Combine genetic algorithms (GA) with Abstract Machine having multiple computation cycles

CS3650 Course Project
Mahbaneh Eshaghzadeh Torbati

1. Abstract

This study is a try to map an evolutionary algorithm, Genetic Algorithm (GA), to the Abstarct Machine model. In this study, we explain the GA methodology and propose an abstract machine model for this algorithm. As deed1, we propose a GA based design for the TDR system. We simulate data for an optimization problem, design and implement GA for the problem and plot the results. As deed2, I compare TDR system implemented with GA to Chi system implemented by Ant Colony algorithm.

2. Combining GA with Abstract Machine Model

Abstract machine model is a general model for designing the learning cycles in slow intelligence systems. Evolutionary algorithms, such as Genetic Algorithm (GA), Particle swarm optimization (PSO), and Ant colony, inherently follow an iterative and slow process learning process. In this study, we want to show that how we can map the GA algorithm to abstract machine model and give the formal cycle definition for GA, called GAcycle. In this section, first we explain the GA algorithm, then we propose the GAcycle.

Genetic Algorithm

GA is a probabilistic optimization algorithm, inspired by the biological evolution process. It uses the concepts of "Natural Selection" and "Genetic Inheritance" proposed by Darwin. This algorithm is particularly well suited for hard problems where little is known about the underlying search space. The data presentation, evaluation function, and the iterative steps in this algorithm is inspired from the evolutionary process exists in nature. The process in which each generation of species tries to adapt itself to the environment and the individuals with stronger personalities and characteristics survive and dominant the population. Here is the scenario we follow in the GA algorithm: We have an initial population of individuals. These individuals are the problem's solutions, selected randomly at the beginning of the algorithm. Then, in an iterative process individuals are selected as the parents and bear children. These children are new solutions who inherit their parents' characteristics. By selecting a new generation from the combination of parents and children, we try to select the stronger new generation which can be considered as better set of solutions.

Genetic Algorithm Design

The GA design is dependent on the problem we have. However, the principal for designing the GA is still the same for any problem. For designing a GA algorithm, we should design three main components:

Representation:

Each individual is considered as a set of genotype or chromosomes that can transfer these characteristics to its children. We can consider each of these chromosomes as a feature and represent each individual as a vector of features. We have different kinds of representations, such as binary string, string of integer or double values. The type of representation depends on the problem we are solving.

Fitness function:

The design of this function also depends on the problem and it is used for evaluating the sample domain. The fitness of every individual in the population is evaluated, the fitness is usually the value of the objective function in the optimization problem being solved.

Stochastic operators:
We define four different kinds of stochastic operators. These operators construct the GAcycle.
Selection:
Selection allocates more copies of the solutions with higher fitness values. Thus, we can impose the survival-of-the-fittest mechanism for the solutions. By this operator want to create the next generation from the parents with higher fitness function. In this way, we hope to have a new generation with higher fitness values. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection, and random selection.

Recombination:
Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring or children). The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner. There are many recombination methods, such as k-point Crossover and Uniform Crossover.

k-point Crossover: One-point, and two-point crossovers are the simplest and most widely applied crossover methods. In one-point crossover, illustrated in Figure 4.1, a crossover site is selected at random over the string length, and the alleles on one side of the site are exchanged between the individuals. In two-point crossover, two crossover sites are randomly selected. The alleles between the two sites are exchanged between the two randomly paired individuals.

Uniform crossover: In uniform crossover, illustrated in Figure 1, every allele is exchanged between a pair of randomly selected chromosomes with a certain probability, pe, known as the swapping probability. Usually the swapping probability value is taken to be 0.5.

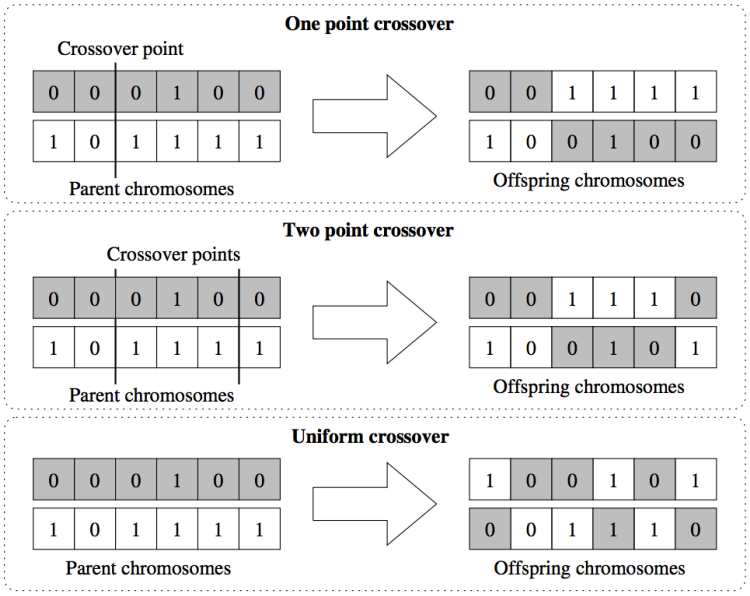

Figure 1. different methods of crossover.

Mutation:
While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's characteristics. In other words, mutation performs a random walk in the vicinity of a candidate solution.

Replacement:
In replacement, we want to select the next better generation. It means that we select the new generation from both parent and offspring populations. Many replacement techniques such as elitist replacement, generation-wise replacement and steady-state replacement methods are used in GAs.

GAcycle:
Based on our GA explanation, we can map GA to Abstract Machine model. Here is our formal definition:

$$\textbf{Cycle1 [guard1,1]: } P^1 0 \textbf{ –enum1< = } P^1 1 \textbf{ –enum2< = } P^1 2 \textbf{>conc1= } P^1 3$$

enum1: Selection and Recombination in GA.
enum2: Mutation in GA.
conc1: Replacement in GA.

3. Implementation

We decided to apply the GAcycle to the TDR system. TDR is an experimental multi-level slow intelligence system for personal health care. The TDR system can be used by a single user or a group of users who will interact to understand, maintain and improve each other's state of health. The TDR system mainly consists of three super-components: Tian, Di and Ren. According to the Chinese philosophy these three super-components are the essential ingredients of a human-centric psycho-physical system. They can be thought of as human beings (Ren) interacting with the environment consisting of heaven (Tian) and earth (Di). For personal health care, there is a fourth higher level super-component called Chi (or Qi), which in this context represents the state of health of a person (or persons).

Our idea in here is to simulate the Chi high-level super-component by applying GAcycle to three lower-level super-components, Tian, Di, and Ren. In this regard, we simulated data for an optimization problem, design a GA algorithm for our optimization problem, implement the GA algorithm, and apply it to the TDR system. As you can see in Figure2, each GAcycle for the Chi component consists of three GAcycle for the lower-level super-components. In this way, we decrease the dimension of the data we are working on during applying our optimization algorithm, GA.

Simulated problem

As the simulated problem, we define an optimization problem: We want to find those individuals who has the best health level (Chi value). As the health level evaluation method, we selected a sphere function, Figure 2 and Equation (1). What we want to reach is the minimum value for this function which is the best level of health for individuals.

$$f(X) = \sum_{i=1}^{n} x_i \qquad \textit{Equation (1)}$$

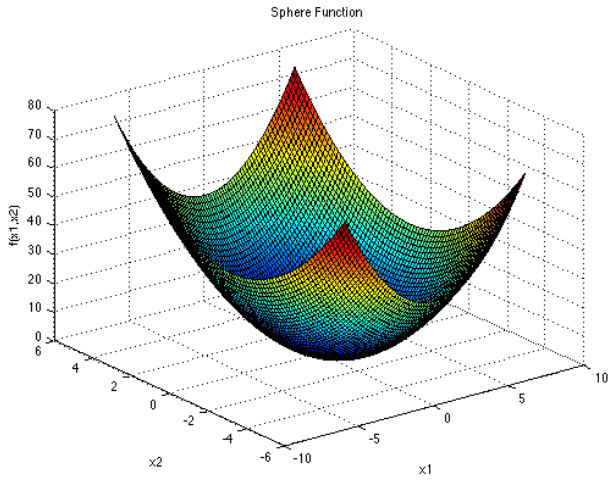

Figure 2. Sphere function.

GA design:

- Recombination: Integer string of features.
- Fitness function: Patient health level which simulated by sphere function.
  The goal is reaching value zero.
  Sphere function.
- Initialization: 50 individual of length 30.
- Parent selection: Randomly assign parents.
- Recombination: One-point crossover.
- Mutation: Replace worst individual with a new individual.
- Survivor selection: Best individuals from Parents and offspring.

Experiment

The following two figures, Figure 3 and Figure 4, show how the population evolves during running the GA algorithm. Figure 5 shows the applying of Ant colony to the same population used for GA algorithm. Ant colony is another evolutionary algorithm designed based on the trend ants uses to find their food. While the GA algorithm is used for the TDR system, the Ant colony is used directly for the Chi system. In other word, no feature splitting has been done for the Ant colony algorithm. In Figure 6, you can see that GA outperforms Ant colony algorithm for the same optimization problem.

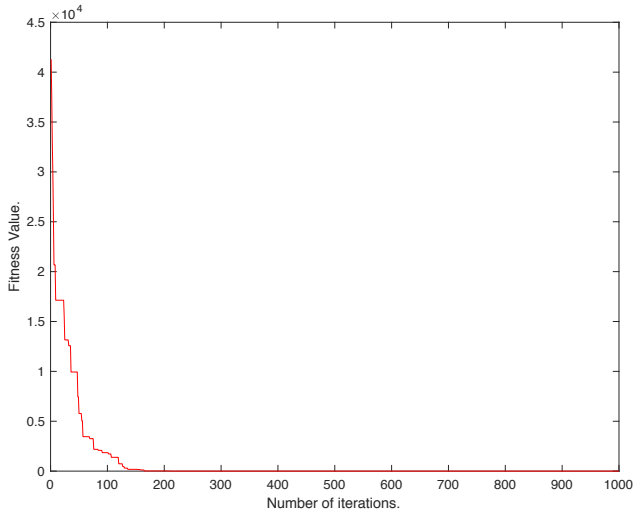

Figure 3. GA on simulated data for 800 iterations.

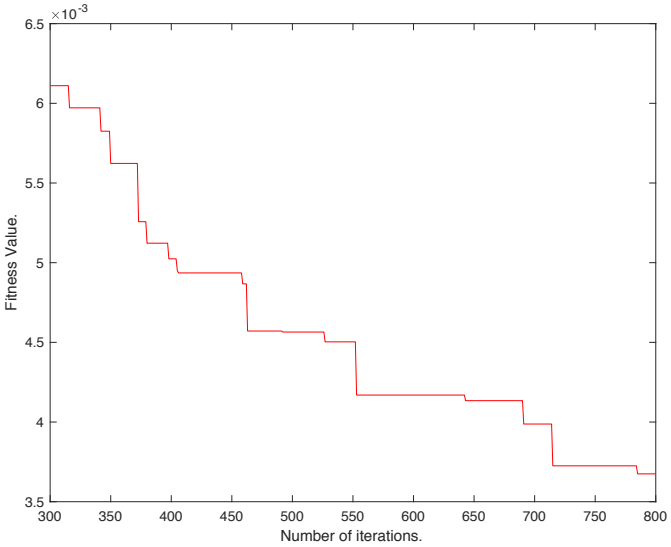

Figure 4. GA on simulated data for iterations [300-800]

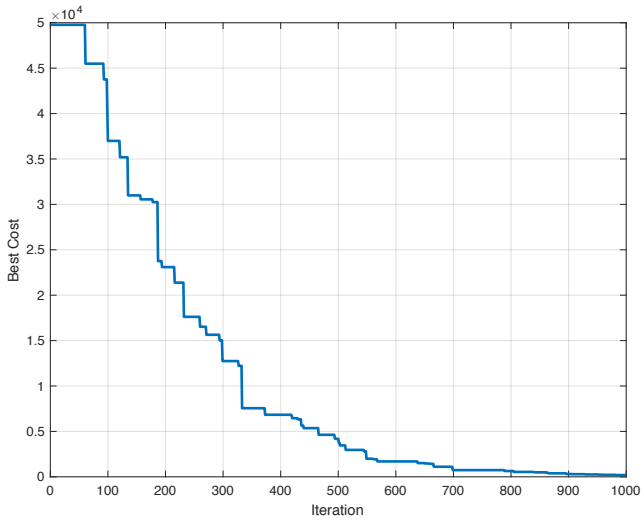

Figure 5. Ant colony on simulated data for Chi

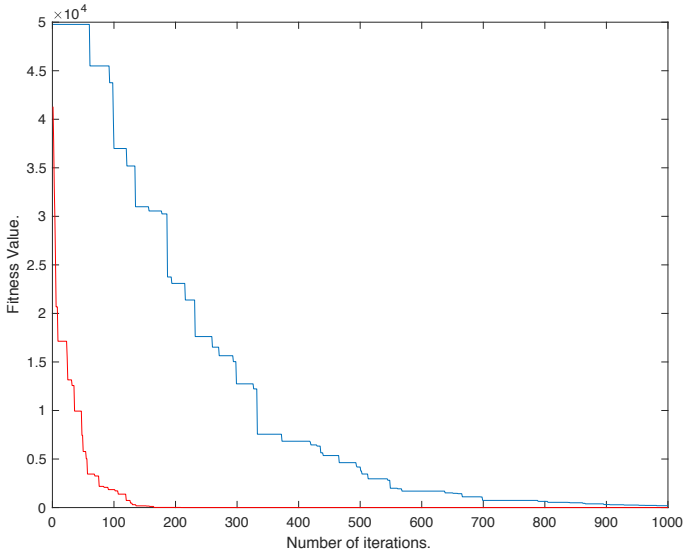

Figure 6. Comparison of GA for TDR (in red) to Ant colony for Chi (in blue).

4. References

[1] Sastry, K., Goldberg, D.E. and Kendall, G., 2014. Genetic algorithms, Chapter 4. In *Search methodologies* (pp. 93-117). Springer US.

[2] Reeves, C., 2003. Genetic algorithms, Chapter 3. In *Handbook of metaheuristics* (pp. 55-82). Springer US.

[3] Tang, Y., Zhang, H., Liang, Z. and Chang, S.K., Social Network Models for the TDR System.

5. Appendix

In this section, we added the code for TDR system.

```
#!/usr/bin/python3.4
# (c) Mohammad H. Mofrad, 2016
# (e) hasanzadeh@cs.pitt.edu

import numpy as np
from math import *
import os
import time
import xml.etree.ElementTree as et
import random
from random import randint

# Parabola benchmark function (y = x^2)
def parabola(x):
  size, dim = x.shape
  y = np.array([np.sum([np.power(a,2) for a in x[i,:]]) for i in range(size)]).reshape(size,1)
  return(y)

# Context vector
def context_vector(best, swarm, x):
  best[:,swarm] = x
  return(best)

# Action selection
def actionselection(action, probability, numactions, numdims):
  for i in range(numdims):
    a = np.random.choice(np.arange(0, numactions), p = probability[:,i])
    mask = np.zeros(numactions,dtype=bool)
    mask[a] = True
    action[mask,i] = 1
    action[~mask,i] = 0
  return(action)

# Update probabilities
def probabilityupdate(action, probability, numactions, numdims, signal, alpha, beta):
  for i in range(numdims):
    a = np.where(action[:,i] == 1)
    mask = np.zeros(numactions,dtype=bool)
    mask[a] = True
    if not signal:
      probability[mask,i] = probability[mask,i] + alpha * (1 - probability[mask,i])
      probability[~mask,i] = (1 - alpha) * probability[~mask,i]
    else:
      probability[mask,i] = (1 - beta) * probability[mask,i]
      probability[~mask,i] = (beta/(numactions-1)) + (1-beta) * probability[~mask,i]
  return(probability)

XML = 'init.xml'
actionset = []
if os.path.isfile(XML):
  print ('*****************************************************************')
```

```
else:
  print ('**********************************************************************')
  print('Initializing using local configs')
  # Maximum iterations
  imax = 1000
  # Acceleration coefficients
  c1   = 1.49445
  c2   = 1.49445
  # Weight
  wmax = 0.9
  wmin = 0.4

  # Number of dimensions
  dim  = 30

  # Population size
  size = 50

  # Enumeration size
  efactor = 2/3

  # TDR constraints
  tdrfactor = 3
  # Action set

  # MutationRate
  MutationRate = 15
  CrossoverMethod = "OnePointCrossOver"
  MutationMethod = "FlipAllCells"
  SelectionMethod = "SelectMaxFitness"
  Algo = 'GA'

  actionset.append('pso')

  actionset.append('ga')

# Elite size
esize = size - round(size * efactor)

# Max and min position bounds
xmax =  100
xmin = -xmax

# Max and min velocity bounds
vmax = 0.2 * (xmax - xmin)
vmin = -vmax

# TDR constraints
k1 = dim % tdrfactor
k1len = ceil(dim/tdrfactor)
k2 = tdrfactor - k1
k2len = floor(dim/tdrfactor)
tdrtable = np.zeros((1, dim))
index = np.zeros((1, dim))
if(k1):
```

```
    for i in range(k1):
      for j in range(k1len):
        tdrtable[0,(i*k1len)+j] = i

    for i in range(k2):
      for j in range(k2len):
        tdrtable[0,(k1*k1len)+(i*k2len)+j] = k1+i
else:
    for i in range(k2):
      for j in range(k2len):
        tdrtable[0,(i*k2len)+j] = i

# Position
x    = np.zeros((size, dim))
x    = xmin + ((xmax - xmin) * np.random.rand(size, dim))

# Fitness
fx   = np.zeros((size,1))
fx   = parabola(x)

# Personal best position
pb   = np.zeros((size, dim))
fpb  = np.zeros((size, 1))
pb   = np.copy(x)
fpb  = np.copy(fx)

# Global best position
gb   = np.zeros((1, dim))
fgb  = 0
gb   = np.copy(x[np.argmin(fx), :].reshape(1,dim))
fgb  = np.copy(fx[np.argmin(fx)]).reshape(1,1)

print ('**********************************************************************')
print ('Initial Health Level: ', fgb[0,0])

alpha = 0.1
beta = 0.1
numactions = len(actionset)
action = np.zeros((numactions,tdrfactor))
probability = np.tile(1/numactions, (numactions,tdrfactor))

numactions_d = tdrfactor
action_d = np.zeros((numactions_d, dim))
probability_d = np.tile(1/numactions_d, (numactions_d,dim))


FinalPopulation = np.zeros((size, dim)) #added from GA
# Main loop for updating particles
for i in range(imax):


    action_d = actionselection(action_d, probability_d, numactions_d, dim)
    action = actionselection(action, probability, numactions, tdrfactor)
    w = wmax - (((wmax - wmin) / imax) * i) # Small step size

    #added from GA
```

```
for h in range(size):
  FinalPopulation[h, :] = np.copy (x[0, :])
print ('Chi supercomponent Iteration: (',i , ')')
# The indices that would sort the x, v, fv, pb
# 5 Concentration on the elite population
# TDR split swarm
for j in range(tdrfactor):
  signal = 1

  a_d = np.arange(dim)
  swarm = a_d[action_d[j,:] == 1]
  swarmsize = len(swarm)
  if swarmsize:
    swarm = swarm.reshape(1,swarmsize)
    a = np.arange(numactions)
    c = a[action[:,j] == 1]

    if  Algo == 'GA':


   #New copies of population
      modifiedPopulation = np.zeros((size, dim))#added from GA
      for h in range(size):#added from GA
        modifiedPopulation[h, :] = np.copy (gb[0, :])
      for k in range(size):
        modifiedPopulation[k, swarm[0, :]] = np.copy (x[k, swarm[0, :]])

   #-----------apply crossover to population----------------
   #SelectParentsForFirstGeneration()
      data = [[i] for i in range(size)]#added from GA
      random.shuffle(data)#added from GA
      h = 0
      children = np.zeros((size, dim))#added from GA

      while (h < size):

      #find the split point
        SplitIndexInSwarm = randint(0, swarmsize - 1)
        SplitPoint = swarm [0, SplitIndexInSwarm];


      #Split both parents
        SplittedParent1 = np.split(modifiedPopulation[data[h][0], :], [SplitPoint])
        SplittedParent2 = np.split(modifiedPopulation[data[h+1][0], :], [SplitPoint])

      #Bear child1
        child1 = np.concatenate((SplittedParent1[0], SplittedParent2[1]))
      #Bear child2
        child2 = np.concatenate((SplittedParent2[0], SplittedParent1[1]))

      #Add them to new generation
        children[h,:] = child1
        children[h + 1, :] = child2
        h = h + 2
   #---------------------- Compute Fitness and select best individual--------------------
      Fitness = parabola(children)
```

```
            minFitness = Fitness.min()
            MinIndex = np.where(Fitness == minFitness)
            SelectedMinVAlue = children[MinIndex[0][0], :]

        #------------------------------- Add mutation-----------------------------
        # First Mutation
            MutationRate = randint(1, 100)
            if (MutationRate < 15):
              maxFitness = Fitness.max()
              MaxIndex = np.where(Fitness == maxFitness)

         # Rebuild the sample
              newSample = np.copy( np.reciprocal(children[MaxIndex[0][0], :]))
              children[MaxIndex[0][0]] = newSample
              Fitness[MaxIndex[0][0], :] = np.sum((newSample)**2)

        #--------- Select next population: children are always next generation. (constructed from gbest)
            FinalPopulation[:, swarm] = np.copy (children[:, swarm])
            if (minFitness < fgb):
          #Update gb and fgb
              gb = np.copy([SelectedMinVAlue])
              fgb = np.copy(minFitness).reshape(1,1)
              signal = 0

        CycleStep ='';

        if (j==0):
            CycleStep = 'Tian'
        elif (j==1):
            CycleStep = 'Den '
        else:
            CycleStep = 'Ren '

        changingstatus = 'Failure'
        if (signal == 0):
            changingstatus = 'success'
        if (c):
            print ('    ',CycleStep , 'Component: ', 'Selected cycle is ', Algo, ' Health level is: ', fgb[0,0] , '(',
changingstatus, ')')
        else :
            # a = fgb
             print ('    ',CycleStep , 'Component: ', 'Selected cycle is ', Algo, 'Health level is: ', fgb[0,0],
'(',changingstatus , ')')
    x = np. copy (FinalPopulation)
    print ()
    time.sleep(.1)
    probability_d = probabilityupdate(action_d, probability_d, numactions_d, dim, signal, alpha, beta)
    probability = probabilityupdate(action, probability, numactions, tdrfactor, signal, alpha, beta)
    #print('Iteration', i, 'Global best', fgb)
print (probability_d)
```