# 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 0-1 knapsack problem is an optimization problem in which a set of n items, each with a weight and value, must be selected to be placed into a knapsack. The goal is to maximize the total value of the items placed in the knapsack while ensuring that their combined weight does not exceed the knapsack's capacity. In this problem, items are one-dimensional and their shape or volume does not need to be considered. Additionally, an item can only be placed in the knapsack once, and it cannot be partially included.

The objective of this assignment is to compare the effectiveness of two metaheuristic algorithms, Genetic Algorithm (GA) and 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 a 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 a 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 performance of the algorithm, particularly when these values were set above 4. This is likely because many instances had a small number of items, so a larger tournament size turned the algorithm into a random search.

Surprisingly, a **mutation rate** of 0.6-0.9 resulted in better performance, with more optimal solutions achieved on average. This may be because mutation only flips one bit, resulting in a small change to the individuals. Additionally, the crossover rate was already quite high initially.

The **numGenerations** parameter had little impact on performance, as increasing it only increased the time taken to complete the algorithm, and decreasing it resulted in worse performance.

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

The **populationSize** parameter had the greatest impact on the performance of a genetic algorithm due to the stochastic nature of the algorithm and individual generation. 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 process of selecting components where each component is assigned a probability that represents the upper bound of a 'slice' of an imaginary roulette wheel. A random number is generated, and the selected component corresponds to the slice in which the random number lands. This selection process is highly effective for ACO because it allows for exploitation of components with higher heuristic and pheromone values while still enabling exploration by occasionally selecting random, possibly worse components.

To explore the search space, the ants start by adding a single random component to the knapsack. Components that would cause the knapsack's weight to exceed its capacity are assigned a probability of 0. The probability of selecting a component is calculated by multiplying the component's **heuristic** **value (eta)** (raised to the power of the heuristic weight parameter **beta**) by its **pheromone** **value (tau)** (raised to the power of the pheromone weight parameter **alpha**). The resulting product is then normalized by dividing it by the sum of the probabilities of all candidate components. This calculation is performed for each candidate component.

The equation used for selection is given below:

A picture containing text

Description automatically generated

### Pheromones and Pheromone Value

At the start of each iteration, all items have an equal initial pheromone value (**tau0**). As the ants traverse the search space and find good solutions, they deposit pheromones on the items they have used. Rather than updating pheromones for specific combinations of items (since the order of items in the knapsack does not matter for this problem), each item has a single associated pheromone value. The amount of pheromone deposited on an item is determined by the quality of the solution found with that item.

The pheromone update rule used in the algorithm is based on the fitness function's evaluation of a solution containing that item, as well as the pheromone evaporation rate (**rho**). The pheromones are updated based on the following formula:

A picture containing text

Description automatically generated

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 to escape local optima, items’ pheromone values are kept in a range dictated by the **tauMax** and **tauMin** parameters. By updating the pheromone values in this way, the algorithm can give more weight to good solutions and encourage exploration of promising areas of the search space.

### 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. 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 and was done and measured in the same way as GA.

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 required much more effort to tune compared to the GA, primarily because there are many parameters and changing one parameter at a time seemed to make little difference in the algorithm's performance, regardless of whether the parameter was increased or not. This was likely due to the **small number of items present** in most of the problem instances, where the algorithm would find the optimal solution within one or two iterations, regardless of the parameter values.

ACO relies heavily on the **numAnts** parameter as it provides more opportunities for diversity in the initial solutions, leading to a higher likelihood of the algorithm converging to the optimal solution. This is particularly true for instances where the number of items is very small. The **heuristic** **value function** (value/weight) worked well and was well-suited to most of the problem instances, so the algorithm could easily identify the best solution due to the significant gap between items with good and bad heuristic values.

As a result, none of the parameters had a significant impact on the algorithm's performance, except for **numAnts**. To address this issue, the **number** **of** **ants** was significantly reduced, and some values were tweaked to improve the performance on knapPI\_1\_100\_1000\_1, the only instance affected by something other than **numAnts**. However, the power of the heuristic function was so great that simply increasing **beta** was enough to improve ACO performance. None of the other parameters had any impact on the algorithm's performance, regardless of this.

There is a risk that this approach may have resulted in overfitting the algorithm, so further exploration of other parameter combinations was approached with caution.

## Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Problem Instance** | **Optimal Value** | **ACO** | | **GA** | |
|  |  | Best Solution | Runtime (seconds) | Best Solution | Runtime (seconds) |
| f1\_l-d\_kp\_10\_269 | 295.0 | 295.0 | 0.006 | 295.0 | 0.006 |
| f2\_l-d\_kp\_20\_878 | 1024.0 | 1024.0 | 0.025 | 1024.0 | 0.027 |
| f3\_l-d\_kp\_4\_20 | 35.0 | 35.0 | 0.002 | 35.0 | 0.000 |
| f4\_l-d\_kp\_4\_11 | 23.0 | 23.0 | 0.001 | 23.0 | 0.000 |
| f5\_l-d\_kp\_15\_375 | 481.0694 | 481.0694 | 0.013 | 481.0694 | 0.007 |
| f6\_l-d\_kp\_10\_60 | 52.0 | 52.0 | 0.006 | 52.0 | 0.001 |
| f7\_l-d\_kp\_7\_50 | 107.0 | 107.0 | 0.003 | 107.0 | 0.002 |
| **knapPI\_1\_100\_1000\_1** | 9147.0 | 9147.0 | 1.339 | 0.0 | 1.039 |
| **f8\_l-d\_kp\_23\_10000** | 9767.0 | 9747.0 | 0.040 | 9757.0 | 0.018 |
| f9\_l-d\_kp\_5\_80 | 130.0 | 130.0 | 0.002 | 130.0 | 0.000 |
| f10\_l-d\_kp\_20\_879 | 1025.0 | 1025.0 | 0.040 | 1025.0 | 0.036 |
| Total Time |  |  | 1.477 |  | 1.136 |

**Table 1: Comparison of ACO and GA on 11 knapsack problem instances**

|  |  |
| --- | --- |
| **ACO** | **GA** |
| Total Instances: 11 | Total Instances: 11 |
| Total Optimal: 10 / 11 | Total Optimal: 9 / 11 |
| % Optimal: 90.9090909090909% | % Optimal: 81.81818181818183% |
| Total Time: 1477ms | Total Time: 1136ms |
| Average Time: 134ms | Average Time: 103ms |

**Table 2: Summary of ACO and GA’s performance on 11 knapsack problem instances**

## Conclusion

As GA constructs solutions randomly, it benefits naturally from a large population size, regardless of the number of items in the problem instances. Therefore, tuning the **populationSize** parameter significantly increased its performance. However, ACO intelligently constructs solutions, and there are only so many valid solutions it can create with so few values for most instances. This, together with a very good heuristic function that separates good and bad items widely, makes the other parameters unnecessary if the number of ants is high enough, as at least one ant is likely to find the optimal solution within one or two iterations. Thus, tuning these parameters without overfitting the algorithm is challenging.

Despite these issues, as shown in Table 1 above, ACO performed better than GA for most instances, achieving approximately 91% optimal solutions (10/11 optima), while GA achieved approximately 82% (9/11). In the instance where both algorithms failed to find an optimal solution, both obtained similar suboptimal solutions, with GA finding a slightly better solution faster. However, in instance knapPI\_1\_100\_1000\_1, GA did not find an optimal or near-optimal solution, whereas ACO did. This instance has many items with values and weights that are very similar to each other, and the item weights are close to the capacity. In such cases, ACO may outperform GA due to its ability to construct solutions intelligently that do not exceed the knapsack capacity. Nevertheless, it is also possible that ACO was overfitted to this instance, although it had outperformed GA in several runs, whether or not it achieved an optimal solution. ACO may be more suitable for instances with a greater number of items that require more careful consideration of values and weights since its decision rules make it much more likely to achieve optimal solutions than GA's stochastic nature, thus performing much faster in instances where GA would have to brute-force the optimal answer through many iterations.

In general, GA was faster than ACO, achieving an average runtime of 103ms compared to ACO's 134ms. Therefore, in cases where speed is more important than accuracy, and where solutions can be achieved relatively quickly (due to GA's stochastic nature), GA may be a better option than ACO, as ACO may run much longer unnecessarily due to the many calculations it has to make.

## 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>