Algorithm				
name	My values	My fitness	ref. values	reference fitness
g06	(14.74, 2.56)	-5198.78	(14.095,0.843)	-6961.81
g08	(1.22, 4.27)	–0.095	(1.23, 4.2)	-0.096
g11	(-0.68, 0.46)	0.75	(0.7, 0.5)	0.749
g24	(2.14, 3.84)	-5.98	(2.33,3.18)	-5.51

To my surprise, I have reached in my opinion sound results. In spite of them not being as good as implementation of stochastic ranking or NSGA-II in some of the preceeding sections and in g24 finding often solutions outside the boundaries of constraints, when it comes to g08 or g11, I believe the results are comparable to those found with other previously mentioned algorithms.

6 MULTI-OBJECTIVE APPROACH WITH MORE OBJECTIVES (1 PTS)

In an endeavour to see how a multi-objective approach differs from the one with a single objective under quasi-controlled conditions, I have decided to implement the NSGA-II algorithm to account for multiple objectives and to compare it tete-a-tete with my former implementation from the minimal requirement section. The results can be observed in the table below.

6.1 table describing NSGAII and NSGAII with multiple objectives

NSGA-II vs NSGA-II with mutliple objectives				
name	NSGAII values	NSGAII fitness	Multi version values	Multi version fitness
g06	(14.15, 0.96)	-6827.2	(14.97,5.74)	-2773.7
g08	(1.23, 4.24)	-0.096	(1.07, 3.95)	0.0042
g11	(-0.82, 0.66)	0.79	(-0.58, 0.32)	0.79
g24	(2.3, 3.06)	-5.36	(2.4,2.81)	-5.21

As can be viewed from the table, in spite of achieving results close to the ones in in the first section, the multi-ojective version seems to be worse than the original one, perhaps with the exception of g11.The source of this code can be viewed in the file 3criterionnsngaii.py.