Genetic Algorithm: The Knapsack Problem

Hiren Shah, Riddhi Kakadiya

Introduction to Genetic algorithm:

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, non differentiable, stochastic, or highly nonlinear. The genetic algorithm can address problems of mixed integer programming, where some components are restricted to be integer-valued.
The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- Selection rules select the individuals, called parents, that contribute to the population at the next generation.
- Crossover rules combine two parents to form children for the next generation.
- Mutation rules apply random changes to individual parents to form children.

Introduction to Problem statement :

In this project, we have used Genetic Algorithm to solve the 0-1 Knapsack problem (an optimization problem) where one has to maximize the benefit of all the items in a bag without exceeding its total capacity. A large variety of resource allocation problems can be cast in the framework of Knapsack problem. General idea is to think of the capacity of the knapsack as the available amount of a resource and the item types as activities to which the resource can be allocated.

Suppose we are given following parameters:

w_t = weight of each item, for t = 1,2,….,N

v_t = value of each item, for t = 1,2,….,N

c = capacity of the knapsack

The, our problem can be formulated as :

$$\sum_{k=1}^{N} w_t v_t <= c$$

Walkthrough to the process:

Java Classes :

- Main.java : initializes the products, population (also evolves it) and tracks the progress.
- Population.java: creates individuals, calculates average fitness & the fittest individual.
- Individual.java : randomly creates genes of each individual.
- Product.java : pojo class for products
- Values.java : contains assumed values

Test Classes :

KnapsackTest.java : tests using following methods

- generateGene(): checks uniqueness of randomly generated genes
- generateProduct(): checks the total products generated
- selectionTest(): tests the sorting mechanism i.e used to get the fittest individual
- evolveTest(): tests culling and cross-over of the population

Initial Steps:

- Initially set the no.of items to be inserted in the bag as required. Each item is assigned a weight and a value
- Set the capacity of the bag and the mutation probability
- Set the generation count and repeat the evolution of population for the generation count

Step 1: Initializing population

Initially create a population of 100 individual each having a unique chromosome. We have used binary encoding for generating chromosomes, where 0-gene is supressed, 1-gene dominates. 4

0		1		2	
4	5	8	4	7	1

Chromosome representation of selecting first and last item:

0	1	2
1	0	1

Step 2: Selection of individuals

Using priority queue, we sorted the top 75% of fittest population and culled the rest. Comparator lambda function was implemented to reversely sort the individuals.

Step 3: Crossover

Top 50% of the fittest were selected and mated. Expression of each gene among the two from each parent was decided randomly.

Consider the following:

Parent1:

1	0	1

Parent 2:

0	1	1

Child generated from crossover of Parent1 and Parent2:

0	0	1
From parent2	from parent1	from parent2

Now, the generation consists of 25% new population & 75% already existing population.

Step 4: Mutation

Mutation depends on Mutation Probability variable which was initially set by the user. For each child decimal number was randomly generated & if it was above Mutation Probability variable, mutation occurred. One gene (at randomly chosen position) of the child eligible for mutation was fliped.

Child selected for mutation:

0	0	1

Child chromosome after mutation (position 0 was selected to flip):

1	0	1

Step 5: Determining Fitness

Fitness is determined taking into consideration two traits; value and weight. If the bit in the chromosome is 1 then value of the item in that particular index is added to the fitness of the individual. If the fitness of the individual exceeds the capacity, set its fitness to 0 else return the total fitness.

Result:

The project was executed 8 times by taking different values of total items, capacity of the sack and mutation probability. The output from these runs is in logger.log.

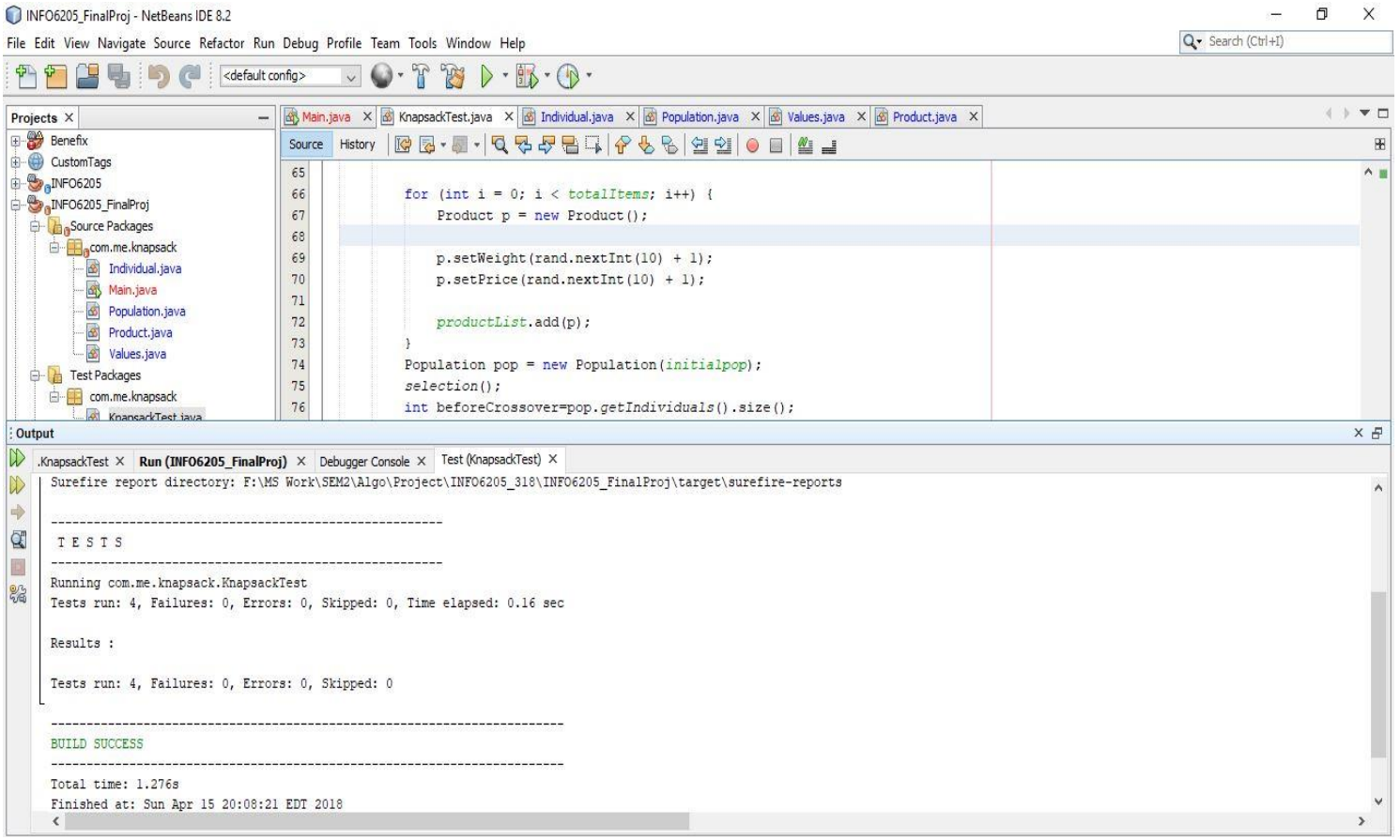
Following is the output of project for total items = 200, capacity = 835, total generation=75

```
0 [main] INFO com.me.knapsack.Main - Fitness of fittest chromosome of Generation 0: 718
1 [main] INFO com.me.knapsack.Main  - Average Fitness: 566.43

3 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosome of Generation 1: 718
3 [main] INFO com.me.knapsack.Main  - Average Fitness: 591.93

3 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosome of Generation 2: 762
4 [main] INFO com.me.knapsack.Main  - Average Fitness: 610.25

4 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosome of Generation 3: 762
4 [main] INFO com.me.knapsack.Main  - Average Fitness: 627.35
….
….
….
….

34 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosome of Generation 73: 904
34 [main] INFO com.me.knapsack.Main  - Average Fitness: 800.74

34 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosome of Generation 74: 906
34 [main] INFO com.me.knapsack.Main  - Average Fitness: 837.0
```

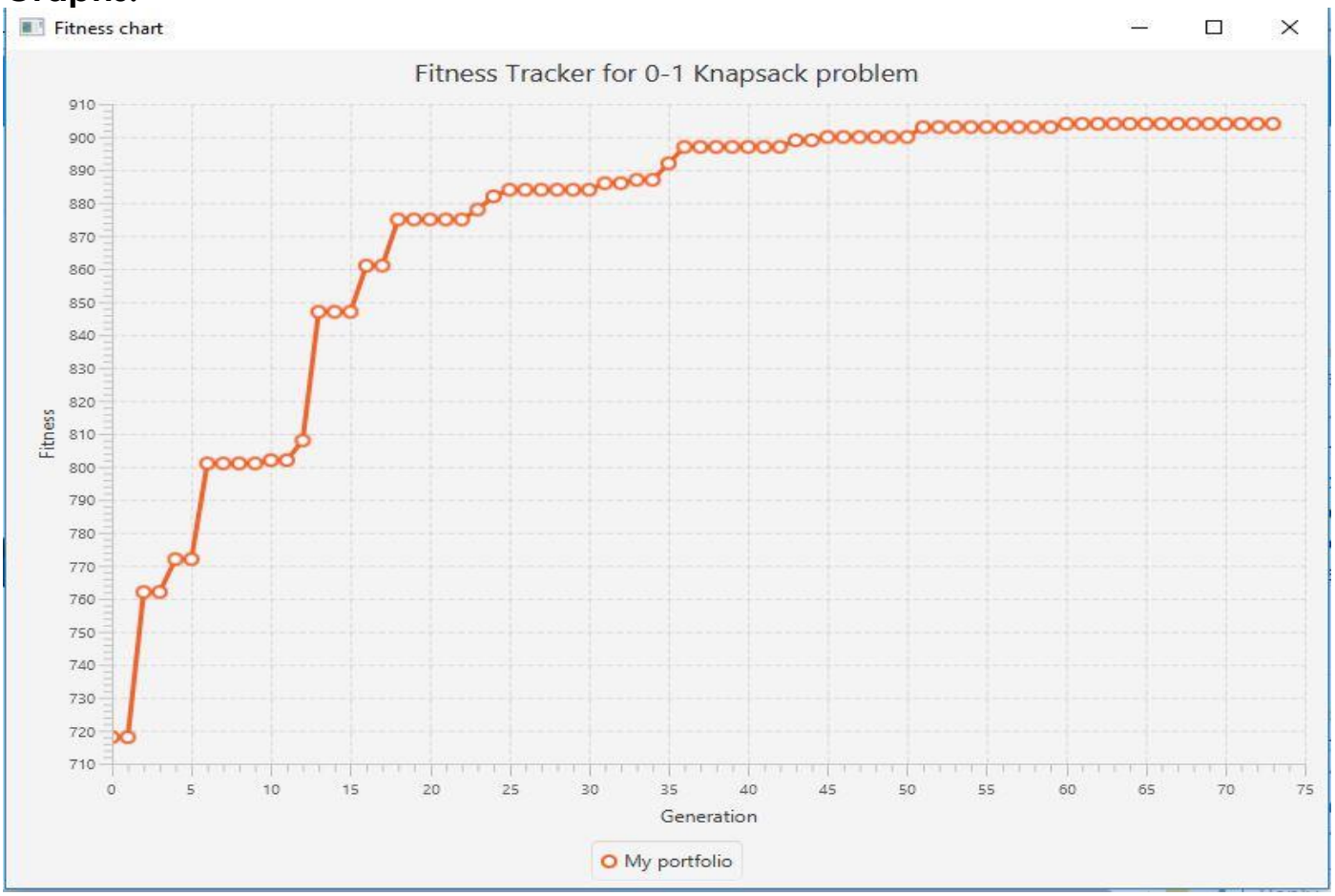
Following is the output of project for total items = 20, capacity = 40, total generation=14

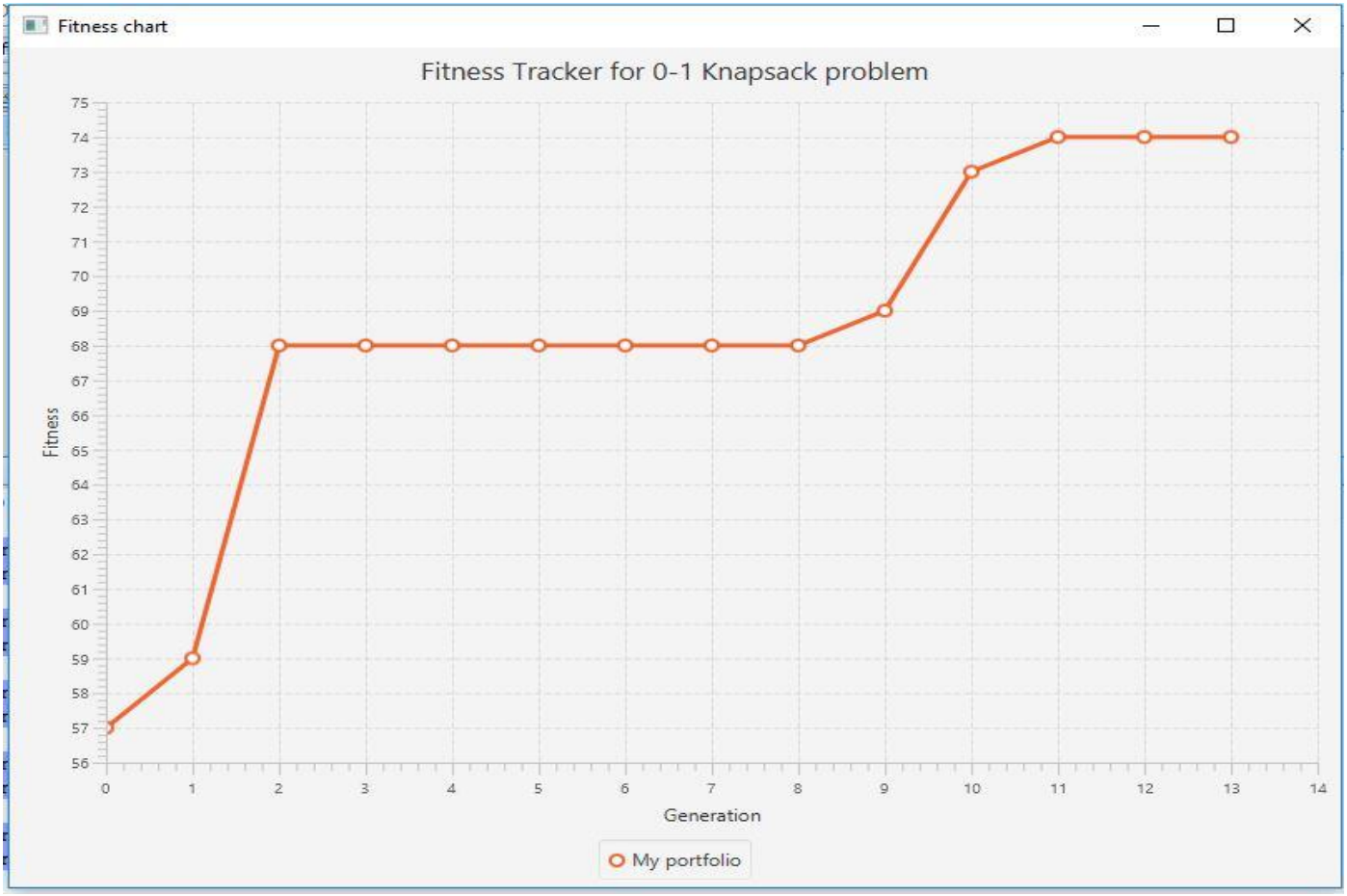
```
0 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 0: 57
2 [main] INFO com.me.knapsack.Main  - Average Fitness: 6.6

3 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 1: 59
3 [main] INFO com.me.knapsack.Main  - Average Fitness: 9.05

3 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 2: 68
3 [main] INFO com.me.knapsack.Main  - Average Fitness: 13.36

….
….
….

7 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 11: 74
7 [main] INFO com.me.knapsack.Main  - Average Fitness: 55.16

7 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 12: 74
8 [main] INFO com.me.knapsack.Main  - Average Fitness: 57.7

8 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 13: 74
8 [main] INFO com.me.knapsack.Main  - Average Fitness: 59.86

8 [main] INFO com.me.knapsack.Main  - Fitness of fittest chromosone of Generation 14: 82
8 [main] INFO com.me.knapsack.Main  - Average Fitness: 61.53
```

Test Cases:



Graphs:





Generated using JXchart

Observation:

From the graph, it can be observed that for the successive generations, fitness tends to improve.

Conclusion:

Thus, GS helps us to find optimal solutions for np problems like knapsack. Our results show implementation of a good selection method is very important for the good performance of a genetic algorithm.

Reference:

[1] https://en.wikipedia.org/wifi/Knapsack_problem