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; The Knapsack Problem; The Traveling Salesman Problem; Genetic 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 techniques 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 the 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 the TSP with the genetic algorithm. In section 4, we talk what can possibly be done in the future. The last section shows references to articles and sites from where we grabbed minor ideas and concepts.

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

# PHASE 1

In this section of the paper, we will be discussing about the KP background, the input file description, the methodology used, and the results.

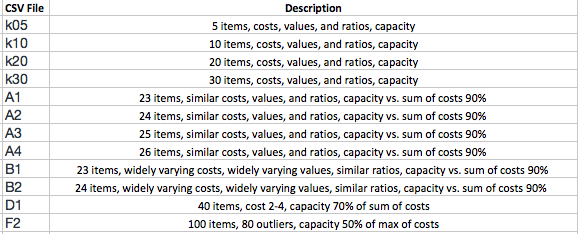
## The Knapsack Problem Background

The Knapsack problem is a classical problem that searches the highest combinational values from a list of items that consist of cost and values. The knapsack problem is a decision problem such that given a set of items that each consists of a cost and a value, find the highest possible value from each of the item while remaining in the range of the cost that the problem is being constrained. Which means that the cost may only remain less than or equal to the constrained cost limit. The decision form of the Knapsack problem is a NP-complete problem such that a precise solution for a huge input is nearly practically impossible to obtain. 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.

## Input File Description

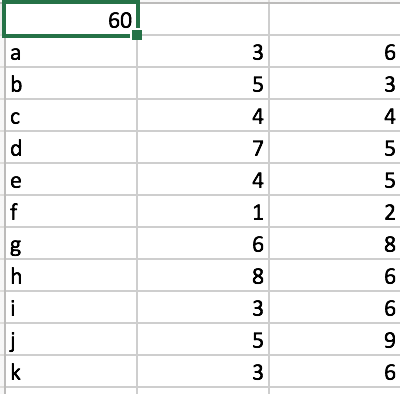
This section is used to define each and every test file that we have used with our knapsack problem. It is important to first understand the sample inputs that we have used in this KP research as well as the parameters that we are using for the genetic algorithm.

**Table 1. Description of each file**



In table 2, it shows some sample items in k24.CSV which is arranged in the same format as all the other test files. The first line of the csv file will always be the cost limit of the knapsack problem while each row from the second line onwards represents an item. Each item will have three columns which are arranged by name, cost, and values as illustrated in table 2 below.

**Table 2. Items in k24.CSV**



## Methodology

This part of the section will be broken down into many subsections to show the nature of how the genetic algorithm solves the knapsack problem. Since GA has a specific nature to its algorithm, it is important that we go through the elements of its algorithm to fully understand the structure of it.

### Genetic Representation../../Desktop/Screen%20Shot%202017-04-03%20at%202.17.58%20PM.png

Figure 1. Organism Representation

.

The KP organism is basically represented in n-bits of string in which consists of 0 and 1 where n is the number of items in the knapsack problem. The 0 and 1 in this representation are the item in the string that the algorithm considers as a solution as illustrated in figure 1. In figure 1, the program was tested with 24 items in a knapsack problem which generates a 24-bits string. Each index of the string represents an item in the list.

### Initialization

The program first randomly generates a list of population from a file of input items that consist of a name, cost, and value for each item, and a total cost limit from a csv file as illustrated in table 2. Random generation of population appends 0 and 1 into a string of n-bits length.

### Selection

The genetic algorithm’s selection process for the KP is determined by the total cost of each organism in a population. The fitness function basically goes through every single bit in an organism from a population and calculates the cost and value of each organism. The purpose of this fitness function is to determine the level of fitness of each organism in a population that would be used to produce an offspring that has a higher fitness level. The fittest will move on to the next generation of random population where the offspring will fight for survival with an organism that has the lowest fitness level in the population.

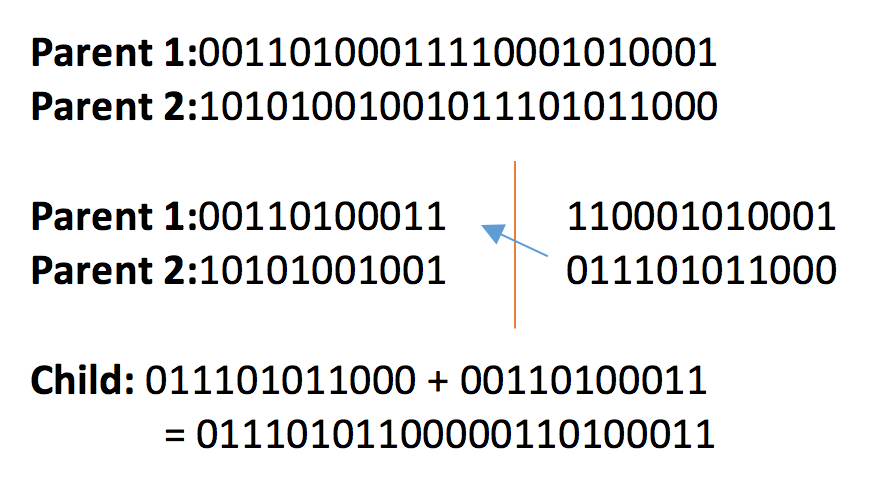
### Genetic operators

In this section of the paper, we will discuss the crossover and the mutation technique that we have used to solve the KP.

#### Crossover and Mutation

Figure 2. Crossover Technique

.



The crossover method that we have adopted into our program is to first select two random organisms from the population which will serve as our parents. After the parents are chosen, we then split the two parents into halves and then merge one part from each parent to form a child organism as shown in Figure 2 above. The cutting point or midpoint is found by n/2.

Once a child has been generated from the crossover process, we then mutate the child to keep the diversity of the population to avoid convergence to happen at a premature stage of the algorithm. Since our genetic representations are 0 and 1, it is extremely easy for us to adopt a mutation technique that can easily flip from 0 to 1 and vice versa with a mutation rate of as low as 0.5% per bits per string.

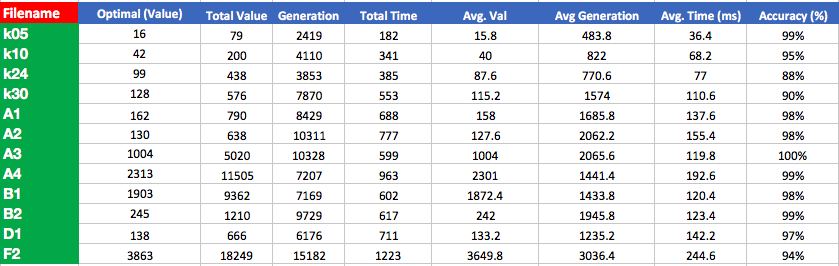
### Termination

In the termination process, the program will terminate when there are three successful cataclysmic mutations on the same organism or will automatically be terminated at ten minutes. Convergence happens when all of the population are equal to one another. Upon convergence, one organism is saved and the others will continue to be mutated with a 20% chance of mutation rate.

## RESULTS

This section will be used to show the results of solving the knapsack problem with GA by running the test files as described above in [section 2.2](#_Input_File_Description) to test the runtime differences as the number of item increases and attributes of the items changes.

**Table 3. Results from Test Files**

****

All of the results recorded in table 3 were calculated based on the total value, generation, and time obtained by running each test file for five times. The ***optimal*** column shows the most optimal solution obtainable for the specific list of items in a file. The ***total value*** *and* ***total time*** columns are the sum of all values and runtimes obtained from each run. The ***generation***column describes the number of iterations the program took to converge and generate a final solution. The **average value**, **average fit**,and **average time** are the total value, generation, and total time divided by the number of times the files were tested, in our case, five times. The ***accuracy***column shows how accurate the average value is as compared to the optimal value.

As we can conclude from the results illustrated in table 3 above, as the number of item increases, the time taken to find a solution does not significantly increase. In fact, the average time taken to find a solution for F2.CSV with 100 items in the file, takes only about twice the time of D1.CSV with 40 items. Also, the accuracy of both are almost as similar to each other with F2 at 94% and D1 at 97%.

# 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 bits in a population are represented by the position number of each of the cities. For example, if there were 5 cities, then an organism can look like [1, 2, 3, 4, 5]. Each number represents one of the cities, and each organism 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

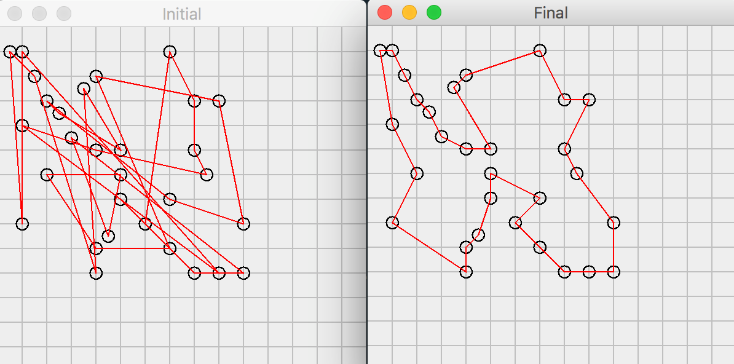
The next generation is then generated using two genetic operators: the crossover and mutation function. The crossover function is used to create a child from two random parents. The way this is done is by grabbing a sequence of cities from the first parent and implanting them into the child. Then the 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. When the program starts, it shows a graph of the initial route. However, when the program ends, it shows a graph with the final route.

Figure 3. Graph of Initial and Final Route Calculated

.



## TSP Optimizations

### Tournament Selection

The way the tournament selection method works in our project is that it randomly collects 5 different organisms. Out of these, the 2 organisms with the best fitness levels are picked to be the parents for the child, thus increasing the chances that the child will have better genes.

### Varied Population Size

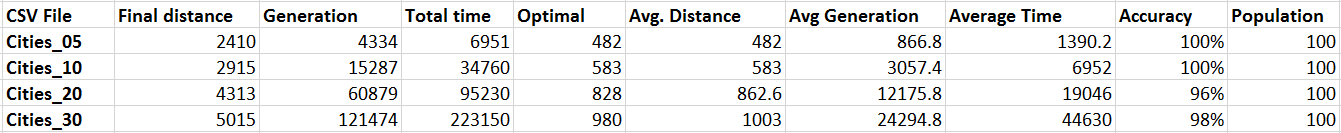
This optimization makes use of a varied population size to increase the chances of an organism mutating, thus increasing the chances of getting a more fit organism.

### Random Organism

This optimization creates a random organism which then replaces the lowest fit organism in the population, thus increasing the chances getting a more fit organism.

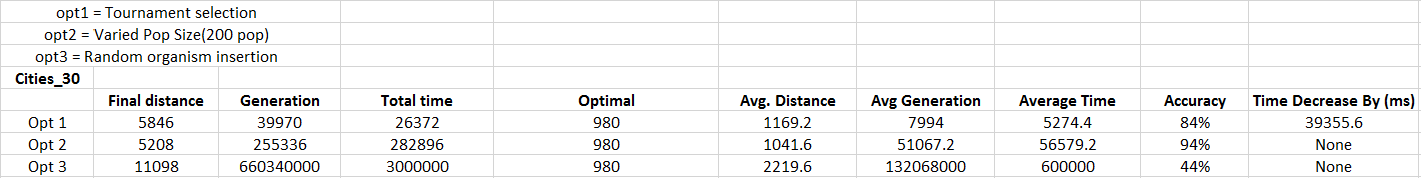
## Results

**Table 4. Results from Test Files**



The results recorded in Table 4 were conducted in a similar fashion as those done in Table 3 where each file was run 5 times and averages were calculated. Unsurprisingly, the genetic algorithm seemed to work best for a smaller amount of cities because of the smaller combination of cities. Surprisingly, accuracy was higher for the file that contained 30 cities than for the file that contained 20 cities.

**Table 5. Results with Optimizations**



The results recorded in Table 5 were conducted in a similar fashion as those done in Table 3 and Table 4 where only cities\_30.csv was run 5 times for each optimization and averages were calculated. Opt 1 had interesting results. Even though our accuracy dropped by about 14% as compared to the default GA setting, it did manage to decrease about 40 seconds of runtime. The results were peculiar because we had assumed that the Tournament Selection function would increase time and increase accuracy but the results showed the inverse. Opt 2 had a about twice the time average with a slightly lower accuracy. We would have expected Opt 2 to have had better accuracy since with a bigger population there would have been a bigger chance for the fitter routes to appear. Opt 3 ran to 10 minutes each time and a huge decrease in accuracy. It probably never stopped running because there was a new random organism and a child being added each time while only a single organism with the lowest fitness was being replaced.

# FUTURE WORK

Throughout the implementation of this project, we thought of another way that could be used in future work. Instead of only giving the child a chance of mutation, we thought we could get interesting results if every – except for the fittest - organism was given a chance of mutation for the new generation. This however, would make the convergence of organisms increasingly difficult to accomplish. Thus, we propose to limit the number of generations to a fixed amount.

# 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>
3. Traveling salesman problem. Wikipedia. https://en.wikipedia.org/wiki/Travelling\_salesman\_problem
4. Groba, Carlos, Antonio Sartal, and Xosé H. Vázquez. "Solving the dynamic traveling salesman problem using a genetic algorithm with trajectory prediction: An application to fish aggregating devices." *Computers & Operations Research* 56 (2015): 22-32.