Aalto University

Department of Computer Science

Genetic Algorithms
Final Project

A Genetic Algorithm for Project Assignment Problem

Team: Lilit Janjughazyan, Elen Vardanyan

Summer, 2017

Abstract

Genetic algorithms (GAs) have recently become a powerful tool for solving optimization problems. This paper illustrates a real-world application of genetic algorithms for the project assignment problem. The assignment problem is a complex problem in which every single student should be assigned a single project. The goal is to find a set of assignments such that the overall desirability of the match is maximized while the cost of the match is minimized. This project is an implementation of a GA described in [PRH05].

Keywords: Genetic algorithms; Project assignment; NP-hard problems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Setting and Description

Over the last few years biological evolution has become an appealing source of inspiration for tackling combinatorial optimization problems. The assignment problem is one of the fundamental combinatorial optimization problems. There are numerous methods for solving the assignment problem from general simplex algorithm to more specific ones. However, the widespread use of genetic algorithms in many computational problems has made it possible to achieve tangible results and has shown a satisfying performance.

The aim of this Project is to develop a genetic algorithm for solving the project assignment problem.

1.2 History and Background of Genetic Algorithms

Genetic Algorithms (GA) are adaptive techniques for optimization and search that find solutions to problems by an evolutionary process inspired in the mechanisms of natural selection and genetics. But why use evolution and nature-inspired techniques as an incentive for solving computational problems? To the researchers of the field, the mechanisms of evolution seem very well suited for some of the most challenging computational problems in various fields.

In fact, evolution is an approach of searching among an huge set of possible genetic sequences for desired "solutions" that are highly fit organisms able to survive and reproduce in their environments. Since the organisms constantly evolve, their fitness criteria changes continually. Therefore, the nature is constantly in search for a set of new possibilities. Keeping track of changing conditions and searching for solutions in the face of those conditions is what is ideally required for adaptive

computer programs. Additionally, evolution can be considered as a parallel search method, since rather than working on one species at a time, it works and changes millions of species simultaneously. Finally, the "rules" of evolution are remarkably simple: species evolve by means of random variation (i.e mutation, recombination, etc.), followed by natural selection in which the fittest tend to survive and reproduce, thus propagating their genetic material to future generations. Yet these simple rules are thought to be responsible, in large part, for the extraordinary variety and complexity we see in the biosphere.

Thus, applications of genetic algorithms are a growing trend that have already proven to be very useful. In this paper, the mathematical formulation of the project assignment problem is going to be explained, as well as the development of our algorithm to solve the problem. In addition, the overall performance of our algorithm is going to be discussed.

1.3 The Structure of the Project Paper

The next chapter introduces the biological terminology that will be used throughout the paper, explains the idea behind the project assignment problem and gives the mathematical description of it. Moreover, it gives the theoretical background of the genetic algorithms, and serves as an introduction to their use in tackling the assignment problem. The genetic algorithm structure and its implementation are provided in Section 3. Lastly, Section 4 discusses conclusions and the further work that can be done.

Chapter 2

Theoretical Background

2.1 Biological Terminology

All living organisms consist of cells. Each cell contains the same set of chromosome(s) (strings of DNA) that serve as a model for the organism. A chromosome consists of blocks of DNA, called genes, which are located at a particular position (locus) on the chromosome and are responsible for forming a particular protein. In other words, each gene encrypts a specific character, say hair color. Alleles are responsible for coding various forms of the character (i.e. for hair color - brown, red, etc.).

The first step of reproduction is recombination, or so called crossover. During this process the genes of parents are exchanged and a whole new chromosome is formed in some way. Because of copying errors, the single nucleotides, elementary bits of DNA, can vary from parent to offspring. This variation is called a mutation.

The fitness of an organism is measured by the probability that it will be able to survive and and reproduce.

In genetic algorithms, chromosomes are typically referred to the set of possible solutions to the given problem, which are often encrypted as a bit string. The single bits or short blocks of adjacent bits encoding a specific element of the possible solution are referred to as genes. Most of the time alleles are either 0 or 1. Crossover is about exchanging the "genetic material" between the parents. Mutation is, usually, just flipping a single bit at some locus. [Mit96]

2.2 The Assignment Problem

The assignment problem arises in a number of contexts, from assigning taxis to customers to assigning projects to students. In the latter case, students are asked

Student:	Project: 1	2	3	4	5	...	P
1		1			3		2
2	1		3				
3				1	2		
4	2		4	1			
:							
S		1			5		3

Figure 2.1: A student-project matrix with S students and P projects.

to indicate their preferences in a scoring system form student-project matrix (see figure 2.1), expressing their choices in consecutive manner (i.e. 1 indicates a first choice, 2 indicates a second choice, and so on) until a predefined maximum number of preferences is reached.

The increase of the number of students and projects leads to the increase of conflicts on project choices. To illustrate the possible conflicts clearly, let's assume that two students have chosen the same project as their first choice. Since every student should get exactly one project, therefore one of the two students might be assigned his/her first preference and the other one - the second preference. But what if the second preference of that student coincides with the first preference of a third student? So, again it should be decided either to deny a student's first choice or to further demote him/her to a lower preference. And the same problem reoccurs on and on. So finding a suitable project for each of the students becomes a tough job.

Sometimes it's possible to find an optimal solution to the project allocation problem using the classical integer programming approach to the generalized assignment problem (GAP) given limited number of constraints, however genetic algorithms have shown better performance and more valuable results. One clear advantage of the usage of genetic algorithms to solve this problem is that a project assignment can be accomplished given ***any*** student-project matrix. This makes ground for multi-objective decision making without worrying about explicitly defining the objectives in the model formulation. This is an extremely useful property when tackling problems of this format. [Wil97]

2.2.1 Mathematical formulation of the problem

Let $S = \{1, 2, ..., s\}$ be a set of students, and let $P = \{1, 2, ..., p\}$ be a set of projects $(s \leq p)$. For $i \in S$, $j \in P$, we define c_{ij} which indicates the preference of student $\boldsymbol{i}$ to being assigned the project $\boldsymbol{j}$.

The student-project matrix $(s \times p)$ contains students' preferences of 1 for the first, 2 for the second choices up until the maximum value of p. In case, if c_{ij} has not been assigned an integer value, meaning that the student $\boldsymbol{i}$ has not included the project $\boldsymbol{j}$ in his/her wishlist, then c_{ij} is assigned a suitably large penalty value B. Moreover, we assign priority weights w_i to every student, so that some of them have a better possibility to be allocated the project of higher preference.

So, now we are ready to describe the problem mathematically:

Minimize:

$$\sum_{i=1}^{s} \sum_{j=1}^{p} w_i c_{ij} x_{ij}$$

Subject to:

$$\sum_{j \in P} x_{ij} = 1, \qquad \forall \boldsymbol{i} \in S,$$

$$\sum_{i \in S} x_{ij} \leq 1, \qquad \forall \boldsymbol{j} \in P,$$

$$x_{ij} \in 0, 1 \qquad \forall \boldsymbol{i} \in S, \forall \boldsymbol{j} \in P$$

where

$$x_{ij} = \begin{cases} 1, & \text{if project } \boldsymbol{j} \text{ is assigned to student } \boldsymbol{i} \\ 0, & \text{otherwise} \end{cases}$$

and

$$\forall \boldsymbol{i} \in S, \boldsymbol{j} \in P, \quad c_{ij} = B \quad if \quad c_{ij} \notin \{1, 2, ..., p\}.$$

The first constrain guarantees that each of the students gets exactly one project, whereas the second constrain ensures that a single project can be assigned at most to one person. The lower the values of student preferences the better match will be formed, and priority weights are in ascending order (a higher ranking indicates greater priority), so it's time to minimize the objective function.

Chapter 3

The Algorithm

3.1 Genetic Algorithm Structure

As mentioned earlier GAs are inspired by Darwin's theory of evolution and are powerful search mechanisms that attempt to mimic the processes of natural selection and evolution. GAs work with a population of strings and as in nature, its goal is to see the survival of the fittest within a structured but stochastic information exchange framework. After completing every cycle new strings are created combining the parts of the fittest members of the last iteration and sometimes also incorporating new parts as a result of mutation.

The following are the basic steps of the developed GA:

1. Generate a family of initial chromosomes (solutions)
2. Calculate the fitness of each chromosome
3. Choose two parents from the set of initial chromosomes
4. Produce an offspring
5. Perform mutation
6. Calculate the fitness of the offspring
7. Replace the least fit family member of the initial set of chromosomes with the child produced, if the child is fitter than that family member and is not yet in the family
8. If the number of predefined iterations has been reached, then stop, if not, go to the Step 3

3.2 The Algorithm Implementation

Firstly, as GA-s are heavily based on random number generation, as well as vector/matrix operations, let's import the packages that we will use throughout the code.

```
import numpy as np
import random
```

Listing 3.1: Importing necessary packages

We will also need the global variables **SP_matrix** and **priority_weights**.

```
SP_matrix           # GLOBAL, an sxp matrix of preferences
priority_weights    # GLOBAL, an s-dimensional list
```

Listing 3.2: User-defined global variables

1. **The initial family of chromosomes**: We firstly examine the dataset and remove assignments that can be preset (e.g., a students has indicated a first choice for a project that no other student has included in their preference list). For the remaining N students, we generate an initial set of N chromosomes. Each gene in the chromosomes is defined by randomly assigning a project to a student from their wishlist. Let r_{ig} be the project assigned to student $i (i \in S)$ in chromosome $g (g = 1, ..., N)$.

```
def generate_chromosomes(SP_matrix):

    """
    param SP_matrix: the student-project matrix (a two-dimensional numpy
    array)
    return: a list of lists, each of which is a chromosome
    """

    rows = [] # students
    cols = [] # projects

    # remove assignments that can be preset
    for p in range(SP_matrix.shape[1]):
        if 1 in np.unique(SP_matrix[:,p]):
            rows.append(list(SP_matrix[:, p]).index(1))
            cols.append(p)
    np.delete(SP_matrix, rows, 0)
    np.delete(SP_matrix, cols, 1)

    N = SP_matrix.shape[0]
    chromosomes = [random.sample(range(SP_matrix.shape[1]), SP_matrix.
    shape[0]) for chromosome in range(N)]

    return chromosomes
```

Listing 3.3: Python function for generating the initial family of chromosomes

2. **The fitness function**: For finding the fitness f_g of chromosome g, we use the following function:

$$f_g = \sum_{i \in S} h(C_{ir_{ig}}) W_i$$

The choice of $h(\cdot)$ is up to us. If a suitable choice of $h(\cdot)$ is made, the GA will perform better and converge faster to the feasible solutions.

As in the mathematical formulation of the problem, a project g outside a student's preference list has a suitably large penalty value of B assigned to the $C_{ir_{ig}}$ coefficient. Moreover, if a project g has been assigned to more than one student, then a large value M is assigned to the $C_{ir_{ig}}$ coefficient. Since we look for solutions with lower fitness scores, the algorithm will tend towards solutions in which the third constraint is satisfied, therefore attempting to assign all students to one of their preferred projects ($\forall i \in S, j \in P; c_{ij} \in 1, 2, ..., p$).

```
def fitness_function(chromosome, B=100, M=10, h=lambda x: x**2):

    """
    param chromosome: a list of genes/projects, i-th element of which is the project allocated to the i-th student
    param B: a penalty, a suitably large value assigned to the C_ir_ig coefficient if a project g is outside of the student's preferences
    param M: a penalty, a suitably large value assigned to the C_ir_ig coefficient if a project g is assigned to more than one student
    param h: a lambda function, preferably a non-linear function on the student's preferences
    return fitness: the fitness of the chromosome
    """

    c = np.zeros(len(chromosome))
    for i in range(len(chromosome)):
        if chromosome[i] not in SP_matrix[i, :]:
            c[i] = B
        for j in range(len(chromosome)):
            if chromosome[i] == chromosome[j] & i!=j:
                c[i] = c[j] = M
        else:
            c[i] = SP_matrix[i, chromosome[i]]

    fitness = sum([h(c[i])*priority_weights[i] for i in range(len(chromosome))])

    return fitness
```

Listing 3.4: Python function for computing the fitness function

3. **Parent selection using a binary tournament**: Applying the method of the binary tournament selection, two chromosomes are chosen randomly from the set

of chromosomes. The chromosome with the lowest fitness value (i.e. the fittest chromosome) is kept. This fittest chromosome is then selected to be a parent. The process is repeated to find another parent. The two parents are then combined to produce an offspring.

```
def binary_tournament(chromosomes):

    """
    param chromosomes: a list of lists, each of which is a chromosome
    return parent1, parent2: lists, the parents chosen to reproduce the
    child in the next step
    """

    parent1, parent2 = random.sample(chromosomes, 2)
    if fitness_function(parent1) > fitness_function(parent2):
        parent1 = parent2

    parent3, parent4 = random.sample(chromosomes, 2)
    if fitness_function(parent3) < fitness_function(parent4):
        parent2 = parent3
    else:
        parent2 = parent4

    return parent1, parent2
```

Listing 3.5: Python function for the Binary Tournament

4. **Producing an offspring**: After the crossover of the two selected parents an offspring is constructed. GA crossover operators are implemented by a process of randomly generating one or more crossover points in the chromosome and then exchanging subsets of genes from the parents to form a child chromosome. We use a generalized fitness-based crossover operator, the fusion operator. This algorithm produces a single child from the two parents by trying to build on the fitness of the parents. To prevent unfeasible solutions, we slightly modify the fusion operator: Let f_u and f_v denote the fitness of two parents U and V, respectively. Let C be the child chromosome to be produced. Finally, let the subscript i denote the ith gene in the chromosome (i = 1, ..., s).

If both U_i and V_i are different from $C_k, k = 1, ..., i1$, then

1. If $U_i = V_i$, then $C_i = U_i = V_i$

2. If $U_i \neq V_i$, then

   a) $C_i = U_i$ with probability $f_v = (f_u + f_v)$

   b) $C_i = V_i$ with probability $f_u = (f_u + f_v)$

Else, if both U_i and V_i are already genes in $C_k, k = 1, ..., i-1$, choose at random a project not in $C_k, k = 1, ..., i-1$, and assign it to gene C_i. Otherwise, make Ci equal to the gene U_i or V_i not already in $C_k, k = 1, ..., i-1,$. This is done for each gene to be generated in the new chromosome, C. This process passes parental fitness from the fitter parent with a higher probability, while keeping the solutions (chromosomes) feasible.

```
def crossover(u, v):

    """
    param u, v: lists , parents that have to form a child
    return c: a list , the child produced from the crossover
    """

    f_u = fitness_function(u)
    f_v = fitness_function(v)
    c = np.zeros(len(u), dtype=int)
    if (f_u == 0) & (f_v == 0):
        prob_u = prob_v = 0.5
    else:
        prob_u = f_v/(f_u+f_v)
        prob_v = f_u/(f_u+f_v)


    c[0] = np.random.choice([u[0],v[0]], p=[prob_u, prob_v])

    for i in range(1,len(u)):
        if (u[i] not in c[:i]) & (v[i] not in c[:i]):
            if u[i] == v[i]:
                c[i] = u[i] = v[i]
            else:
                c[i] = np.random.choice([u[i],v[i]], p=[prob_u, prob_v])
        elif (u[i] in c[:i]) & (v[i] in c[:i]):
            c[i] = np.random.choice(list(set(range(SP_matrix.shape[1]))-
   set(c[:i])))
        elif (u[i] in c[:i]) & (v[i] not in c[:i]):
                c[i]=v[i]
        elif (v[i] in c[:i]) & (u[i] not in c[:i]):
                c[i]=u[i]

    return c
\vspace{2mm}
```

Listing 3.6: Python function for performing the crossover

5. **Mutation**: Mutation allows the possibility of introducing new genes in to the population, therefore providing the algorithm with a randomizing element to work alongside the crossover operators. Without mutation, the GA could easily converge to a local minimum due to lack of genetic diversity. For each of the s genes within the child chromosome C, the probability of changing the value of the ith gene, by choosing a different allele, is given by the mutation rate. This rate is

set by the user. Again, to keep the chromosome applicable, the new allele is chosen among the projects that are not already in C.

```
def mutate(chromosome, prob=None):

    """
    param chromosome: a list of genes/projects, i-th element of which is the project allocated to the i-th student
    param prob: a float between (0,1), the probability of mutation, default is None (makes the probability )
    """

    for i in range(len(chromosome)):
        p = random.uniform(0, 1)
        if p < (prob == None) * 1/(SP_matrix.shape[0] * np.sqrt(len(chromosome))) + (prob != None) * prob:
            if SP_matrix.shape[0] == SP_matrix.shape[1]: # in case when the num of students == the num of projects, do the mutation by swapping
                chromosome[i], chromosome[len(chromosome)-i] = chromosome[len(chromosome)-i], chromosome[i]
            else: # otherwise, choose the new gene among the projects not already in the chromosome
                chromosome[i] = np.random.choice(list(set(range(SP_matrix.shape[1])) - set(chromosome)))
```

Listing 3.7: Python function for Mutation

5. **Replacement**: The offspring now replaces an existing member of the set of chromosomes. This type of replacement makes the set of chromosomes become fitter whilst preserving the genetic diversity. Moreover, this process generates a number of different solutions to choose from. If the child is added to the set, then it immediately becomes eligible to be a parent and to be replaced.

```
def replace(child, chromosomes):

    """
    param child: a list, the newest child which may replace the least fit chromosome
    param chromosomes: a list of lists, each of which is a chromosome
    """

    least_fit_index = 0
    least_fitness = fitness_function(chromosomes[0])
    for i in range(1, len(chromosomes)):
        if least_fitness < fitness_function(chromosomes[i]):
            least_fitness = fitness_function(chromosomes[i])
            least_fit = chromosomes[i]
            least_fit_index = i
    chromosomes[least_fit_index] = child # checked if the child is already in chromosomes or not in GA(num_of_cycles)
```

Listing 3.8: Python function for replacing the least fit chromosome with the new child

6. **GA**: The whole process described above is repeated until the program performs the user defined maximum number of cycles.

```
def GA(num_of_cycles):
    chromosomes = generate_chromosomes(SP_matrix)
    pop_size = len(chromosomes)
    for i in range(num_of_cycles):
        fitnesses = []
        for j in range(len(chromosomes)):
            fitnesses.append(fitness_function(chromosomes[j]))

        parent1, parent2 = binary_tournament(chromosomes)
        child = crossover(parent1, parent2)
        mutate(child, 0.01)
        child = list(child) # to avoid problems of comparisons with lists
        child_fitness = fitness_function(child)
        if (child not in chromosomes) & (child_fitness < min(fitnesses)):
            replace(child, chromosomes)
        fitnesses = [] # update fitnesses after possible replacement
        for j in range(len(chromosomes)):
            fitnesses.append(fitness_function(chromosomes[j]))

        print("Fittest in the {}-th cycle:   \t\t\t".format(i),
    chromosomes[fitnesses.index(min(fitnesses))], "Fitness: ", min(
    fitnesses))
        print("Least Fit in the {}-th cycle: \t\t\t".format(i),
    chromosomes[fitnesses.index(max(fitnesses))], "Fitness: ", max(
    fitnesses), "\n")
    print("Family of chromosomes after {} cycles \n".format(num_of_cycles
    ))
    for chromosome in chromosomes:
        print("{}) ".format(chromosomes.index(chromosome)+1), *chromosome
    , " with fitness ", fitness_function(chromosome))
```

Listing 3.9: The main function GA

Chapter 4

Conclusion and Further Work

4.1 Conclusion

In this project we have implemented the genetic algorithm presented in [PRH05] as an aid for project assignment. The developed GA applies reproduction, crossover and mutation operators to chromosomes to create new individuals. These three operators together act as a very powerful search mechanism.

As described in the paper, we have incorporated the possibility of using various penalty weights defined on the student preferences, as well as changing the default nested scoring function $h(x) = x^2$ in the fitness function.

An advantage of GA approach to assignment problems of this type is that it provides the user with a number of different assignments to facilitate the multi-objective decision-making process.

4.2 Further Work That Can Be Done

A logical continuation of this project would be a comparison of the developed GA to an optimal integer programming approach, both for small and large complex assignments. Lastly, a user-friendly program can be developed to incorporate all inputs and parameters to produce a population of very fit solutions more easily.

Bibliography

[Mit96] M. Mitchell. *An introduction to Genetic Algorithms*. 1996.

[Wil97] J. M. Wilson. "A genetic algorithm for the generalised assignment problem". In: *The Journal of the Operational Research Society* (1997).

[PRH05] Israel T. Vieira Arjan K. Shahani Paul R. Harper Valter de Senna. "A genetic algorithm for the project assignment problem". In: *Computers Operations Research* (2005).