

Czestochowa University of Technology

**Evolutionary Algorithms  
and  
Search Strategies.**

**Project for the problems 3, 4, 5**

**Students worked on the project:**

- 1. Rodier Sangibala**
- 2. Sindayigaya Laurent**
- 3. Fazil Raheem**

## 1. Problems description

- Problem 1: Knapsack problem

- a) General description of the problem

The knapsack problem is popular in the research field of constrained and combinatorial optimization with the aim of selecting items into the knapsack to attain maximum profit while simultaneously not exceeding the knapsack's capacity. We explain how a simple genetic algorithm (SGA) can be utilized to solve the knapsack problem and outline the similarities to the feature selection problem that frequently occurs in the context of the construction of an analytical model.

- b) The goal of the problem

Our goal is to find the best solution, so we have to identify a subset that meets the constraint and has the maximum total benefit.

Let  $X_i$  determines how many copies of item  $i$  are to be placed into the knapsack. The goal is to:

Maximize

$$\sum_{i=1}^N B_i X_i$$

Subject to the constraints

$$\sum_{i=1}^N$$

$$V_i X_i \leq V$$

$$i = 1 \text{ And } 0 \leq X_i \leq Q_i$$

Exemple

Item	A	B	C
Benefit	4	3	5
volume	6	7	8

We seek to maximize the total benefit:

$$\sum_{i=1}^3$$

$$B_i X_i = 4X_1 + 3X_2 + 5X_3$$

Subject to the constraints:

$$\sum_{i=1}^3$$

$$V_i X_i = 6X_1 + 7X_2 + 8X_3 \leq 13$$

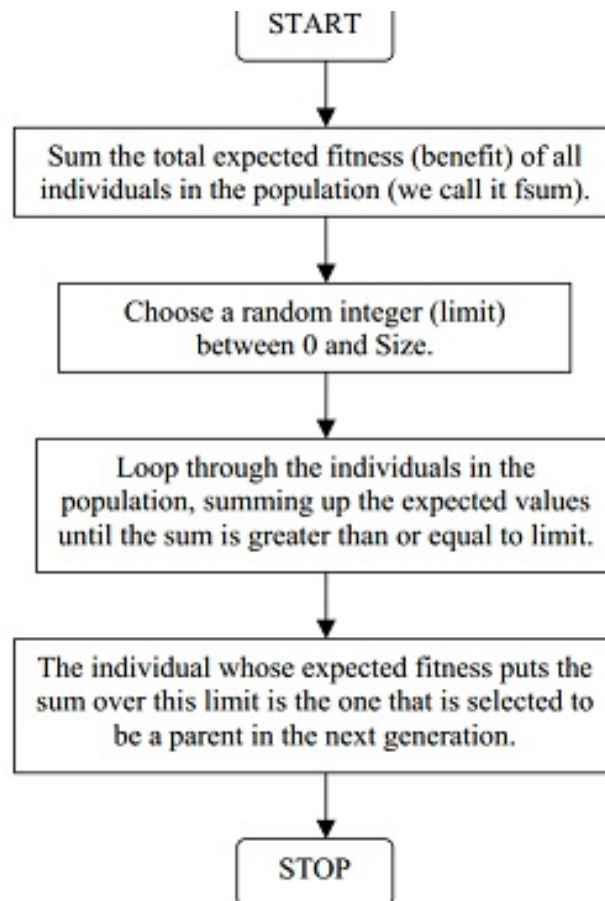
And

$$X_i \in \{0,1\}, \text{ for } i=1, 2, \dots, n$$

- c) Fitness function description, what is important in the function, description of all variables.

GAs require a fitness function which allocates a score to each chromosome in the current population.

Thus, it can calculate how well the solutions are coded and how well they solve the problem. We calculate the fitness of each chromosome by summing up the benefits of the items that are included in the knapsack, while making sure that the capacity of the knapsack is not exceeded. If the volume of the chromosome is greater than the capacity of the knapsack then one of the bits in the chromosome whose value is '1' is inverted and the chromosome is checked again.



d) Constrains for the search space (testing for feasible solutions)

The program has to eliminate any conflicts of items that are already been taken into account. If there are many items with same values the program has to run it many times for the best solution. There must not be any repetition of a taken item. The population converges when either 90% of the chromosomes in the population have the same fitness value or the number of generations is greater than a fixed number.

e) . Encoding solutions of the problem for the used evolutionary algorithm

(chromosome description, vector/matrix description)

We use a data structure, called cell, with two fields (benefit and volume) to represent every item.

Then we use an array of type cell to store all items in it,

A chromosome can be represented in an array having size equal to the number of the items (in our example of size 4). Each element from this array denotes whether an item is included in the knapsack.

To represent the whole population of chromosomes we use a tri-dimensional array (chromosomes [Size][number of items][2]). Size stands for the number of chromosomes in a population. The second dimension represents the number of items that may potentially be included in the knapsack. The third dimension is used to form the new generation of chromosomes.

## 2. Short description of the used algorithm/algorithms

### Crossover

We tried both single and double point crossover. Since there was not a big difference in the results we got from both methods, we decided to use single point crossover. The crossover point is determined randomly by generating a random number between 0 and `num_items - 1`. We perform crossover with a certain probability. If crossover probability is 100% then whole new generation is made by crossover. If it is 0% then whole new generation is made by exact copies of chromosomes from old population. We decided upon crossover rate of 85% by testing the program with different values. This means that 85% of the new generation will be formed with crossover and 15% will be copied to the new generation.

### Mutation

Mutation is made to prevent GAs from falling into a local extreme. We perform mutation on each bit position of the chromosome with 0.1 % probability.

Complexity of the program Since the number of chromosomes in each generation (Size) and the number of generations are fixed, the complexity of the program depends only on the number of items that may potentially be placed in the knapsack. We will use the following abbreviations, N for the number of items, S for the size of the population, and G for the number of possible generations. The function that initializes the array chromosomes has a complexity of  $O(N)$ . The fitness, crossover function, and mutation functions have also complexities of  $O(N)$ . The complexities of the two selection functions and the function that checks for the terminating condition do not depend on N (but on the size of the population) and they have constant times of running  $O(1)$ . The selection, crossover, and mutation operations are performed in a for loop, which runs S times. Since, S is a constant, the complexity of the whole loop is  $O(N)$ . Finally, all these genetic operations are performed in a do while loop, which runs at most G times. Since G is a constant, it will not affect the overall asymptotic complexity of the program. Thus, the total complexity of the program is  $O(N)$ .

```
Microsoft Visual Studio Debug Console

Solution when Knapsack Max Cost = 20

Name : Food      Cost : 4      Value : 5
Name : Game      Cost : 1      Value : 8
Name : diamond   Cost : 5      Value : 5
Name : cable     Cost : 1      Value : 3
Name : tablet    Cost : 6      Value : 1
Name : bag       Cost : 3      Value : 7

Total Cost = 20
Total Value = 29

-----
p_c=0.1, fit=31.43
p_c=0.2, fit=30.57
p_c=0.30000000000000004, fit=32.45
p_c=0.4, fit=30.11
p_c=0.5, fit=32.94
p_c=0.6, fit=32.33
p_c=0.7, fit=29.27
p_c=0.7999999999999999, fit=33.41
p_c=0.8999999999999999, fit=30.68
p_c=0.9999999999999999, fit=31.69

C:\Users\sangi\Desktop\Semester 1\Evolutionary Algorithms\KnapsackSubmission\KnapsackSubmission\bin\Release\netco
0\KnapsackSubmission.exe (process 11904) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close th
le when debugging stops.
Press any key to close this window . . .
```

```
Microsoft Visual Studio Debug Console

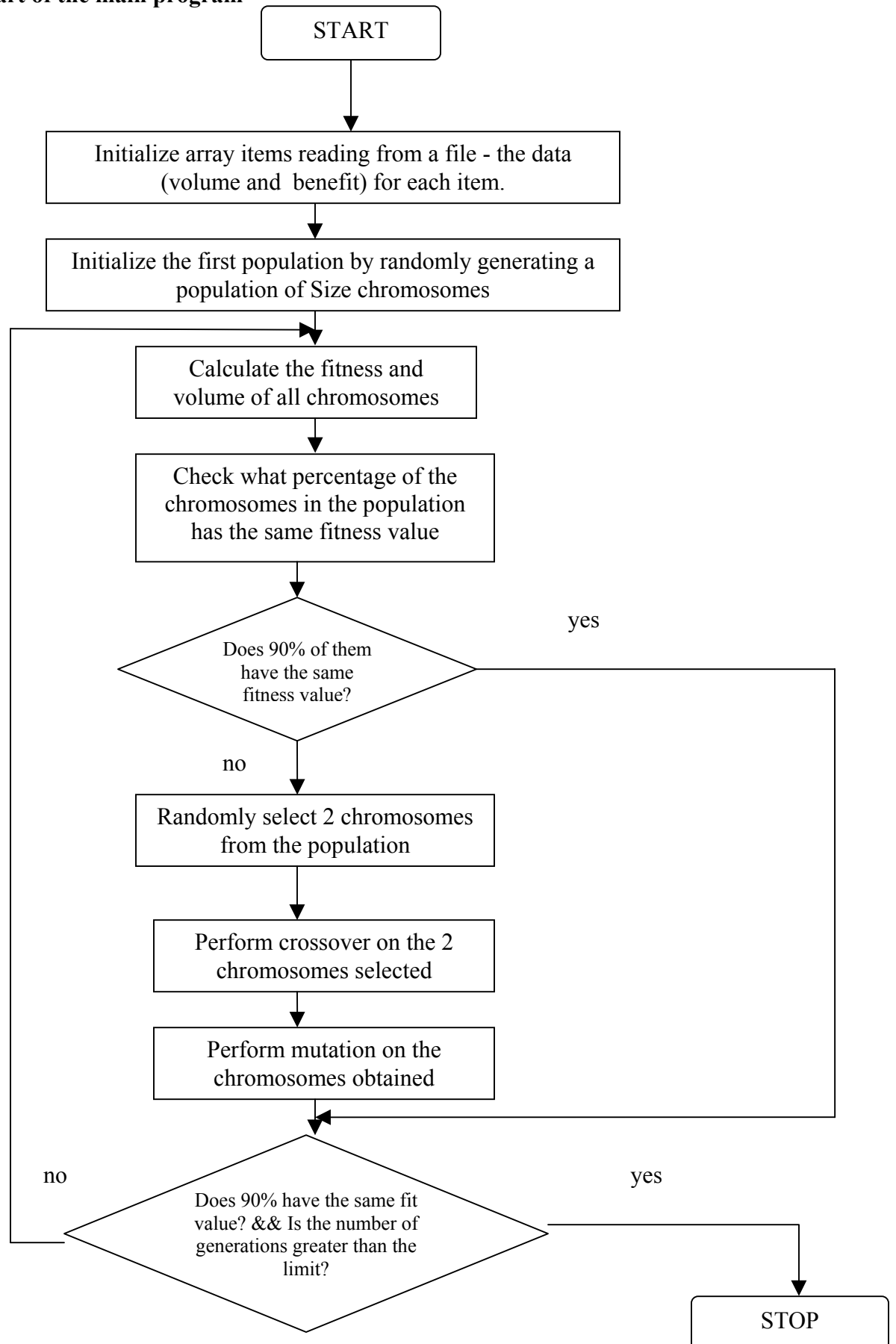
Name : painting Cost : 2      Value : 7
Name : Phone    Cost : 10     Value : 10
Name : printer  Cost : 3      Value : 9
Name : TV       Cost : 6      Value : 6
Name : juice    Cost : 3      Value : 9
Name : tablet   Cost : 6      Value : 1
Name : ketchup  Cost : 1      Value : 2
Name : rop      Cost : 1      Value : 12

Total Cost = 32
Total Value = 56

-----
p_c=0.1, fit=27.5
p_c=0.2, fit=26.91
p_c=0.30000000000000004, fit=27.17
p_c=0.4, fit=28.4
p_c=0.5, fit=27.2
p_c=0.6, fit=27.37
p_c=0.7, fit=27.13
p_c=0.7999999999999999, fit=26.17
p_c=0.8999999999999999, fit=24.77
p_c=0.9999999999999999, fit=26.07

C:\Users\sangi\Desktop\Semester 1\Evolutionary Algorithms\KnapsackSubmission\KnapsackSubmission\bin\Release\netco
0\KnapsackSubmission.exe (process 18468) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close th
le when debugging stops.
Press any key to close this window . . .
```

### Flowchart of the main program



The program is working fine and the final result will appear showing the best generation that give us the best fitness function.

### algorithm

input: A set of items with cost and values.

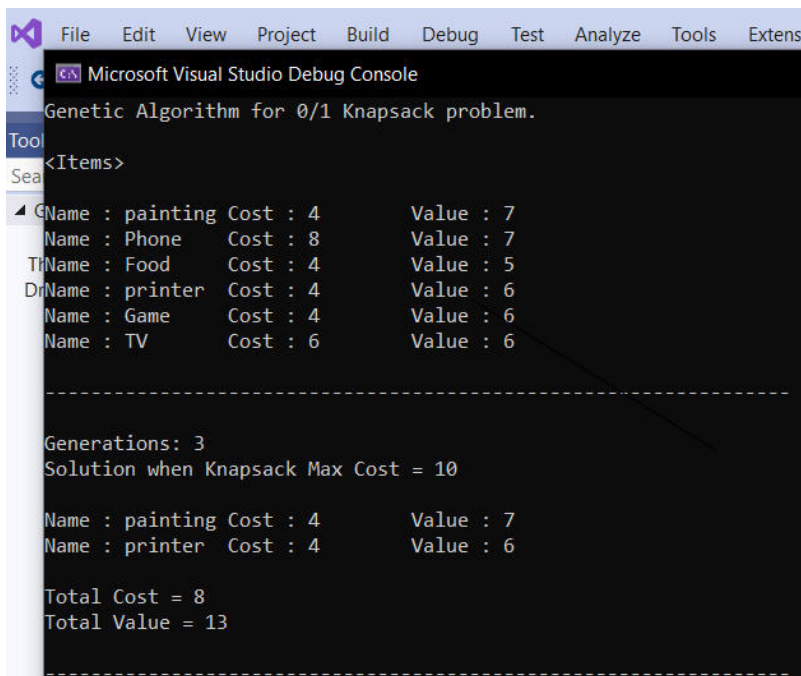
output: The greatest combined value of a subset.

partition the set  $\{1...n\}$  into two sets A and B of approximately equal size

compute the weights and values of all subsets of each set

for each subset of A do

find the subset of B of greatest value such that the combined cost is less than W



```

File Edit View Project Build Debug Test Analyze Tools Extensi
Microsoft Visual Studio Debug Console
Genetic Algorithm for 0/1 Knapsack problem.
<Items>
Name : painting Cost : 4 Value : 7
Name : Phone Cost : 8 Value : 7
Name : Food Cost : 4 Value : 5
Name : printer Cost : 4 Value : 6
Name : Game Cost : 4 Value : 6
Name : TV Cost : 6 Value : 6

-----

Generations: 3
Solution when Knapsack Max Cost = 10

Name : painting Cost : 4 Value : 7
Name : printer Cost : 4 Value : 6

Total Cost = 8
Total Value = 13
-----

```

### Conclusion

We have shown how Genetic Algorithms can be used to find good solutions for the Knapsack Problem. GAs reduce the complexity of the KP from exponential to linear, which makes it possible to find approximately optimal solutions. The results from the program show that the implementation of a good selection method and elitism are very important for the good performance of a genetic algorithm.

## 1.Problems description

### Job shop scheduling problem solved by an evolutionary algorithm

#### A) General description of the problem

The task of production scheduling consists in the temporal planning of the processing of a given set of orders. The processing of an order corresponds to the production of a particular product. It is accomplished by the execution of a set of operations in a predefined sequence on certain resources, subject to several constraints. The result of scheduling is a schedule showing the temporal assignment of operations of orders to the resources to be used. In this report, we consider a flexible job shop problem.

Each operation can be preformed by some machines with different processing times, so that the problem is known to be NP hard. The difficulty is to find a good assignment of an operation to a machine in order to obtain a schedule which minimizes the total elapsed time(make span) .

We have structured the scheduling problem in the following way:

- consider a set of  $N$  jobs  $\{J_j\}_{1 \leq j \leq N}$ ; these jobs are independent of one another;
- each job  $J_j$  has an operating sequence, called  $G_j$ ;
  - each operating sequence  $G_j$  is an ordered series of  $x_j$  operations,  $O_{i,j}$  indicating the position of the operation in the technological sequence of the job;
- the realization of each operation  $O_{i,j}$  requires a resource or a machine selected from a set of machines,  $\{M_k\}_{1 \leq k \leq M}$ ;  $M$  is the total number of machines existing in the shop, this implying the existence of an assignment problem;
- there is a predefined set of processing times; for a given machine, and a given operation, the processing time is denoted by  $P_{i,j,M_k}$ ;
- an operation which has started runs to completion (non-preemption condition);
- each machine can perform operations one after another (resource constraints);
- the time required to complete the whole job constitutes the makespan  $C_{max}$ .

#### B) The goal of the problem

Our objective is to determine the set of completion times for each operation  $\{C_{i,j,k}\}_{1 \leq i \leq X_j, 1 \leq j \leq N, 1 \leq k \leq M}$  which minimizes  $C_{max}$ .

Data Structures and Representation :

Genetic algorithms process populations of strings. We construct a population as an array of individuals where each individual contains the phenotype, genotype (artificial chromosome) and the fitness (objective function). In job-shop scheduling problem we noted the job order on



each machine as phenotype, the machine schedule as genotype and make span value as fitness.

A schedule can be represented by the set of permutations of jobs on each machine which are called job sequence matrix. Figure 1 shows the job sequence matrix for  $3 \times 3$  problem. Rows will represent the machines and columns represent the order of jobs.

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 1 & 3 \end{bmatrix}$$

Figure1: A *job sequence matrix* for  $3 \times 3$  problem.

This technique will allow the search process in a wide space. However, this kind of technique will take a lot of time to get the optimal value. In this paper, we chose to start the initial population with a randomly generated schedules, including some of the schedules obtain by the well known priority rules such as the shortest processing time and the longest processing time.

**Parents Selection** In parent selection we choose two individuals to reproduce. There are many ways of choosing parents to evaluate. In order to avoiding the premature convergence, we randomly choose two individuals from the population and called them parent1 and parent2. This means that all individuals in the population have the same chances to reproduce

#### **D) Fitness function description**

In our project the what is important in the function was the minimization of a makespan.

Here in our case of this project the fitness function can be expressed in two different manners with description of all variables.

we focus on the static and deterministic environment. In other words, the number of jobs and their processing time are known, fixed and we assumed that there are no machines breakdown occurred. We used

makespan of the operation as fitness value. Makespan is denoted as  $\max C$  is the time when the last operation leaves the workplace.  $(\ ) nCCCC \dots, \max 21\max =$  where,  $(\ ) \sum ++ = n k k j m j k j j p W r C 1$ .

$jC$  is the completion time of job  $j$ ,  $j r$  is the release time of job  $j$ ,  $j k W$  is the waiting time of job  $j$  at sequence  $k$  and  $(\ ) k j m p$  is the processing time needed by job  $j$  on machine  $m$  at sequence  $k$ .

### F).The way of repairing solution after crossover and mutation

Typically, the crossover is the main operator applied in genetic algorithms which is in contrast to evolution strategies. By crossover, the genetic structure of two selected individuals of the current population is mixed. The idea of this technique is to evaluate a point  $x$  by the distance  $(\ ) 2, p x d$ . Let's denote parent1 and parent2 as  $1 p$  and  $2 p$ .

First, set  $1 p x =$ . Then, we generated the CB neighbourhood for  $x$ ,  $(\ ) x N$ . For each member,  $i y$ , in  $(\ ) x N$  we calculated the distance between the members and  $2 p$  to produce  $(\ ) 2, p y D i$ . Then, we sort  $(\ ) 2, p y D i$  in ascending order. Starting from the first index in  $(\ ) 2, p y D i$  sort, we accepted  $i y$  with probability one if the fitness value is less than the current fitness value  $(\ ) ( \ ) x V y V i \leq$ . Otherwise, we accepted it with probability 0.5. Starting from  $1 p$ , we modified  $x$  step by step approaching  $2 p$ . After some iteration, we will find that  $x$  will gradually lose  $1 p$ 's characteristics and started to inherit  $2 p$ 's characteristics although in a different ratios. We choose the child depending on the less DG distance between the child and both its parents.

Genetic algorithms solve a problem using the principal of evolution. In the search process it will generate a new solution using genetic operator such as selection, crossover and mutation. In hill-climbing, the search procedure will stop once it detects no improvement in next iteration. This criterion make the hill-climbing technique tend to stop at local optima. In the other hand, genetic algorithms start its search space in a population and will maintain the number of population in iteration. It will generate a new schedule by selecting two individuals in population to apply crossover and mutation

## G). Encoding solutions of the problem for the used evolutionary algorithm (chromosome description, vector/matrix description)

---

### Algorithm 1: Crossover

1. Let  $p_1$  and  $p_2$  be the parent solution.
  2. Set  $x = p_1 = q$ .
  3. Find CB Neighbourhood for  $x$ ,  $N(x)$ .
  4. Do
    - a. For each member  $y_i \in N(x)$ , calculate the distance between  $y_i$  and  $p_2$ ,  $D(y_i, p_2)$ .
    - b. Sort the distance value in ascending order,  $D_{sort}(y_i, p_2)$ .
    - c. Starting from  $i = 1$ , do
      - i. Calculate the fitness value for  $y_i$ ,  $V(y_i)$ .
      - ii. If  $V(y_i) \leq V(x)$  accept  $y_i$  with probability one, and with probability 0.5 otherwise.
      - iii. If  $y_i$  is not accepted, increase  $i$  by one.
    - Repeat i-iii until  $y_i$  is accepted.
    - d. Set  $x = y_i$ .
    - e. If  $V(x) \leq V(q)$  then set  $q = x$ .
  - Repeat 3-4 until number of iterations.
  5.  $q$  is the child.
- 

## Mutation

Instead of using some random probability, we apply mutation if the DG distance between parent1 and parent2 are less than some predefined value. It is also defined based on the same idea as crossover. However, we choose the child which has the largest distance from the neighborhood. The algorithm for mutation is shown in Algorithm 2.

---

### Algorithm 2: Mutation

- 1 Set  $x = p_1$ .
- 2 Find Neighbourhood for  $x$ ,  $N(x)$
- 3 Do
  - a. For each member  $y_i \in N(x)$ , calculate the distance between  $y_i$  and  $p_1$ ,  $D(y_i, p_1)$ .
  - b. Sort the distance value in descending order,  $D_{sort}(y_i, p_1)$ .
  - c. Starting from  $i = 1$ , do
    - i. Calculate the fitness value for  $y_i$ ,  $V(y_i)$ .
    - ii. If  $V(y_i) \leq V(x)$  accept  $y_i$  with probability one, and with probability 0.5 otherwise.
    - iii. If  $y_i$  is not accepted, increase  $i$  by one.
  - Repeat i-iii until  $y_i$  is accepted.
  - d. Set  $x = y_i$ .
  - e. If  $V(x) \leq V(q)$  then set  $q = x$ .
- Repeat 2-3 until number of iteration.
- $q$  is the child.

we also consider if the child has the same fitness value with the member of population. Instead of dropping that child, we replaced the old one with the child assuming that we have given chance for the old individual to reproduce. Noted that we could not take both of the individuals to avoid having problem later, we might have problem falling in the local optima. The algorithm for the whole procedure is shown in Algorithm 3.

---

**Algorithm 3: Genetic Algorithm**

---

1. Initialize population: Randomly generated a set of 10 schedules including the schedules obtained by some priority rules.
2. Randomly select two schedules, named them as  $p_1$  and  $p_2$ . Calculate DG distance between  $p_1$  and  $p_2$ .
3. If DG distance is smaller from some predefined value, apply Algorithm 2 to  $p_1$ . Generate child. Then go to step 5.
4. If DG distance is large, we apply Algorithm 1 to  $p_1$  and  $p_2$ . Generate child.
5. Apply neighbourhood search to child to find the fittest child in the neighbourhood. Noted it as child'.
6. If the makespan for the child' is less than the worst and not equal to any member of population, replace the worst individual with child'. If there is a member having the same makespan value, replace the member with the child'.
7. Repeat 2-6 until some termination condition are satisfied.

## 2. Description of the experiments

### Evolutionary Algorithms

Evolutionary algorithms are general purpose search procedures based on the mechanisms of natural selection and population genetics. These algorithms have been applied :

1. Selection of the chromosome structure

The problem was supposed to be translated into a chromosome representation, Each gene of the chromosome corresponds to a decision variable of the problem. According to our problem complexity, the chromosome structure can be either conventional (a binary string) or not (reals, a series or a sequence of orders, a parallel form, etc.).

2. Initialization of the population of chromosomes

The initial population can be generated at random if the problem structure allows it.

3. Perform genetic operations on chromosomes Some operators was introduced in genetic algorithms to produce a solution. Among them there was two categories of operators: crossover and mutation.

4. Evaluation chromosomes

During evolutionary generation, an evaluating system was set up to assess the chromosomes and to select those chromosomes that are fit enough for the next generation.

### **I)Representation of a solution**

For flow shop problems with the permutation condition ( $\text{prmu} \in \beta$ ), a standard way of representing a feasible solution is the permutation code, i.e. an individual consists of a string of length  $n$ , and the  $i$ -th gene contains the index of the job at position  $i$ , so an individual describes the job sequence chosen on all machines. In the case of a regular criterion a permutation is decoded into a feasible solution by construction the resulting semi-active schedule. For job shop scheduling problems, the situation is bit different. Here several encoding strategies exist, and it is not clear in advance which is the best.

## **3 . Results from the performed experiments**

The result also gives us the job sequence for each machine to process, the starting time and the finish time for each operation. For example, on machine 1, we start to process job 3 at time 0 and finished at 7. Then we process job 1, followed by job 4, job 5 and job 2.

	Sequence1	Sequence2	Sequence3	Sequence4	Sequence5
Job1	Machine3	Machine1	Machine2	Machine4	Machine5
Job2	Machine2	Machine3	Machine5	Machine1	Machine4
Job3	Machine1	Machine5	Machine4	Machine3	Machine2
Job4	Machine4	Machine3	Machine2	Machine1	Machine5
Job5	Machine5	Machine3	Machine1	Machine2	Machine4

Table2: Processing Time for each Operation

	Machine1	Machine2	Machine3	Machine4	Machine5
Job1	8	4	2	6	7
Job2	3	6	5	2	4
Job3	7	3	9	4	8
Job4	4	5	5	4	3
Job5	3	6	7	4	5

Table3: Result

	Processing Time	Start	Finish
<b>Machine1</b>			<b>25</b>
Job3	7	0	7
Job1	8	7	15
Job4	4	15	19
Job5	3	21	24
Job2	3	24	27
<b>Machine2</b>			<b>24</b>
Job2	6	0	6
Job4	5	9	14
Job1	4	15	19
Job5	6	24	30
Job3	3	30	33
<b>Machine3</b>			<b>28</b>
Job1	2	0	2
Job4	5	4	9
Job2	5	9	14
Job5	7	14	21
Job3	9	21	30

We applied both types of initial population to the data. First we used the combination of schedules we generated using the priority rules and the randomly generated schedules as the initial population. From the five runs, we find the optimum before the generation exceeded 100. we found the optimum value at generation 139.

From the five runs, we could conclude that if we used the randomly generated schedules as the initial population, we will only find the optimum value at generation larger than 100. However, both results gave the same makespan value which is 34.

#### 4.General conclusion from the performed work

The application of evolutionary algorithms to a flexible job-shop scheduling problem with real-world constraints has been defined. We demonstrated that choosing a suitable representation of chromosomes (parallel encoding) is an important step to get better results.

A proper selection of genetic parameters for an application of EAs is still an open issue. These parameters (crossover rate, mutation rate, population size, etc.) are usually selected heuristically.

we use c sharp programming language for the implementation of this project. We encountered some difficulties in parameter settings during the implementation but finally we fixed the error.

The study on GA and job shop scheduling problem provides a rich experience for the constrained combinatorial optimization problems. Application of genetic algorithm gives a good result most of the time. Although GA takes plenty of time to provide a good result, it provides a flexible framework for evolutionary computation and it can handle varieties of objective function and constraint. For further research, the technique in this paper would be applied to a larger size problem to see how it performed

## 1. Problems description

### **Problem 3: Class schedule created by use of an evolutionary algorithm** **General Description of the problem**

Creating course schedules is a time consuming operation for both humans and computers. When creating a schedule, the scheduler must allocate resources (i.e. rooms, instructors, and meeting times) while ensuring that multiple constraints are satisfied. This paper presents an automated scheduling algorithm that applies the concepts of genetic Algorithms to the course scheduling problem. For ease of use, a graphical user interface has been incorporated into the system. The result is an easy to use scheduling algorithm that produces good schedules in a short time frame.

#### **The goal of the problem**

Every semester members of a college's faculty or staff must grapple with the problem of scheduling the courses to be taught in the next semester. Creating these semester schedules is a time consuming and error prone task that causes many frustrations for the person in charge of their creation. When creating a schedule, the scheduler must ensure that every course is assigned a classroom, instructor, and a timeslot.

The difficulty of creating schedules is due mainly to the fact that the scheduler must assign courses to a finite number of resources (e.g. classrooms, instructors, etc). When assigning these resources to a course, the scheduler must be sure that no conflicts are created, such as assigning two courses to the same room at the same time. Scheduling Conflicts often go undetected during the planning process and result in incorrect schedules being delivered to students.

#### **Fitness function description, what is important in the function, description of all variables.**

Now we need to assign a fitness value to the chromosome. As previously said, only hard requirements are used to calculate the fitness of a class schedule. This is how we do it:

- Each class can have 0 to 5 points.
- If a class uses a spare classroom, we increment its score.
- If a class requires computers and it is located in the classroom with them, or it doesn't require them, we increment the score of the class.
- If a class is located in a classroom with enough available seats, guess what, we increment its score.
- If a professor has no other classes at the time, we increment the class's score once again.
- The last thing that we check is if any of the student groups that attend the class has any other class at the same time, and if they don't, we increment the score of the class.



- If a class breaks a rule at any time-space slot that it occupies, its score is not incremented for that rule.
- The total score of a class schedule is the sum of points of all classes.

### **Constrains for the search space (testing for feasible solutions)**

- A course cannot be set to conflict with itself.
- Instructors cannot be specified for a course they are not qualified to teach.
- A course time cannot be set during the school's blackout times.
- An instructor cannot be given a time preference that occurs during the schools Blackout times.
- A room cannot be specified for a course if the room's capacity is less than the Course's capacity.

### **Encoding solutions of the problem for the used evolutionary algorithm (chromosome description, vector/matrix description)**

The first thing we should consider when we deal with a genetic algorithm is how to represent our solution in such a way that it is feasible for genetic operations such as crossover and mutation. Also, we should know how to specify how good our solution is. In other words, we should be able to calculate the fitness value of our solution.

#### **Representation**

How can we represent the chromosome for a class schedule? Well, we need a slot (time-space slot) for each hour (we assume that time is in one hour granules), for every room, every day. Also, we assume that classes cannot begin before 8am, and should finish before or at 17pm (9 hours total), and working days are from Monday to Friday (5 days total). If a class starts at 1pm and lasts for three hours, it has entries in the 1pm, 2pm, and 3pm slots.

#### **Fitness**

Now we need to assign a fitness value to the chromosome. As I previously said, only hard requirements are used to calculate the fitness of a class schedule. This is how we do it:

- If a class uses a spare classroom, we increment its score.
- If a class requires computers and it is located in the classroom with them, or it doesn't require them, we increment the score of the class.
- If a class is located in a classroom with enough available seats, guess what, we increment its score.
- If a professor has no other classes at the time, we increment the class's score once again.
- The last thing that we check is if any of the student groups that attend the class has any other class at the same time, and if they don't, we increment the score of the class.

- If a class breaks a rule at any time-space slot that it occupies, its score is not incremented for that rule.
- The total score of a class schedule is the sum of points of all classes.

### Crossover

A crossover operation combines data in the hash maps of two parents, and then it creates a vector of slots according to the content of the new hash map. A crossover 'splits' hash maps of both parents in parts of random size. The number of parts is defined by the number of crossover points (plus one) in the chromosome's parameters. Then, it alternately copies parts from parents to the new chromosome, and forms a new vector of slots.

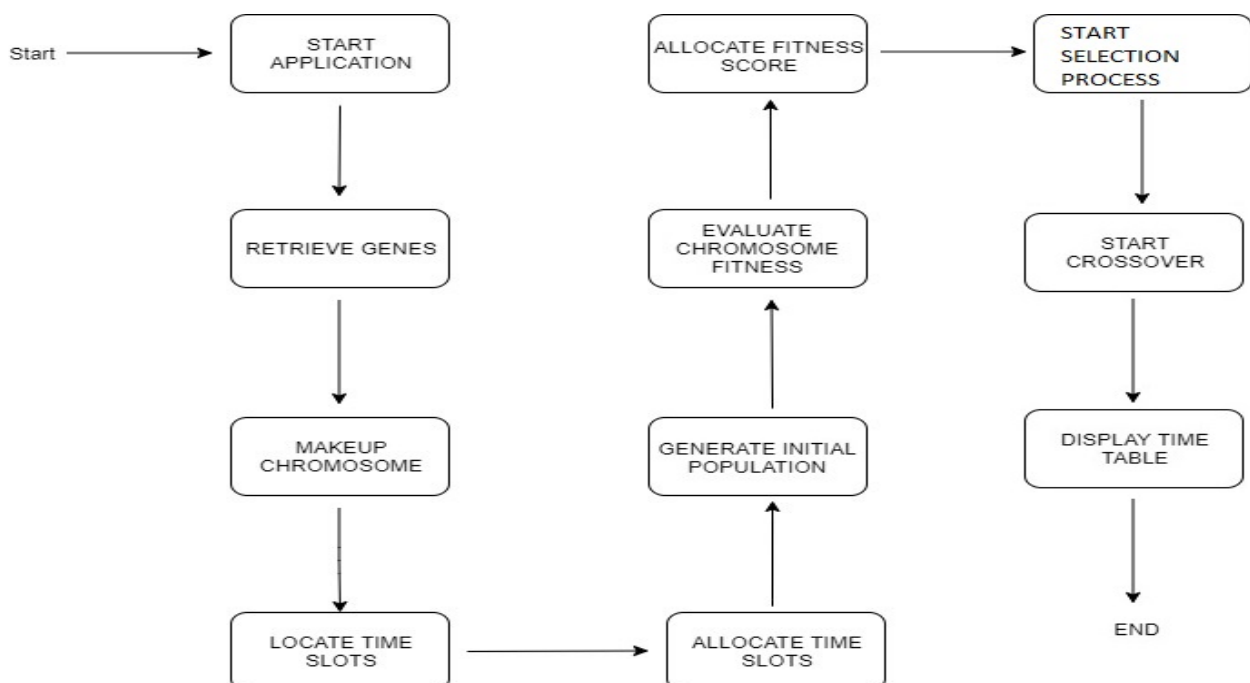
## 2. Short description of the used algorithm/algorithms

The genetic algorithm is fairly simple. For each generation, it performs two basic operations:

- Randomly selects N pairs of parents from the current population and produces N new chromosomes by performing a crossover operation on the pair of parents.
- Randomly selects N chromosomes from the current population and replaces them with new ones. The algorithm doesn't select chromosomes for replacement if it is among the best chromosomes in the population.

And, these two operations are repeated until the best chromosome reaches a fitness value equal to 1 (meaning that all classes in the schedule meet the requirement). As mentioned before, the genetic algorithm keeps track of the M best chromosomes in the population, and guarantees that they are not going to be replaced while they are among the best chromosomes

### How works



## **Source of the algorithms**

The Algorithm is implemented on python and we used pycharm as IDE. We used 2 importation import prettytable as prettytable and import random as rnd as Additional libraries. The codes are implemented using the help of YouTube tutorials

## **Description of the experiments**

We are facing some implementation errors which failed to give any output for the program  
The compiler cannot import then error is preventing us to see the output.

## **General conclusion from the performed work about:**

The topic is to implement Class schedule created by use of an evolutionary algorithm we have used python programming language to minimize the coding effort and time with the use of pycharm. WE have done multiple experiments on 24, 28 dates of January Due to some coding errors we are failed to get a proper output from this program.