Lab 3

Dr. Donald Davendra
CS471 - Optimization

October 26, 2016

1 Introduction

The lab introduces three of the most common meta-heuristics currently used. These are Genetic Algorithms (GA), Differential Evolution (DE) and Particle Swarm Optimization (PSO).

These algorithms appeared in the 1980's and 1990's and have been extensively researched for the past 20 years or so and form the backbone of many optimization toolkits used in industry.

This lab requires you to code the three algorithms in a common framework and conduct experiments on the unimodel and multimodel problems. The algorithms are described in the following sections.

2 Differential Evolution

Whether in industry or in research, users generally demand that a practical optimization technique should fulfil three requirements:

1. the method should find the true global minimum, regardless of the initial system parameter values;
2. convergence should be fast; and
3. the program should have a minimum of control parameters so that it will be easy to use.

Prof. Price invented the differential evolution (DE) algorithm in a search for a technique that would meet the above criteria. DE is a method, which is not only astonishingly simple, but also performs extremely well on a wide variety of test problems. It is inherently parallel because it is a population based approach and hence lends itself to computation via a network of computers or processors. The basic strategy employs the difference of two randomly selected parameter vectors as the source of random variations for a third parameter vector.

The parameters used in DE are $\Im$ = cost or the value of the objective function, D = problem dimension, NP = population size, P = population of X$-$vectors, G = generation

number, $Gmax$ = maximum generation number, X = vector composed of D parameters, V = trial vector composed of D parameters, CR = crossover factor. Others are F = scaling factor ($0 < \mathrm{F} \leq 1.2$), *(U)* = upper bound, *(L)* = lower bound, **u**, and **v** = trial vectors, $x_{best}^{(G)}$ = vector with minimum cost in generation G, $x_i^{(G)}$ = ith vector in generation G, $b_i^{(G)}$ = ith buffer vector in generation G, $x_{r1}^{(G)}$, $x_{r2}^{(G)}$ = randomly selected vector, L = random integer ($0 < \mathrm{L} < \mathrm{D} - 1$). In the formulation, N = number of cities. Some integers used are *i, j*.

Differential Evolution (DE) is a novel parallel direct search method, which utilizes NP parameter vectors

$$x_i^{(G)}, i = 0, 1, 2, ..., NP - 1 \tag{1}$$

as a population for each generation, G. The population size, NP does not change during the minimization process. The initial population is generated randomly assuming a uniform probability distribution for all random decisions if there is no initial intelligent information for the system. The crucial idea behind DE is a new scheme for generating trial parameter vectors. DE generates new parameter vectors by adding the weighted difference vector between two population members to a third member. If the resulting vector yields a lower objective function value than a predetermined population member, the newly generated vector replaces the vector with which it was compared. The comparison vector can, but need not be part of the generation process mentioned above. In addition the best parameter vector $x_{best}^{(G)}$, is evaluated for every generation G in order to keep track of the progress that is made during the minimization process. Extracting distance and direction information from the population to generate random deviations result in an adaptive scheme with excellent convergence properties.

Descriptions for the earlier two most promising variants of DE (later known as DE2 and DE3) are presented in order to clarify how the search technique works, then a complete list of the variants to date are given thereafter. The most comprehensive book that describes DE for continuous optimization problems is [**?**].

2.1 Scheme of DE

2.1.1 Initialization

As with all evolutionary optimization algorithms, DE works with a population of solutions, not with a single solution for the optimization problem. Population P of generation G contains NP solution vectors called individuals of the population and each vector represents potential solution for the optimization problem:

$$P^{(G)} = X_i^{(G)} \;\; i = 1, \ldots, NP; \;\; G = 1, \ldots, G_{\max} \tag{2}$$

Additionally, each vector contains D parameters:

$$X_i^{(G)} = x_{j,i}^{(G)} \;\; i = 1, \ldots, NP; \;\; j = 1, \ldots, D \tag{3}$$

In order to establish a starting point for optimum seeking, the population must be initialized. Often there is no more knowledge available about the location of a global optimum than

the boundaries of the problem variables. In this case, a natural way to initialize the population $P^{(0)}$ (initial population) is to seed it with random values within the given boundary constraints:

$$P^{(0)} = x_{j,i}^{(0)} = x_j^{(L)} + rand_j[0,1] \cdot \left(x_j^{(U)} - x_j^{(L)} \right) \quad \forall i \in [1, NP];\ \forall j \in [1, D] \tag{4}$$

where $rand_j[0,\ 1]$ represents a uniformly distributed random value that ranges from zero to one. The lower and upper boundary constraints are,$X^{(L)}$ and $X^{(L)}$, respectively:

$$x_j^{(L)} \leq x_j \leq x_j^{(U)} \ \ \forall j \in [1, D] \tag{5}$$

For this scheme and other schemes, three operators are crucial: mutation, crossover and selection. These are now briefly discussed.

2.1.2 Mutation

The first variant of DE works as follows: for each vector $x_i^{(G)}, i = 0, 1, 2, .., NP - 1$, a trial vector v is generated according to

$$v_{j,i}^{(G+1)} = x_{j,r1}^{(G)} + F \cdot \left(x_{j,r2}^{(G)} - x_{j,r3}^{(G)} \right) \tag{6}$$

where $i \in [1, NP]$; $j \in [1, D]$, $F > 0$, and the integers $r1,\ r2,\ r3 \in [1, NP]$ are generated randomly selected, except: $r1 \neq r2 \neq r3 \neq i$.

Three randomly chosen indexes, *r1*, *r2*, and *r3* refer to three randomly chosen vectors of population. They are mutually different from each other and also different from the running index *i*. New random values for *r1*, *r2*, and *r3* are assigned for each value of index i (for each vector). A new value for the random number $rand[0,\ 1]$ is assigned for each value of index j (for each vector parameter). F is a real and constant factor, which controls the amplification of the differential variation.

2.1.3 Crossover

In order to increase the diversity of the parameter vectors, the vector

$$u = (u_1, u_2, \cdot, u_D)^T \tag{7}$$

$$u_j^{(G)} = \begin{cases} v_j^{(G)} for\ j = \langle n \rangle_D, \langle n+1 \rangle_D, ..., \langle n+L-1 \rangle_D \\ \left(x_i^{(G)} \right)_j \quad otherwise \end{cases} \tag{8}$$

is formed where the acute brackets $\langle \rangle_D$ denote the modulo function with modulus D. This means that a certain sequence of the vector elements of u are identical to the elements of v, the other elements of u acquire the original values of $x_i^{(G)}$. Choosing a subgroup of parameters for mutation is similar to a process known as crossover in genetic algorithm. The integer L is drawn from the interval [0, D-1] with the probability $\Pr(L = v) = (CR)^v$. $CR \in [0, 1]$ is the crossover probability and constitutes a control variable. The random decisions for both n and L are made anew for each trial vector v.

Table 1: Differential Evolution Strategies

Strategy	Formulation
Strategy 1: DE/best/1/exp:	$v = x_{best}^{(G)} + F \cdot \left(x_{r2}^{(G)} - x_{r3}^{(G)}\right)$
Strategy 2: DE/rand/1/exp:	$v = x_{r1}^{(G)} + F \cdot \left(x_{r2}^{(G)} - x_{r3}^{(G)}\right)$
Strategy 3: DE/rand-to-best/1/exp	$v = x_{i}^{(G)} + \lambda \cdot \left(x_{best}^{(G)} - x_{i}^{(G)}\right) + F \cdot \left(x_{r1}^{(G)} - x_{r2}^{(G)}\right)$
Strategy 4: DE/best/2/exp:	$v = x_{best}^{(G)} + F \cdot \left(x_{r1}^{(G)} + x_{r2}^{(G)} - x_{r3}^{(G)} - x_{r4}^{(G)}\right)$
Strategy 5: DE/rand/2/exp:	$v = x_{r5}^{(G)} + F \cdot \left(x_{r1}^{(G)} + x_{r2}^{(G)} - x_{r3}^{(G)} - x_{r4}^{(G)}\right)$
Strategy 6: DE/best/1/bin:	$v = x_{best}^{(G)} + F \cdot \left(x_{r2}^{(G)} - x_{r3}^{(G)}\right)$
Strategy 7: DE/rand/1/bin:	$v = x_{r1}^{(G)} + F \cdot \left(x_{r2}^{(G)} - x_{r3}^{(G)}\right)$
Strategy 8: DE/rand-to-best/1/bin:	$v = x_{i}^{(G)} + \lambda \cdot \left(x_{best}^{(G)} - x_{i}^{(G)}\right) + F \cdot \left(x_{r1}^{(G)} - x_{r2}^{(G)}\right)$
Strategy 9: DE/best/2/bin	$v = x_{best}^{(G)} + F \cdot \left(x_{r1}^{(G)} + x_{r2}^{(G)} - x_{r3}^{(G)} - x_{r4}^{(G)}\right)$
Strategy 10: DE/rand/2/bin:	$v = x_{r5}^{(G)} + F \cdot \left(x_{r1}^{(G)} + x_{r2}^{(G)} - x_{r3}^{(G)} - x_{r4}^{(G)}\right)$

2.1.4 Selection

In order to decide whether the new vector u shall become a population member of generation *G+1*, it will be compared to $x_i^{(G)}$. If vector u yields a smaller objective function value than $x_i^{(G)}$, $x_i^{(G+1)}$ is set to u, otherwise the old value $x_i^{(G)}$ is retained.

2.2 DE Strategies

The originators have suggested ten different working strategies of DE and some guidelines in applying these strategies for any given problem (see Table1). Different strategies can be adopted in the DE algorithm depending upon the type of problem for which it is applied. The strategies can vary based on the vector to be perturbed, number of difference vectors considered for perturbation, and finally the type of crossover used.

The general convention used above is as follows: DE/x/y/z. DE stands for differential evolution algorithm, x represents a string denoting the vector to be perturbed, y is the number of difference vectors considered for perturbation of x, and z is the type of crossover being used. Other notations are exp: exponential; bin: binomial). Thus, the working algorithm is the seventh strategy of DE, that is, DE/rand/1/bin. Hence the perturbation can be either in the best vector of the previous generation or in any randomly chosen vector. Similarly for perturbation, either single or two vector differences can be used. For perturbation with a single vector difference, out of the three distinct randomly chosen vectors, the weighted vector differential of any two vectors is added to the third one. Similarly for perturbation with two vector differences, five distinct vectors other than the target vector are chosen randomly from the current population. Out of these, the weighted vector difference of each pair of any four vectors is added to the fifth one for perturbation.

In exponential crossover, the crossover is performed on the D (the dimension or number of variables to be optimized) variables in one loop until it is within the CR bound. For discrete optimization problems, the first time a randomly picked number between *0* and *1* goes beyond the CR value, no crossover is performed and the remaining D variables are left intact. In binomial crossover, the crossover is performed on each the D variables whenever a randomly picked number between *0* and *1* is within the CR value. Hence, the exponential and binomial crossovers yield similar results.

The outline for the DE algorithm is given in Figure 1.

$$
\begin{array}{l}
\text{1.Input :} D, G_{\max}, NP \geq 4, F \in (0, 1+), CR \in [0, 1], \text{and initial bounds :} \vec{x}^{(L)}, \vec{x}^{(U)}. \\
\text{2.Initialize :} \left\{ \begin{array}{l} \forall i \leq NP \wedge \forall j \leq D : x_{i,j}^{G=0} = x_j^{(L)} + rand_j\,[0,1] \cdot \left(x_j^{(U)} - x_j^{(L)} \right) \\ i = \{1, 2, \ldots, NP\}, j = \{1, 2, \ldots, D\}, G = 0, rand_j[0,1] \in [0,1] \end{array} \right. \\
\left\{ \begin{array}{l} \text{3.While } G < G_{\max} \\ \quad \forall i \leq NP \left\{ \begin{array}{l} \text{4. Mutate and recombine :} \\ \quad \text{4.1 } r_1, r_2, r_3 \in \{1, 2, ..., NP\}, \text{ randomly selected, except : } r_1 \neq r_2 \neq r_3 \neq i \\ \quad \text{4.2 } j_{rand} \in \{1, 2, ..., D\}, \text{ randomly selected once each } i \\ \quad \text{4.3 } \forall j \leq D, u_{j,i}^{(G+1)} = \left\{ \begin{array}{l} x_{j,r_3}^{(G)} + F \cdot (x_{j,r_1}^{(G)} - x_{j,r_2}^{(G)}) \\ \qquad \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i}^{(G)} \quad \text{otherwise} \end{array} \right. \\ \text{5. Select} \\ \vec{x}_i^{(G+1)} = \left\{ \begin{array}{l} \vec{u}_i^{(G+1)} \text{ if } f(\vec{u}_i^{(G+1)}) \leq f(\vec{x}_i^{(G)}) \\ \vec{x}_i^{(G)} \quad \text{otherwise} \end{array} \right. \end{array} \right. \\ G = G + 1 \end{array} \right.
\end{array}
$$

Figure 1: Differential Evolution Algorithm

3 Genetic Algorithm

Genetic Algorithms (GA) are the heuristic search and optimization techniques that mimic the process of natural evolution. Using this process, it aims to build a computational model that uses natural selections as the optimizing model.

There are three basic GA operators that are used in the model. The basic design of a *simple* GA is give as:

```
1 Function SimpleGeneticAlgorithm() /* simple Genetic Algorithm          */
2 |  Initialize population
3 |  Calculate fitness function
4 |  while !Terination_Criteria do
5 |  |  Selection
6 |  |  Crossover
7 |  |  Mutation
8 |  |  Calculate fitness function
9 |  end while
```

3.1 Selection

The process that determines which solutions are to be preserved and allowed to reproduce and which ones deserve to die out.

The primary objective of the selection operator is to emphasize the good solutions and eliminate the bad solutions in a population while keeping the population size constant.

There are different techniques to implement selection in Genetic Algorithms. They are:

- Tournament selection
- Roulette wheel selection
- Proportionate selection
- Rank selection
- Steady state selection

3.1.1 Tournament Selection

In tournament selection, several tournaments are played among a few individuals. The individuals are chosen at random from the population. The winner of each tournament is selected for next generation. Selection pressure can be adjusted by changing the tournament size. Weak individuals have a smaller chance to be selected if tournament size is large.

3.1.2 Roulette wheel and proportionate selection

In this selection, parents are selected based on the fitness values. Better solutions have a higher chance of getting accepted.

If f_i is the fitness of individual i in the population, its probability of being selected is $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, where N is the number of individuals in the population.

3.2 Crossover

The crossover operator is used to create new solutions from the existing solutions available in the mating pool after applying selection operator.

This operator exchanges the gene information between the solutions in the mating pool.

The most common is the 1-point crossover. Iterating through the population, another individual is randomly selected from the population, a random cut-off point is generated. The pairing is done pairwise, therefore from two individuals, we obtain two new child individuals. If either one of this child solutions improve on the current indexed parent, then it replaces the parent in the population. There are more selection criterias.

3.3 Mutation

Mutation is the occasional introduction of new features in to the solution strings of the population pool to maintain diversity in the population. Though crossover has the main responsibility to search for the optimal solution, mutation is also used for this purpose.

Mutation is obtained randomly, using some proportional selection criteria, a new variable is introduced in the solution.

3.4 Elitism

Crossover and mutation may destroy the best solution of the population pool by accident. Elitism is the preservation of few best solutions of the population pool. Elitism is defined in percentage or in a number of solutions that need to be preserved.

3.5 Genetic Algorithm Outline

The GA can be described as in the following algorithms.

input : NS: Number of solutions
DIM: Problem dimension
$Bounds$: Problem bounds (U - upper bound, L - lower bound)
t_{max}: Maximum number of iterations
CR: Crossover rate
$M.Rate$: Mutation rate
$M.Range$: Mutation value range
$M.Precision$: Mutation precision
ER: Elitism rate
output: x^*: best solution found

```
Elitism = ER · NS
Population = ∅
randomInit(Population,Bounds)        /* initialize solutions randomly     */
evaluate(Population)                 /* calculate fitness                 */
t = 1                                /* initialize t to 1                 */
while t ≤ t_max do
    NewPopulation = ∅
    for s = 1,...,NS; s += 2 do
        /* select parents by roulette wheel selection                     */
        P_1, P_2 = select(Population)
        /* perform crossover with probability CR                          */
        O_1, O_2 = crossover(P_1, P_2, CR)
        /* perform mutation with mutation parameters M                    */
        mutate(O_1, M, Bounds)
        mutate(O_2, M, Bounds)
        add(NewPopulation, O_1)
        add(NewPopulation, O_2)
    end for
    /* calculate cost for each solution                                   */
    evaluate(NewPopulation)
    /* combine Population and NewPopulation into Population               */
    reduce(Population, NewPopulation, EliteSN)
    /* calculate normalized fitness value                                 */
    getFitness(Population)
    getBestSolution(Population)
end while
```

Algorithm 1: Genetic Algorithm

```
1 Function reduce(Population, NewPopulation, EliteSN) /* sort solutions in
  each population by cost, so that minimal cost solutions come first    */
2 |  sortByCostAscending(Population)
3 |  sortByCostAscending(NewPopulation)
4 |  /* replace first EliteSN worst solutions in NewPopulation by best
  |     solutions in Population                                          */
5 |  for s = 1,...,EliteSN do
6 |  |  NewPopulation[SN+1-s]=Population[s]
7 |  end for
8 |  /* swap population data :  set Population to NewPopulation         */
9 |  swapData(Population, NewPopulation)
```

Algorithm 2: `reduce` function

```
1 Function select(Population)
2 |  P_1 = selectParent(Population)
3 |  P_2 = selectParent(Population)
4 |  return(P_1,P_2)
```

Algorithm 3: `select` function

```
1 Function selectParent(Population)
2 |  r = rand(1,Population.totalFitness)
3 |  s = 1
5 |  while s ≤ NS and r > 0 do
6 |  |  r -= Population.fitness[s]
7 |  end while
8 |  return(s)
```

Algorithm 4: `selectParent` function

```
1 Function getFitness(Population)
2 |  for s = 1,...,SN do
3 |  |  if Population.cost[s] >= 0 then
4 |  |      Population.fitness[s] = 1/(1+ Population.cost[s]);
5 |  |  else Population.fitness[s] = 1 + abs(Population.cost[s]);
6 |  end for
7 |  Population.totalFitness = sum(Population.fitness)
```

Algorithm 5: `getFitness` function

```
1 Function mutate(S,M,Bounds)
2 |  for i = 1,...,DIM do
3 |  |  if random(0,1) < M.Rate then
4 |  |  |  S[i] += random(-1,1) ·(Bounds[i].U − Bounds[i].L) · M.Range ·
  |  |  |  power(2, (−1 · random(0, 1) · M.Precision))
5 |  |  end if
6 |  end for
```

Algorithm 6: `mutate` function

```
1 Function crossover(P1, P2, CR)
2 |   if random(0,1) < CR then
3 |       d = random(1,DIM)
4 |       O1 = join(P1[1..(d-1)], P2[d..DIM]) ;
5 |       O2 = join(P2[1..(d-1)], P1[d..DIM]) ;
6 |   else
7 |       O1 = P1 ;
8 |       O2 = P2 ;
```

Algorithm 7: `crossover` function

4 Particle Swarm Optimization

Inspired by the flocking and schooling patterns of birds and fish, Particle Swarm Optimization (PSO) was invented by Russell Eberhart and James Kennedy in 1995. Originally, these two started out developing computer software simulations of birds flocking around food sources, then later realized how well their algorithms worked on optimization problems.

Particle Swarm Optimization might sound complicated, but it's really a very simple algorithm. Over a number of iterations, a group of variables have their values adjusted closer to the member whose value is closest to the target at any given moment. It's an algorithm that's simple and easy to implement.

The algorithm keeps track of three global variables:

- Target value or condition
- Global best (gBest) value indicating which particle's data is currently closest to the Target
- Stopping value indicating when the algorithm should stop if the Target isn't found

Each particle consists of:

- Data representing a possible solution
- A Velocity value indicating how much the Data can be changed
- A personal best (pBest) value indicating the closest the particle's Data has ever come to the Target

The particles' data could be anything. In the flocking birds example above, the data would be the X, Y, Z coordinates of each bird. The individual coordinates of each bird would try to move closer to the coordinates of the bird which is closer to the food's coordinates ($gBest$). If the data is a pattern or sequence, then individual pieces of the data would be manipulated until the pattern matches the target pattern.

The velocity value is calculated according to how far an individual's data is from the target. The further it is, the larger the velocity value. In the birds example, the individuals furthest from the food would make an effort to keep up with the others by flying faster toward the gBest bird. If the data is a pattern or sequence, the velocity would describe how

different the pattern is from the target, and thus, how much it needs to be changed to match the target.

Each particle's $pBest$ value only indicates the closest the data has ever come to the target since the algorithm started.

The $gBest$ value only changes when any particle's $pBest$ value comes closer to the target than $gBest$. Through each iteration of the algorithm, $gBest$ gradually moves closer and closer to the target until one of the particles reaches the target.

It's also common to see PSO algorithms using population topologies, or "neighborhoods", which can be smaller, localized subsets of the global best value. These neighborhoods can involve two or more particles which are predetermined to act together, or subsets of the search space that particles happen into during testing. The use of neighborhoods often help the algorithm to avoid getting stuck in local minima.

The general outline of the PSO algorithm is given in the following pseudocode.

input : *Iterations*: maximum number of iterations
Particles: number of particles p_i
gBest: the best solution in the population
pBest: the best solution found by specific particle
Bounds: Problem bounds (U - upper bound, L - lower bound)
output: *gBest*: best particle found

for i = *1,...,Particles* **do**
    /* generate particles randomly */
    p_i = L+rand[0,1](U-L)
    /* calculate particles velocity */
    $v_i = \mathtt{f}(p_i)$
    /* set *pBest* for each particle */
    $pBest_i = v_i$
end for
/* set gBest from all particles */
$gBest = \mathtt{min}(pBest)$
for t = *1,...,Iterations* **do**
    for j = *1,...,Particles* **do**
        /* calculate new velocity v_j of particle p_j */
        ${v_j}^{(t+1)} = {v_j}^{(t)} + c_1 \cdot rand \cdot \left({pBest^{(t)}}_j - {p_j}^{(t)}\right) + c_2 \cdot rand \cdot \left(gBest^{(t)} - {p_j}^{(t)}\right)$
        /* update particle p_j */
        ${p_j}^{(t+1)} = {p_j}^{(t)} + {v_j}^{(t+1)}$
        /* calculate particles velocity */
        $v_j = \mathtt{f}(p_j)$
        /* check if the particle velocity has improved */
        if $v_j < pBest_j$ **then**
            /* update *pBest* of particle */
            $pBest_j = v_j$
        end if
        /* check if *gBest* has improved */
        if $pBest_j < gBest$ **then**
            /* update *gBest* */
            $gBest = pBest_j$
        end if
    end for
end for

Algorithm 8: Particle Swarm Optimization

5 Experimentation

The student is required to code all three algorithms in the language of their choice. Ideally, all three algorithms should share same auxiliary structures, such as population generation, memory management, problems definitions etc.

The experimentation parameters is given in Table 2.

Table 2: Experiment parameters

Parameters	Values
Population size	50 (min)
Iterations	100 (min)
Dimensions	20

Submission

The student must submit the following separate files to canvas:

1. source codes for the problems
2. a LaTeX typeset report on the results and its analysis

The report must contain an introduction in the algorithms, the full experimentation results in tabular format and condensed results with statistical analysis compared with what was obtained in Lab 1 and 2.

The files must be submitted through Canvas by midnight November 7, 2016. The penalty for late submission is 10% for 1 day, 20% for 2 day, after which it will be zero. The grading rubric is given in Table 3.

Table 3: Grading rubric

File	Aspects	Points
`Code`	Compiles and executes	35
	Explanation	15
`Report`	Results	25
	Analysis	25