Genetic Algorithm: Traveling Salesman

Shawn Yap

School of Information Technology  
734 E North 14th St. 79601 Abilene, TX  
Apt. D  
+1(325)232-2559

scy12a@acu.edu

Aldo Anaya

School of Information Technology

1207 Pueblo Rd  
Alamo, TX 78516  
+1(956)429-2598

axa13a@acu.edu

**ABSTRACT**

In this paper, we present the impact of the Genetic Algorithm on the Knapsack and the Traveling Salesman problem. First, we use the Genetic algorithm to solve the Knapsack problem. We then use the genetic algorithm to solve the Traveling Salesman problem. There were certain attributes which were changed to improve the performance of the genetic algorithm used for the Traveling Salesman Problem. We compare the results of the optimizations along with the default GA. The genetic algorithm used for the Knapsack problem is compared to the genetic algorithm used for the Traveling Salesman problem.

**CCS Concepts**

• **Symbolic and algebraic manipulation➝ Optimization Algorithm; Combinational Algorithm;**

**Keywords**

Combinatorial optimization; the Knapsack problem; the Traveling Salesman Problem; Genetic Algorithm;

# INTRODUCTION

The purpose of this project is to analyze the effectiveness of using a genetic algorithm, along with some optimization technique performed on the Knapsack Problem and the Traveling Salesman Problem. In this paper, we will discuss the methodology for each of the three phases. The paper will first transition into the phase 1 section in which we will show the methods and the results of solving the knapsack problem with genetic algorithm. We will then move into section 3 which we will talk about the second phase that shows the methods and results of solving TSP with genetic algorithm. In section 4, the paper will show the methods and the results gained from the third phase which is the optimization approaches that we have tried and recorded.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Conference’10*, Month 1–2, 2010, City, State, Country.

Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.

DOI: http://dx.doi.org/10.1145/12345.67890

# THE KNAPSACK PROBLEM BACKGROUND

Suppose a group of hikers is planning on a hiking trip and their plan is to fill their knapsack with items that are considered a necessity for the trip. There are N number of different items that have different **name, weight,** andthe **value** of the item. For example, there are water, sandwich, and more. Each item obviously has their own name, weight, and value. Since the hikers are only able to fit a certain amount of items into the knapsack due to the space or weight limit of how much the knapsack is able to hold, they have to obtain a combination of items that produces a maximum value while staying in within the weight limit of the Knapsack.

## Method Descriptions

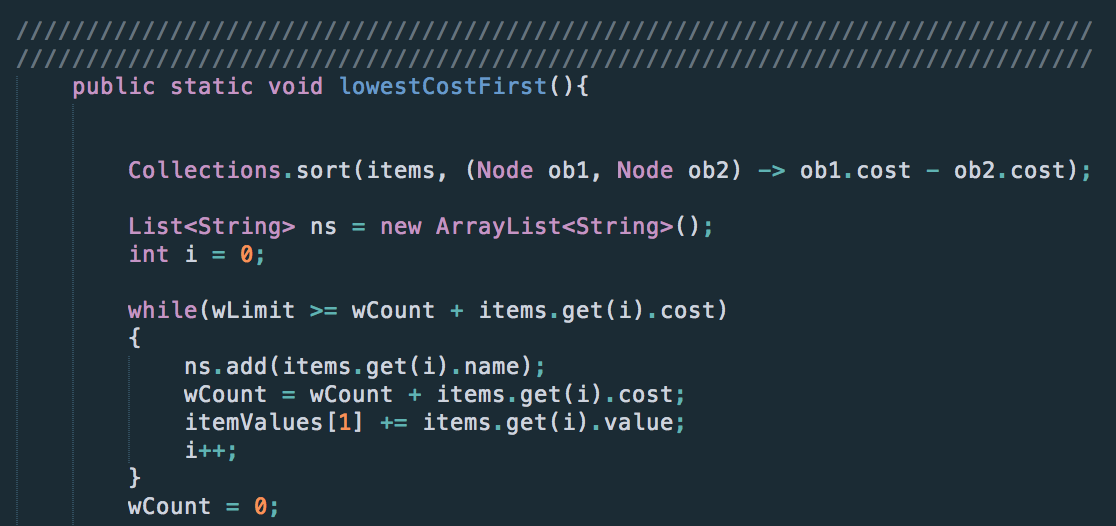
In this section, we will define the Knapsack problem, then we will provide the details of the attempted approaches in which we have tried to get the most optimal results.

### Greedy Solutions

The greedy approach is the primary methods that we have used to obtain the upper bound and the lower bound of the Knapsack problem. There are four type of sorts used to sort the list of items: Sort by highest-value first, sort by lowest-cost-first, sort by highest ratio first, and partial knapsack. However, since it is impossible to determine if the result returned from one of the approaches is the best one, all four approaches mentioned above were used to obtain the most optimal result with the cost and value being at its maximum.

#### Sort by Lowest-cost First.

*Code Snippet:*

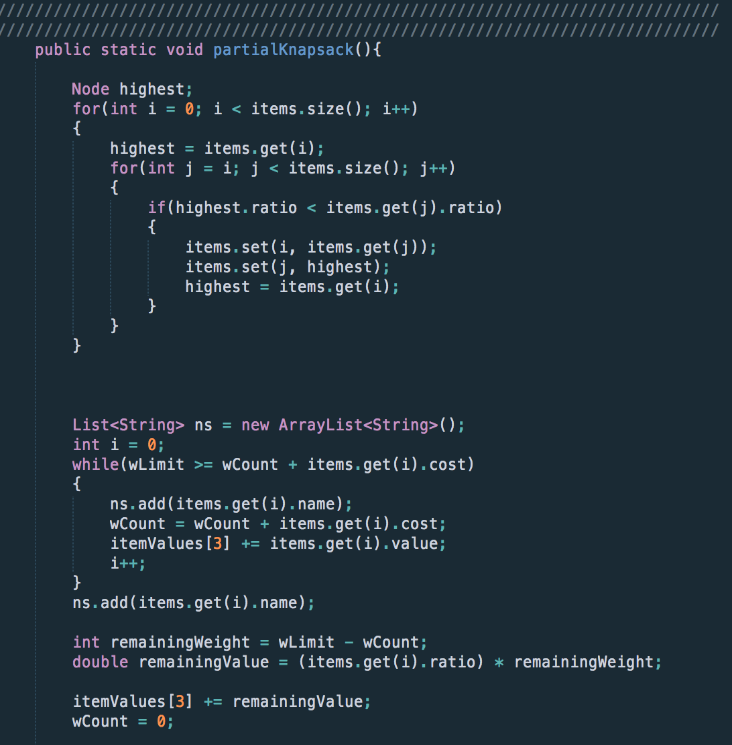


This solution sorts the given list of items from the lowest cost to the highest. By doing so, we are able to keep the cost the lowest for as many iterations as possible, allowing more space for more items to be included in the knapsack. Although this method would work for a list of items with low cost and high values, this method does not produce the optimal answer when the list of items contains both low cost and low values. There were cases where a single item would have a significantly higher value than the result obtained from this greedy approach.

#### Partial Knapsack

This greedy approach first sorts the given list of inputs by the highest cost to value ratio then grabs as many items as possible until the cost is near or equal to the limit that the problem is being constrained to. If the cost is less the constrained limit then we will continue to add part of the cost and value of the next item until we have the cost at its maximum.

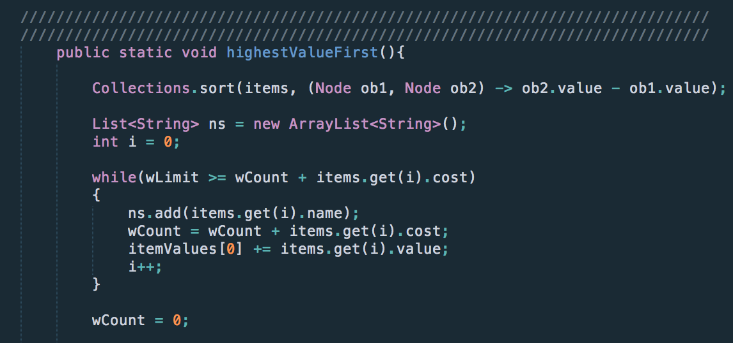
*Code Snippet:*



#### Sort by Highest-Value First

This solution first sorts the given set of items into a collection of highest values first and then sums the values from the top of the list with the highest values while keeping the cost within the range of the limit that the problem is constrained to. Although this algorithm prioritizes the highest values in the list first, there were cases where a more optimal answer was found by looking at the items with lower values with lower cost.

*Code Snippet:*



#### Sort by highest ratio first

This greedy approach sorts the given inputs by the highest ratio first allowing the algorithm to grab the item with the highest cost to value ratio.

*Code Snippet:*



### State Space Search

The state space is an approach that we have used to obtain every possible combination of the given list of items. Thus, allowing us to find the best possible combinations of the items that have its cost at its maximum and the values at its highest.

#### “Dumb” Exhaustive Search

Using a “dumb” exhaustive search, we looked at every possible leaf node of the tree. While looking through each leaf, we calculated the total cost and total value. The last row in the tree consists of all possible combination of items so if we look for the highest total value within the cost limit, clearly the best solution will be found.

#### “Smart” Search

The way we built the smart search will probably be a little different from how most people did it and it involves the way the tree was built. Initially, the tree was built with each node being a certain combination of items. However, with the smart search, the tree was built so that nodes that had already gone over capacity would receive a flag that would indicate that no calculation was needed to be done. This decreased would decrease the time taken to search through leaf nodes for the best solution to the knapsack problem.

# PHASE 2

In this section, we will talk about the TSP background, the methodology used, and the results.

## Traveling Salesman Problem Background

A traveling salesman needs to visit several cities and then return to the city from which it started. The task is to find the shortest possible route, given a list of cities and the distances between them, where each city is visited exactly once and then return to the original city.

## Methodology

### Genetic Representation

The TSP’s chromosomes are represented by the position number of each of the cities. For example, if there were 5 cities, then a chromosome can look like [1, 2, 3, 4, 5]. Each number represents one of the cities. Each chromosome represents a route (solution).

### Initialization

First, the cities and their positions are read from a file. Then, the initial generation is created.

The population size for each generation is a fixed set of 100 routes that are randomly generated.

### Selection

The genetic algorithm’s selection process for the TSP is determined by the route’s total distance traveled. The fitness function basically loops through the population and calculates the total distance of each route. Before the population is looped, the first route in the population is initialized to be the one with the best fitness. Then each route is then compared to the best fitness. If the route has a better fitness, then the variable that has the best fitness is replaced.

### Genetic Operators

The next generation is then generated using two genetic operators: the crossover and mutation function. The crossover function is used to create a child from two random parents. The way this is done is by grabbing a sequence of cities from the first parent and implanting them into the child. Then the cities from the second parent are placed into the route of the child in the same position. Once the child is created, it is passed through a mutation function. The mutation function passes through each of the cities in the route and swaps cities around with a 5% chance.

### Termination

Upon convergence of organisms - meaning that all routes are identical - one route is saved, and cataclysmic mutation is performed so that all other routes go through a 20% of mutation. After three successive cataclysmic mutations occur on the same organism, the program then ends. Also, if the program keeps running after 10 minutes, it will end and record the best results.s

# RESULTS

Table 1. Results of All Optimizations

|  |  |  |
| --- | --- | --- |
| **Optimization Methods** | **Time taken** | **Memory issue** |
| “Dumb” Exhaustive Search | 25(seconds) | None |
| Greedy | < 1 seconds | None |
| Tree with Nodes (Including Search) | Over a minute | Compile Error; insufficient  ram |
| Tree with Array (Including Search) | 22.43(seconds) | None |

As we can see from Table 1, the time taken to load trees with nodes was significantly longer than the tree with array. The Node class takes up quite a bit of memory, especially when making a tree with Nodes. The Greedy method is seen to have considerably fast search times. The obvious reason for that is because no tree was needed to be built nor requires to search through a huge state space.

# FUTURE OPTIMIZATION

Due to the time constraint of this Knapsack research project, there were optimization ideas that we could not implement in time but thought that it might be worth the time to dig-deeper into. To further optimize the huge memory usage used in order to build a -2) sized tree, while building the tree, we could cache the two most recent rows as row A and row B then build the next row using row B and then assign the new row as row A. Continue to iterate until all combinations are found. By doing so, we are not wasting more space than the program needs to. Therefore, opening up more space for larger lists of items. Another method that would prove to be even more efficient in terms of memory usage would be to delete one row as we are building the other row. For example, as we are building row B using the information from row A, we are also deleting used nodes from row A. This would greatly shorten the cost of memory by 50%. However, the time taken to run the program would significantly increase as well.

# REFERENCES

1. SAJJAN. S.P., Roogi, R., Badiger, V., and Amaragatti, S. 2014. A New Approach to solve Knapsack problem. Oriental Scientific Publishing
2. The Knapsack Problem. The University of Texas at Dallas. <https://www.utdallas.edu/~scniu/OPRE-6201/documents/DP3-Knapsack.pdf>