PROGRAM STRUCTURES AND ALGORITHMS
INFO 6205
KNAPSACK PROBLEM USING GENETIC ALGORITHM
TEAM 508

Northeastern University
College of Engineering

BY

Peter Vayda

Vishaka Varma Vimal

Christy Joseph Anoop

KNAPSACK PROBLEM

A knapsack problem or a Rucksack problem is a problem in combinatorial optimization. It states - Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

For example, consider you are traveling to a different city and you have a limitation on your baggage capacity, and hence there is a limitation on how much and what you can carry. The knapsack algorithm deals with finding a solution to this combinatorial problem.

Let **S** be a set such that S belongs to the items, and let **t** be the target value of the how much weight we can carry. Then, the problem is to find a subset **S'** of S whose elements sum to target t.

One of the problems being solved is the 0/1 Knapsack problem, where the number the x_i of copies of each kind of item is restricted to zero or one. Given a set of *n* items numbered from 1 up to *n*, each with a weight w_i and a value v_i, along with maximum weight capacity W,

Maximize

$$\sum_{i=1}^{n} vi\ xi$$

Subject to

$$\sum_{i=1}^{n} wi\ xi \leq W\ and\ xi\ belongs\ to\ \{0,1\}$$

Here, x_i represents the number of instances of item *i* to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

PROBLEM STATEMENT

One example where Knapsack algorithm is used is the preparation for exam paper just a night before exam. Indian students are mastered in applying the Knapsack solution while exam preparation. Let me explain how.

What is our thinking just a night before exams? Get more marks by less study in the remaining number of hours. Right? So, here we are trying to optimize our efforts for getting more marks. Now, we analyze our previous question papers and decide that which chapters have more value and which chapters have less. We also have a sense that how much time it will take to complete the chapter by checking the number of pages of the chapter in the book.

Here we are trying to maximize the marks by selecting the chapters whose questions have a high probability of asking in the exams. And, we must consider the hours required to complete these chapters. Now we check all the combinations of chapters and their values to find the list of chapters that needs to be studied and, add the hours required to study these chapters so the added hours should not exceed the hours we have for the exam.

This approach of solving the problem for exam preparation is analogous to the 0/1 Knapsack algorithm in which the student either skips the whole chapter and studies the whole chapter.

But wait! The story is not over yet. There are some daredevils among us who further want to reduce their efforts for getting more marks. They still ponder over the previous question papers and realize that some parts of the chapter have more value but not the whole chapter. So, they try to fragment chapter into sections and chose only those sections who have high probability of asking in the exams.

Here they find the highly expected questions in the exams and find the time required for them. Now, they find the ratio of marks for the question and the time required for it.

- *marksForQuestiontime/RequiredForQuestion*

They arrange the ratios into their descending order. Now they select the first question in the order because they know that it is highly expected, and it gives them more marks and it requires less time. And they prepare for the questions one by one in the given order until the hours they have until the exam are completely utilized.

This approach is much more efficient in time and efforts. It is again analogous to the Greedy Knapsack algorithm

in which student breaks the chapters into sections to maximize the marks in the given number of hours.

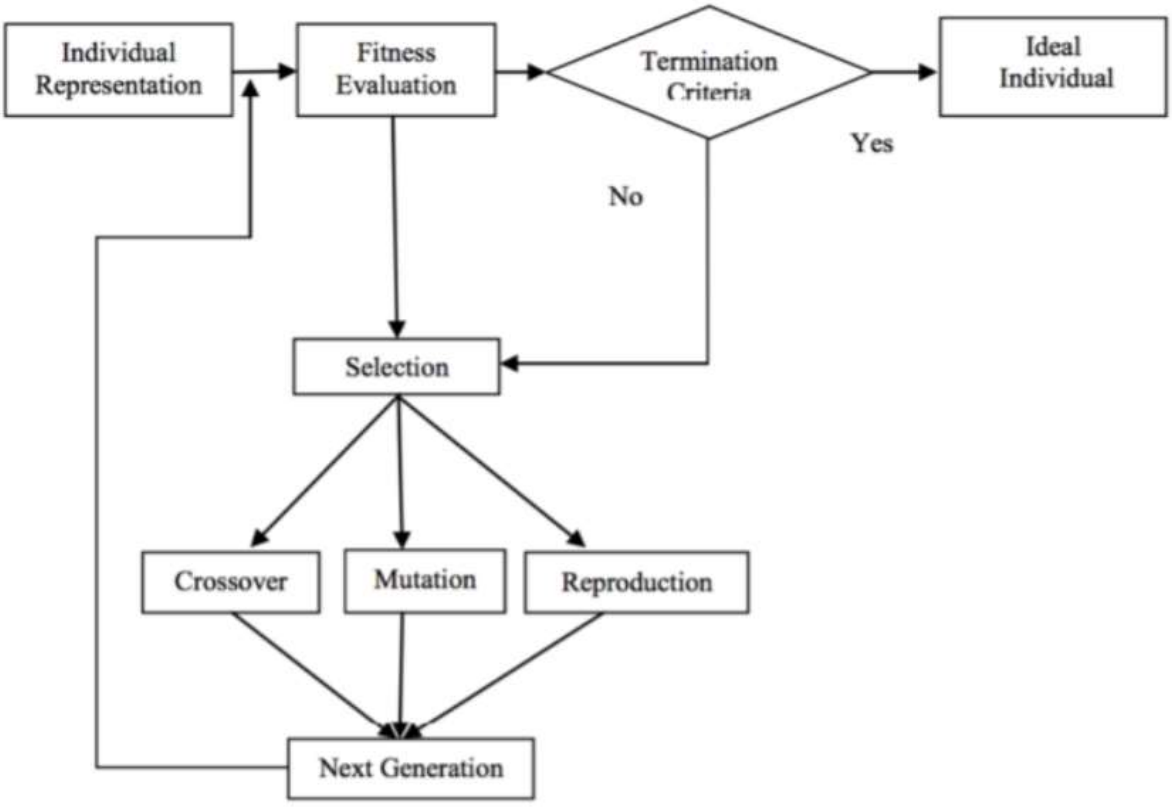
In this project, we are aiming to optimize the 0/1 Knapsack problem by implementing it using Genetic Algorithm.

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, nondifferentiable, 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.



How the Genetic Algorithm is implemented in our approach to optimize the Knapsack problem –

The basic idea of Genetic Algorithm is that it begins with a set of candidate solutions called **population**, and this population is optimized to evolve towards a better solution. Each candidate solution has a set of properties (chromosomes or genotype) which can be mutated and altered. These solutions are represented as strings of 0s and 1s. This evolution usually starts from a population of randomly generated individuals, and is an **iterative process**, with the population in each iteration called a *generation or an offspring*. Solutions that are chosen to form a new population – the **offspring or new generation,** is selected based on their "**fitness**". The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

We use this approach to repeatedly get a better combination of items to choose from to fit into our knapsack such that the most important and relevant items are picked to fill the knapsack to its capacity.

CODE WALKTHROUGH

Java classes :

- MainRun.java – this is the main program extending the application API for GUI.
- Population.java – creates individuals and calculates their fitness.
- Individual.java – generates genes and calculates the gene fitness and phenotype.
- Item.java – an individual item (POJO).
- Values.java – holds static values for the whole project.
- Simulation.java – performs crossovers and mutations.

Test classes :

- KnapsackTest.java – jUnit tester to test eight test cases for the program.

Step 1 :

Initialize Population

Create a population of 100 individuals each having a unique chromosome. This can be achieved through binary 0/1 encoding, where 0 denotes no item and 1 denotes the presence of an item in the knapsack.

Step 2 :

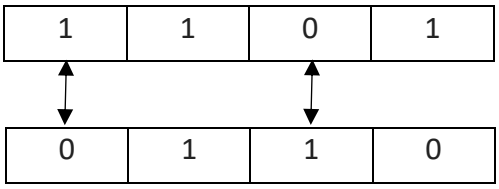
Selecting Individuals

Sort the Array List of fittest individuals using Comparator and selecting the top 50% of the population for crossover.

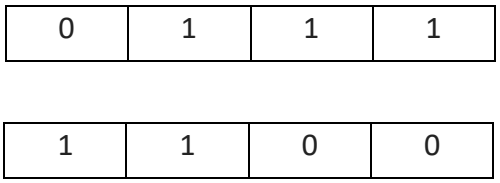
Step 3 :

Crossover

Subsequent individuals were chosen from the Array List and the next generation is generated by taking interchanging the individuals. The subsequent generation contains the new individuals plus the individuals from the previous generation to obtain the optimal fitness.



Previous Generation



New Generation After Random Crossover

Step 4 :

Mutation

Our algorithm naturally mutates individuals by inter-breeding the new generation with the previous generation. Explicit mutation was not implemented as it has a negligible impact on the final result.

Step 5 :

Determining Fitness

A graph is generated to check the stabilization of the fitness over 75 generations and the final fitness was determined.

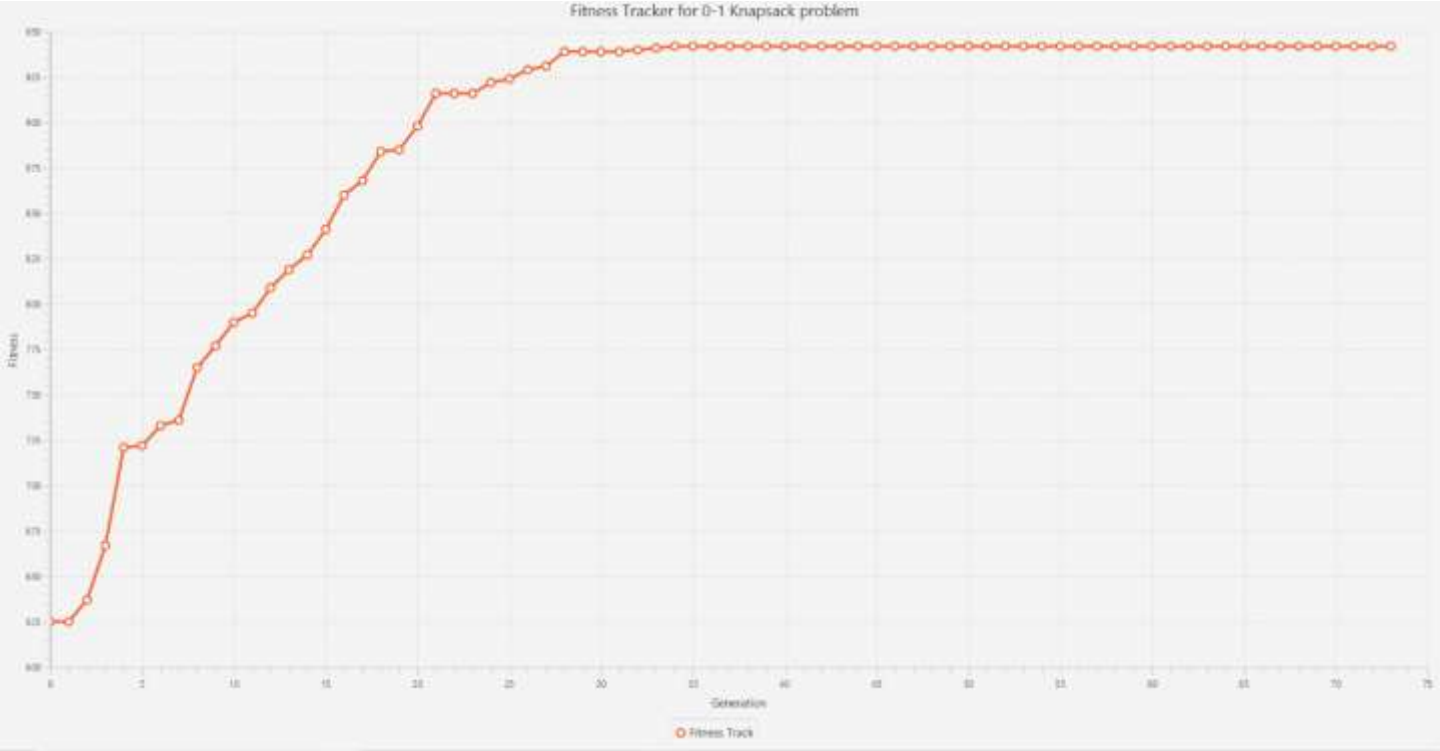
RESULTS

The following output is for these respective values –

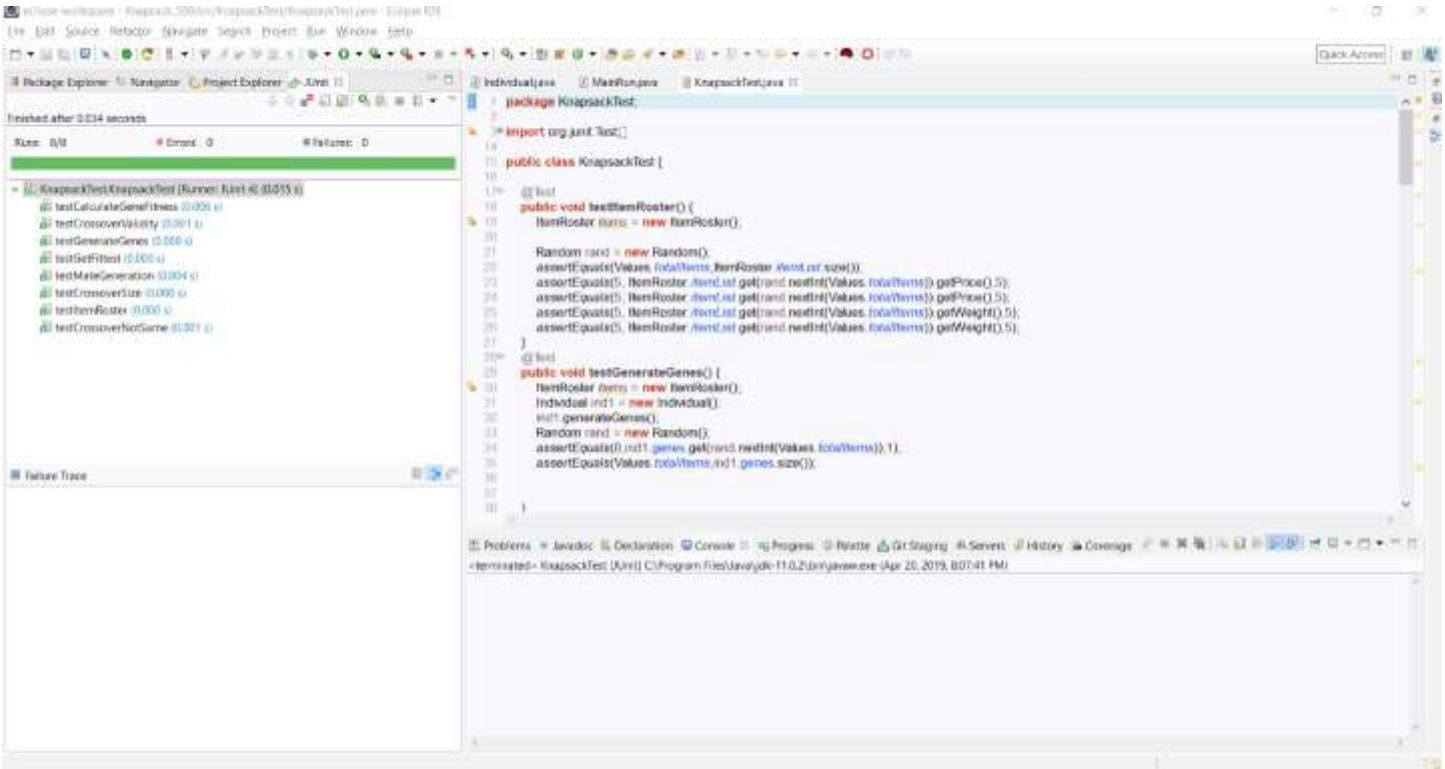
```
public class Values {

   public static int totalItems = 200;
   public static int initialPopulation = 100;
   public static int knapsackCapacity = 835;
   public static int totalGenerations = 75;

}
```

Fitness of fittest individual of Generation 1 is : 570
Fitness of fittest individual of Generation 2 is : 570
Fitness of fittest individual of Generation 3 is : 587
Fitness of fittest individual of Generation 4 is : 597
Fitness of fittest individual of Generation 5 is : 615
Fitness of fittest individual of Generation 6 is : 620
Fitness of fittest individual of Generation 7 is : 626
.
.
.
Fitness of fittest individual of Generation 61 is : 942
Fitness of fittest individual of Generation 62 is : 942
Fitness of fittest individual of Generation 63 is : 942
Fitness of fittest individual of Generation 64 is : 942
Fitness of fittest individual of Generation 65 is : 942
Fitness of fittest individual of Generation 66 is : 942
Fitness of fittest individual of Generation 67 is : 942
Fitness of fittest individual of Generation 68 is : 942
Fitness of fittest individual of Generation 69 is : 942
Fitness of fittest individual of Generation 70 is : 942
Fitness of fittest individual of Generation 71 is : 942
Fitness of fittest individual of Generation 72 is : 942
Fitness of fittest individual of Generation 73 is : 942
Fitness of fittest individual of Generation 74 is : 942

Fitness Graph :



Test Cases :



OBSERVATION

As observed from the generated graph, we can conclude that the fitness of the items tends to improve with successive generations.

SUMMARY

In conclusion, we have optimized the Knapsack Problem by implementing it using the Genetic Algorithm. Through our implementation, we have reduced the time complexity of the knapsack problem (usually ranging from exponential to linear) to a factor of the number of generations that the genetic algorithm takes to find the optimal solution. This also proves why genetic algorithms are good to implement NP problems. Therefore, selecting a good algorithm is crucial to optimize a problem and obtain an efficient result.

REFERENCES

https://en.wikipedia.org/wiki/Knapsack_problem
http://math.stmarys-ca.edu/wp-content/uploads/2017/07/Christopher-Queen.pdf
https://www.dataminingapps.com/2017/03/solving-the-knapsack-problem-with-a-simple-genetic-algorithm/
https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html
http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-knapsack.pdf
https://en.wikipedia.org/wiki/Genetic_algorithm
https://www.youtube.com/watch?v=JgqBM7JG9ew&t=1149s
https://www.quora.com/What-are-some-interesting-applications-of-the-knapsack-algorithm/answer/Suhas-Bhattu?ch=10&share=0d150bb2&srid=DcLI