COS 314 Assignment 2

Tayla Orsmond u21467456

# Comparing the Effectiveness of Utilizing a Genetic Algorithm (GA) and Ant Colony Optimization (ACO) in Solving the Knapsack Problem

## Introduction

The 1-0 knapsack problem is an optimization problem where a subset of n items, each with a weight and value, must be placed into a knapsack such that the total weight of the items in the knapsack does not exceed the capacity, and such that their combined value is maximized. In the simplest form of the problem the items are one dimensional and as such their shape / volume does not have to be considered. Items also cannot be placed more than once into the knapsack, nor can only a portion of the item exist in the knapsack.

This assignment aims to compare the effectiveness of using a Genetic Algorithm (GA) against Ant Colony Optimization (ACO) in solving instances of the knapsack problem.

## Environment (Specs)

This assignment was completed on a Dell Inspiron 7490 laptop with the following specs:

**Processor**: Intel(R) Core (TM) i7-10510U CPU @ 1.80GHz 2.30 GHz

**RAM**: 16.0 GB (15.8 GB usable)

**Development Environment**: Visual Studio Code, JDK version 17.0.6

## Genetic Algorithm (GA)

A genetic algorithm is a meta-heuristic that applies Darwin's principle of natural selection to produce optimal solutions. The algorithm begins by generating an initial population of individuals (chromosomes) and, through several generations, modifies and updates the population to reflect the evolving individuals.

Offspring are produced from selected parents who form part of the new generation that is expected to be more fit than the previous one. The fitness of the individuals is evaluated by a fitness function, and the algorithm is biased towards fitter individuals such that, over time, the average fitness of the population increases, and the population converges towards an optimal solution. The algorithm stops when a specified criterion is met.

When designing a GA, several different methods and parameters must be carefully chosen and developed. In the context of the knapsack problem, these are discussed below.

### Individual Representation

An individual is represented as a chromosome made up of several genes. Individuals are represented as binary fixed-length strings (chromosomes) where each bit (gene) represents whether an item from the problem instance set is selected. The number of bits in the string equals the total number of items in the instance. For example, 10110 would indicate that the 1st, 3rd, and 4th items from the instance set are selected to be placed in the knapsack.

Individuals are initially randomly generated and can have a weight that exceeds the capacity of the knapsack (this is dealt with in the fitness function). The size of the population is controlled by the **populationSize** parameter.

### Fitness function

The knapsack problem aims to maximize the total weight of the knapsack without exceeding its capacity. As such, the fitness function sums the values of all the items in the knapsack with the best fitness being the maximum sum value. If the combined weight of the items exceeds the capacity of the knapsack, the individual is given a fitness of 0.0.

### Selection

Tournament selection is used. A random number of individuals are selected from the population (defined by the **tournamentSize** parameter) that form the tournament. The best individual is then selected from the tournament to be a parent to produce offspring for the next generation. This is done for both parents.

Two perturbation operations are used (below) to produce offspring from the selected parents.

### Crossover

Single point crossover is used to recombine parents into children. This is controlled by theparameter **crossoverRate**. If the random number rolled is less than the crossover rate, crossover occurs. A random point from 0 to the parent’s size is used as the crossover point. The parents then swap genes after that point to produce offspring. Single point crossover is effective in exploiting current best solutions while being simple to implement.

### Mutation

Bit-flip mutation is used to modify children further. This is controlled by the **mutationRate** parameter. If the random number rolled is less than the mutation rate, mutation occurs. A bit (gene) is randomly selected from the individual, and this is flipped (1 to 0 or 0 to 1) to either add or remove an item from the knapsack. This is effective in exploring the search space and allowing for diversity in individuals. The mutation rate is typically lower than the crossover rate, so the search does not become stochastic.

### Elitism

This algorithm uses elitism which allows a few (fittest) individuals from the old generation to replicate in the new generation unchanged. This allows for better exploitation of fitter individuals and is controlled by the **numElite** parameter.

### Replacement

Generational replacement is used. A new generation replaces the older generation if the average fitness of the newer generation is better. Generational replacement allows the algorithm to converge faster than steady-state replacement.

### Termination

The algorithm terminates after a fixed number of iterations (**numGenerations** parameter).

### Parameters

The initial parameters were taken from literature and are shown in the table below (Julstrom, 2005). These were then fine-tuned to improve each algorithm’s performance.

This was done offline, by starting with initial values for each parameter and slowly adjusting the parameters incrementally until there was no longer an improvement in the algorithm. The values were both increased and decreased to attempt to improve performance. “Improvement” was considered as an average enhancement in both algorithms across all the instances over multiple runs.

Below is a list of parameters and their initial and final values after fine-tuning:

|  |  |  |
| --- | --- | --- |
| Parameter | Initial Value | Final Value |
| populationSize | Number of items | Number of items \* 3 |
| tournamentSize | 2 | 2 |
| crossoverRate | 0.7 | 0.8 |
| mutationRate | 0.3 | 0.6 |
| numElite | 1 | 1 |
| numGenerations | 10 \* Number of items | 10 \* Number of items |

increasing the **tournamentSize** and **numElite** parameters worsened the algorithm’s performance, with the worst results appearing when these values were set to above 4. This could be because the number of items for many instances were small, and so a larger tournament size turned the algorithm into a random search. Surprisingly, a **mutation rate** of 0.6-0.9 led to better performance, with more optima achieved on average. This could be because mutation only flips one bit - meaning individuals are only changed a small amount. This could also be because the crossover rate was already quite high initially. The **numGenerations** parameter had little impact on performance, with increasing it only increasing the time taken to complete the algorithm and decreasing it resulting in worse performance.

Increasing the **population size** had a positive impact on the algorithm's performance, resulting in more optima achieved and solutions closer to the optima. Decreasing the population size worsened the performance significantly. Increasing this parameter also resulted in more steady number of optima per run, instead of the sporadic results before.

GA relies heavily on the **populationSize** parameter due to the stochastic nature of the algorithm and of individual generation, as the more diversity within the population initially, the more likely the algorithm will converge into a (global) optimal solution. Thus, it makes sense that changing this parameter has the greatest impact on the algorithm's performance.

## Ant Colony Optimization (ACO)

Ant colony optimization is a meta-heuristic that mimics the behaviour of ants and other swarm animals to find an optimal path to a solution. The “ants” (agents) all begin at different points on a graph and traverse the graph, incrementally constructing a solution path by visiting nodes (solution components).

Components are selected based on their probability of forming a good candidate solution – a combination of their heuristic value and a pheromone value. Pheromone values are deposited by ants according to the quality of the solution formed from those components. Pheromones also evaporate according to a set evaporation rate. Over time the ants converge along the same solution path towards the optimal solution. The algorithm stops when a specified criterion is met.

When designing an ACO algorithm, many parameters must be carefully chosen. These are discussed below.

### Generating Solutions

Several ‘ants’ are initially chosen to construct solutions (given by the **numAnts** parameter).

Each ant constructs a candidate solution (a filled knapsack) by adding solution components (items) to the knapsack iteratively. The ants select components via roulette wheel selection. The ants stop adding items to the knapsack once the knapsack’s capacity is reached to produce valid candidate solutions. These solutions are then evaluated, and pheromones updated to reflect the quality of the solution path generated with those components.

### Component Selection

Roulette wheel selection is a selection process whereby each component is given a probability that represents the upper bound of a ‘slice’ of an imaginary roulette wheel, a random number is rolled which ‘lands’ within some portion of the wheel (or just below that upper bound) – and thus that item is chosen.

This method is extremely effective for ACO as it allows for exploitation as elements with higher heuristic and pheromone values are more likely to be chosen, but still allows for exploration by allowing a random (sometimes worse) component to be chosen.

The ants start by adding one random component to the knapsack to help with exploration of the search space. Components that would result in a knapsack whose weight exceeds the capacity of the knapsack are given a probability of 0.

The probability of the item being selected is calculated by the heuristic value (to the power of parameter **beta** – the heuristic weight) of the item multiplied by the pheromone value (to the power of parameter **alpha** – the pheromone weight) of that item, and the product is then normalized by dividing it by the sum of all probabilities of the neighbouring candidate components. This is done for all candidate components. The equation used for selection is given below:

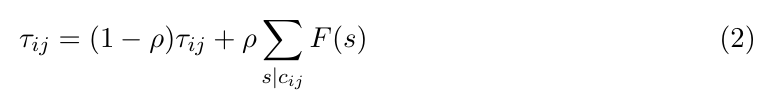
A picture containing text

Description automatically generated

### Pheromones and Pheromone Value

Each item starts off with an initial pheromone value (**tau0**) that is equal for all items. Since neither the order of items, nor having specific combinations of items, matters in this problem, one pheromone value can be associated with each item, instead of an item combination.

Ants then deposit pheromones by increasing the pheromone values of each item based on the quality of the solution produced with that item. The pheromone update rule used is given below. The pheromones are updated according to the pheromone evaporation rate (**rho**) and the result of the fitness function’s evaluation of a solution containing that item.



Pheromones are then globally evaporated using the above equation where the fitness is set to 0. To avoid items’ pheromone values being too high or low and escape local optima, items’ pheromone values are kept in a range dictated by the **tauMax** and **tauMin** parameters.

### Fitness function and Heuristic Value

Individual item’s heuristic value is calculated as the ratio of the item’s value to its weight in order that items with the highest value to weight ratio have the highest probability to be chosen (i.e., items with the highest value for the minimum weight). This value does not change.

As with GA, the fitness function sums the values of all the items in the knapsack with the best fitness being the maximum sum value. If the combined weight of the items exceeds the capacity of the knapsack, the solution is given a fitness of 0.0.

### Termination

The algorithm terminates after a fixed number of iterations (**numIterations** **parameter**).

### Parameters

The initial parameters were taken from literature and are shown in the table below (Schiff, 2013). These were then fine-tuned to improve each algorithm’s performance.

This was done offline, by starting with initial values for each parameter and slowly adjusting the parameters incrementally until there was no longer an improvement in the algorithm. The values were both increased and decreased to attempt to improve performance. “Improvement” was considered as an average enhancement in both algorithms across all the instances over multiple runs.

Below is a list of parameters and their initial and final values after fine-tuning:

|  |  |  |
| --- | --- | --- |
| Parameter | Initial Value | Final Value |
| numAnts | 60 | 5 |
| numIterations | 200 | 60 |
| alpha | 0.1 | 0.1 |
| beta | 0.5 | 2.0 |
| rho | 0.95 | 0.95 |
| tau0 | 0.1 | 0.1 |
| tauMax | 1.0 | 1.0 |
| tauMin | 0.1 | 0.1 |

These parameters took much more effort to tune than for the GA, since there are so many and because changing one parameter (e.g., alpha, beta or rho) at a time seemed to make little difference in the algorithm's performance regardless of whether the parameter was increased or not.

This seemed to be the case because of the **small number of items present** in most of the problem instances, meaning that no matter what parameters were set, the algorithm would find the optimal solution within one or two iterations. ACO relies heavily on the **number of ants** (**numAnts**) parameter as the more opportunity for diversity in the initial solutions, the more likely the algorithm is to stumble upon / converge to the optimal solution and this is especially true for instances where the number of items is very small (like these).

Another issue is the **heuristic** **function ()**, which worked extremely well and was extremely suited to most of the problem instances, so the algorithm was able to pick out the best solution easily due to the wide gap between items with good and bad heuristic values.

Because of these reasons, **none of the parameters hold weight in the algorithm, besides numAnts**. To try fix this, I **lowered the number of ants significantly** and tweaked some of the values from there to improve the performance for knapPI\_1\_100\_1000\_1 since this was the only instance which was affected by something other than **numAnts** (the rest would consistently be solved within one or two iterations even with ants lowered) so that might give me a better clue about other parameters. Unfortunately, the power of the heuristic function was too great and simply increasing beta was enough to swing the ACO performance. None of the other parameters made any difference to the algorithm’s performance regardless of this.

**I fear that this had turned into overfitting the algorithm thus I was hesitant to try other combinations of values to produce better results after this point**.

## Results

## Conclusion

Because GA constructs solutions randomly it naturally benefits from a large population size, regardless of the number of items in the problem instances. So, tuning the **populationSize** parameter increased performance significantly. However, because ACO intelligently constructs solutions, there are only so many valid solutions it can construct with so few values for most of the instances. This coupled with a very good heuristic function that widely separates good and bad items, the other parameters become unnecessary if the number of ants is high enough as it is likely that at least one ant will come across the optimal solution within one or two iterations. It was thus very difficult to tune these parameters without overfitting the algorithm.

## References

1. Julstrom, B.A. (2005) Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem, *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pp. 607–614. Available at: <https://dl.acm.org/doi/pdf/10.1145/1068009.1068111>
2. Schiff, K. (2013) “ANT COLONY OPTIMIZATION ALGORITHM FOR THE 0-1 KNAPSACK PROBLEM,” *Technical Transactions*, pp. 40–52. Available at: <https://doi.org/10.4467/2353737XCT.14.056.3964>