Dominique da Silva – u21629944

COS 314 ASSIGNMENT 2

KNAPSACK PROBLEM

The knapsack problem is a classic optimization problem in computer science. Given a set of items, each with a value and a weight, determine which items to include in the collection so that the total weight is less than or equal to the given limit and the total value of the collection is as large as possible. This is a combinatorial optimization problem that is NP-complete, meaning that there is no known polynomial-time algorithm to solve it exactly for all instances of the problem.

CONTENTS

GA CONFIGURATION DESCRIPTION

DESCRIPTION

Representation: Solutions are encoded as chromosomes using a binary encoding scheme.
Each gene in a chromosome represents a binary decision variable in the solution, i.e. 1 then the item has been included into the knapsack, 0 if the item has not been included. Chromosome length is equal to the number of items of the problem instance, or the number of decision variables. To map the chromosomes back to actual solutions, the binary genes are decoded back to actual solutions and corresponding values, based on their position within the chromosome.

Initialization: Population solutions are randomly generated. The number of solutions in the population is determined or rather depended on the problem size. Thus, a larger problem requires a larger population size. Each chromosome in the population is randomly generated by assigning a randomized binary value to each gene of the chromosome.

Selection: Tournament selection style was used. A subset of the population is randomly selected and the fittest chromosome is chosen as a parent for crossover. The size of the tournament selection style is typically small, and a fraction of the population size, we have chosen 4 individuals. This selection style favors the fitter individuals of the population, although still allows for diversity.

Crossover: Single-point crossover was used. Random point along the chromosome is selected, and genes beyond that point is then swapped between the two parents to create two offspring chromosomes. This is a simple crossover method, creating offspring while preserving some of the information of the parents.

Mutation: Bit-flip mutation was used. A random gene in a chromosome is flipped from 0 to 1 and vice versa. Mutation rate is typically set to a small value such at 1% or less, in order to prevent too much disruption of the population. This helps introduce new genetic material into the population and introduces diversity.

Termination: The stopping criterion used to terminate the algorithm is a maximum number of generations.

CONFIGURATION

Population size: In the final implementation of the algorithm, it was decided to use200 individuals. This was a random choice, as we wanted a larger population. Initially we used 10 times the number of items of the knapsack problem, in order to introduce a larger population, in hopes of introducing more diversity for any larger or more complex instances of an instance of the knapsack problem, however this was inefficient as larger instances of the problem, would have an unreasonably big population size, and smaller instances, would still not have a sufficient population to apply the algorithm to.

Crossover rate: Initially 80%. This choice was made as an initial implementation in order to test whether the algorithm somewhat works. This value was chosen by ChatGPT. Logically it made sense since we wanted to, once again, introduce a lot of diversity into our population. Alternative implementation had a rate of 90%. This was due to testing, we saw a significant improvement with our smaller instances of the problem, however, the larger instances of the problem was not near optimal. After some research, it was decided to rather scale down the crossover rate to 0.6. We do not always want crossover to happen, as in nature. The recommended value for a crossover rate was between 0.4 and 0.6, we found 0.6 to be sufficient. 60% likelihood of crossover still introduces enough diversity into our population.

Mutation rate: Initially 10%, then scaled up to 20%, with good success rate. This was also randomly chosen, by a ChatGPT guideline that the mutation rate should always be lower. https://arpitbhayani.me/blogs/genetic-knapsack . The recommended rate was between 0.01 and 0.02 for the mutation rate. The final value selected was 0.02 as the mutation rate, as we do not want to prematurely converge, but still want to still introduce some diversity into our population.

Selection method: Tournament selection style was used initially and remained within the final submission. This method was discussed in class, and most online articles relating to Genetic Algorithms state that this method is sufficient enough, as it is not to complex, but it still able to find optimal values for the instance of the knapsack problem. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 . Tournament size of 4 was used.

Crossover method: Single-point crossover was used, simple method and introduces diversity.

Mutation method: Bit-flip mutation. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 .

Termination criterion: Maximum number of iterations. Chosen number was 10000 iterations, in order to ensure that we evolve our population enough for the best possible solution for the knapsack instance.

ACO CONFIGURATION DESCRIPTION

DESCRIPTION

The ACO algorithm is a metaheuristic algorithm inspired by the foraging behavior of ants. In the algorithm, ants construct solutions by selecting items to put in the knapsack based on pheromone trails and heuristic information. The pheromone trails represent the ants' memory of previous good solutions, and the heuristic information guides the ants towards promising solutions. The pheromone trails are updated based on the quality of the solutions found, and the algorithm repeats until a termination criterion is met.

The code first initializes the pheromone trails with a default intensity value and calculates the value-to-weight ratio for each item. Then, it runs the main loop of the algorithm, where for each generation, it constructs a solution for each ant, evaluates it, and updates the pheromone trails. The algorithm repeats until a fixed number of generations are reached.

The constructSolution method constructs a solution for a single ant. It calculates leave and take probabilities for each item based on the pheromone trails and heuristic information and selects the item with higher probability. The evaluateSolution method calculates the fitness of a given solution, which is the total value of the selected items. The initialisePheromones method initializes the pheromone trails with a default intensity value, and HeuristicInformationItems calculates the value-to-weight ratio for each item.

CONFIGURATION

Number of ants: The number of ants determines how many ants will be used to construct the solution. In this implementation, the number of ants is set to 500.

Pheromone Evaporation Rate: The pheromone evaporation rate determines how quickly the pheromone trails evaporate. In this implementation, the evaporation rate is set to 0.1, meaning that 10% of the pheromone trails evaporate over each iteration.

Pheromone Intensity: The pheromone intensity determines the initial intensity of the pheromone trails. In this implementation, the pheromone intensity is set to the value of the value to weight ratio of each item.

Heuristic Information: The heuristic information is a value that guides the ants to select items that are likely to result in higher value solutions. In this implementation, the heuristic information is calculated as the value to weight ratio of each item

Ant Decision rule: The ant decision rule is the method used by the ants to decide which items to select. In this implementation, the decision rule is based on the probabilities calculated using the pheromone trails and heuristic information.

Local Search strategy: The local search strategy is the method used to improve the solution constructed by the ants. In this implementation, a local search strategy is not used.

Termination criteria: Termination criteria was set to a maximum number of iterations, i.e. 10000. This was chose due to two factors, the algorithm was efficient enough when set to this termination condition. Secondly, due to the GA being written with this termination condition, I wanted to truly compare the efficiency of both algorithms, when they execute for the same number of times.

EXPERIMENTAL SETUP

* Red blocks indicate an optimal rate percentage of above 90%.

File name	Optimal	GA	GA %	ACO	ACO%	GA seconds	ACO seconds
f1_l-d_kp_10_269	295	295	100.00%	182	61.69%	0	0
f2_l-d_kp_20_878	1024	987	96.39%	758	74.02%	1	0
f3_l-d_kp_4_20	35	35	100.00%	33	94.29%	0	0
f4_l-d_kp_4_11	23	23	100.00%	16	69.57%	0	0
f5_l-d_kp_15_375	481.0694	481.0694	100.00%	352.9954	73.38%	0	0
f6_l-d_kp_10_60	52	52	100.00%	52	100.00%	0	0
f7_l-d_kp_7_50	107	107	100.00%	102	95.33%	0	0
knapPI_1_100_1000_1	9147	1754	19.18%	1914	20.92%	13	0
f8_l-d_kp_23_10000	9767	9761	99.94%	1950	19.97%	1	0
f9_l-d_kp_5_80	130	130	100.00%	130	100.00%	0	0
f10_l-d_kp_20_879	1025	999	97.46%	759	74.05%	1	0

Table A

File name	Optimal	GA	GA %	ACO	ACO%	GA run	ACO run
f1_l-d_kp_10_269	295	295	100.00%	182	61.69%	0	0
f2_l-d_kp_20_878	1024	1013	98.93%	758	74.02%	2	0
f3_l-d_kp_4_20	35	35	100.00%	33	94.29%	0	0
f4_l-d_kp_4_11	23	23	100.00%	16	69.57%	0	0
f5_l-d_kp_15_375	481.0694	481.0694	100.00%	352.9954	73.38%	1	0
f6_l-d_kp_10_60	52	52	100.00%	0	0.00%	0	0
f7_l-d_kp_7_50	107	107	100.00%	102	95.33%	0	0
knapPI_1_100_1000_1	9147	2527	27.63%	1914	20.92%	19	10
f8_l-d_kp_23_10000	9767	9764	99.97%	1950	19.97%	3	0
f9_l-d_kp_5_80	130	130	100.00%	130	100.00%	0	7
f10_l-d_kp_20_879	1025	999	97.46%	759	74.05%	3	0

Table B

File name	Optimal	GA	GA %	ACO	ACO%	GA run	ACO run
f1_l-d_kp_10_269	295	295	100.00%	182	61.69%	0	0
f2_l-d_kp_20_878	1024	1024	100.00%	758	74.02%	2	0
f3_l-d_kp_4_20	35	35	100.00%	33	94.29%	0	0
f4_l-d_kp_4_11	23	23	100.00%	16	69.57%	0	0
f5_l-d_kp_15_375	481.0694	481.0694	100.00%	352.9954	73.38%	1	0
f6_l-d_kp_10_60	52	52	100.00%	52	100.00%	0	0
f7_l-d_kp_7_50	107	107	100.00%	102	95.33%	0	0
knapPI_1_100_1000_1	9147	3118	34.09%	1914	20.92%	41	7
f8_l-d_kp_23_10000	9767	9765	99.98%	1950	19.97%	3	0
f9_l-d_kp_5_80	130	130	100.00%	130	100.00%	0	0
f10_l-d_kp_20_879	1025	1025	100.00%	759	74.05%	3	0

Table C

File name	Optimal	GA	GA %	ACO	ACO%	GA run	ACO run
f1_l-d_kp_10_269	295			216	73.22%		0
f2_l-d_kp_20_878	1024			930	90.82%		0
f3_l-d_kp_4_20	35			33	94.29%		0
f4_l-d_kp_4_11	23			16	69.57%		0
f5_l-d_kp_15_375	481.0694			368.3767	76.57%		0
f6_l-d_kp_10_60	52			50	96.15%		0
f7_l-d_kp_7_50	107			102	95.33%		0
knapPI_1_100_1000_1	9147			1914	20.92%		19
f8_l-d_kp_23_10000	9767			9756	99.89%		1
f9_l-d_kp_5_80	130			130	100.00%		0
f10_l-d_kp_20_879	1025			985	96.10%		0

Table D

GENETIC ALGORITHM

Experimental Setup for Genetic Algorithm (GA) applied to Knapsack Problem:

Entities:

1. Population: A collection of individuals representing potential solutions.
2. Individual: A candidate solution represented by a binary array indicating item inclusion/exclusion in the knapsack.
3. Fitness Function: Evaluates the quality of an individual based on the total value and weight of selected items in the knapsack.
4. Selection Operator: Determines which individuals are chosen for reproduction.
5. Crossover Operator: Combines genetic information from two parents to create offspring.
6. Mutation Operator: Introduces random changes in the genetic information of an individual.
7. Termination Condition: Determines when the algorithm should stop iterating.

Representation:

1. Chromosome: Binary array representing the presence or absence of items in the knapsack.
2. Genes: Elements of the chromosome, each corresponding to an item.

Components:

1. Initial Population Generation: Randomly generate a population of individuals, each with a randomly initialized chromosome.
2. Fitness Evaluation: Calculate the fitness of each individual in the population using a fitness function that considers the total value and weight of the selected items. The total weight it's merely considered since we do not want to exceed the limit that was given for the given instance of the

problem. If the weight has been exceeded then we return a value of 0, else we return the total values of the items included in the solution.

3. Selection: Choose fitter individuals from the population for reproduction using a tournament selection approach.

4. Crossover: Apply crossover operation (e.g., single-point crossover) to selected parents to create offspring.

5. Mutation: Introduce random changes in the genetic information of offspring by flipping individual bits.

6. New Population Generation: Generate a new population using the offspring.

7. Termination Check: Determine whether the termination condition is met (e.g., maximum number of generations reached).

8. Best Individual Selection: Identify the best individual in the final population based on fitness.

Flow Control:

1. Initialize the knapsack problem instance with the weight and value arrays, as well as the maximum weight constraint.

2. Generate an initial population of individuals with randomly initialized chromosomes.

3. Evaluate the fitness of each individual by calculating the total value and weight of the selected items in the knapsack.

4. Repeat the following steps until the termination condition is met:

   a. Select fitter individuals for reproduction using tournament selection.

   b. Apply crossover operation (e.g., single-point crossover) to create offspring.

   c. Introduce random mutations in the offspring by flipping individual bits.

   d. Evaluate the fitness of the offspring.

   e. Generate a new population using the offspring.

   f. Check the termination condition (e.g., maximum number of generations reached).

5. Select the best individual from the final population based on fitness.

6. Return the fitness of the best individual as the solution to the knapsack problem.

Overall, the experimental setup for the GA applied to the Knapsack Problem involves initializing the population, iteratively performing selection, crossover, and mutation operations, evaluating the fitness of individuals based on the total value and weight of selected items, and generating a new population until the termination condition is met. The best individual in the final population represents the solution to the knapsack problem.

ANT COLONY OPTIMIZATION

Experimental Setup for Ant Colony Optimization (ACO) applied to Knapsack Problem:

Entities:

1. Ant: Represents an individual in the ant colony and constructs a solution for the knapsack problem.

2. Solution: A binary array indicating item inclusion/exclusion in the knapsack, constructed by an ant.

3. Pheromone Trails: Represents the intensity of pheromone on each item, guiding ants in the decision-making process.

4. Heuristic Information: Provides guidance to ants based on the value-to-weight ratio of each item.

5. Fitness: Evaluates the quality of a solution based on the total value and weight of selected items in the knapsack.

Components:

1. Initial Pheromone Trail and Parameter Initialization: Set up the initial pheromone trails and heuristic information based on item values and weights.

2. Construct Solution: Ants construct solutions by selecting items based on the probabilities derived from pheromone trails and heuristic information while considering the weight constraint.

3. Evaluate Solution: Calculate the fitness of a solution by summing the values and weights of the selected items, ensuring they do not exceed the maximum weight.

4. Update Local Pheromone: Update the pheromone trails locally based on the quality of the solution found by each ant.

5. Update Global Pheromone: Deposit a higher amount of pheromone on items selected in the best solution, promoting exploitation of promising paths.

6. Termination Criteria: Determine when the algorithm should stop iterating (e.g., maximum number of iterations).

7. Best Solution Selection: Track the best solution found during the iterations.

Flow Control:

1. Initialize the knapsack problem instance with the weight and value arrays, as well as the maximum weight constraint.

2. Initialize the pheromone trails and parameters, such as the pheromone evaporation rate, pheromone intensity, alpha, beta, and Q.

3. Repeat the following steps until the termination condition is met:

   a. Construct solutions for each ant by selecting items based on pheromone trails and heuristic information.

b. Evaluate the fitness of each solution, considering the total value and weight of selected items.

c. Update the local pheromone trails based on the quality of the solution found by each ant.

d. Track the best solution found so far.

e. Update the global pheromone trails, depositing a higher amount on items selected in the best solution.

f. Check the termination condition (e.g., maximum number of iterations reached).

4. Select the best solution found as the output.

Overall, the experimental setup for ACO applied to the Knapsack Problem involves initializing the pheromone trails and parameters, constructing solutions for ants by selecting items based on pheromone trails and heuristic information, evaluating the fitness of solutions, updating the pheromone trails based on local and global information, and selecting the best solution found. The algorithm iterates until the termination condition is met, and the best solution represents the solution to the knapsack problem.

A TABLE PRESENTING THE RESULTS

File name	Optimal	GA	GA %	ACO	ACO%	GA run	ACO run
f1_l-d_kp_10_269	295	295	100%	295	100%	1	2
f2_l-d_kp_20_878	1024	1024	100%	1024	100%	1	3
f3_l-d_kp_4_20	35	35	100%	35	100%	0	0
f4_l-d_kp_4_11	23	23	100%	23	100%	0	0
f5_l-d_kp_15_375	481.0694	481.0694	100%	481.0694	100%	1	2
f6_l-d_kp_10_60	52	52	100%	52	100%	0	1
f7_l-d_kp_7_50	107	107	100%	107	100%	0	1
knapPI_1_100_1000_1	9147	9147	100%	8373	92%	6	13
f8_l-d_kp_23_10000	9767	9767	100%	9767	100%	1	4
f9_l-d_kp_5_80	130	130	100%	130	100%	0	0
f10_l-d_kp_20_879	1025	1025	100%	1025	100%	1	3

STATISTICAL ANALYSIS OF DIFFERENCES IN PERFORMANCE NEED TO BE PRESENTED

First let's focus on the optimal rate, which measure how close the solutions generated by each algorithm are to the optimal solution. Looking at the table we see that for all given 11 problem instances, both the GA and the ACO were able to find or come rather close to the optimal solution. However, the GA had 100% success rate and the ACO has an average of 96% as the success rate. There isn't any significant difference between the two algorithms in terms of their ability to find the optimal solution. We can use a paired t-test in order to justify the claim.

Null Hypothesis is that there is no difference in mean performances of the two algorithms for the given instances.

The alternative hypothesis is that there is a difference in mean performances of the two algorithms.

We are going to make use of the percentage improvement of the ACO algorithm over the GA.

Instance	% improvements of ACO over GA
f1_l-d_kp_10_269	0%
f2_l-d_kp_20_878	0%
f3_l-d_kp_4_20	0%
f4_l-d_kp_4_11	0%
f5_l-d_kp_15_375	0%
f6_l-d_kp_10_60	0%
f7_l-d_kp_7_50	0%
knapPI_1_100_1000_1	-8%
f8_l-d_kp_23_10000	0%
f9_l-d_kp_5_80	0%
f10_l-d_kp_20_879	0%

Mean = -0.8%

Standard deviation = 3.9%

Test statistic: t = (mean difference - hypothesized mean difference) / (standard deviation / sqrt(n))

t = (-0.008 - 0) / (0.039 / sqrt(11)) = -1.78

Using a two-tailed t-test with 10 degrees of freedom (n-1), we find that the p-value is 0.1. Since the p-value is greater than the significance level of 0.05, we cannot reject the null hypothesis. Therefore, we cannot conclude that there is a significant difference in the mean performances of the GA and ACO algorithms on the given instances of the knapsack problem.

Secondly, let's look at the run time performance of each algorithm. The GA was able to solve all 11 instances of the problem in a total of 9 seconds, and the ACO in a total of 26 seconds. This indicates that the GA is faster than the ACO in solving these particular problem instances.

GA mean runtime: (1 + 1 + 0 + 0 + 1 + 0 + 0 + 6 + 1 + 1) / 11 = 0.7

GA standard deviation: 1.46
The ACO mean runtime is: (2 + 3 + 0 + 0 + 2 + 1 + 1 + 13 + 4 + 3 + 3) / 11 = 3.09

The ACO standard deviation is: sqrt((1/10)*((2-3.09)^2 + (3-3.09)^2 + (0-3.09)^2 + (0-3.09)^2 + (2-3.09)^2 + (1-3.09)^2 + (1-3.09)^2 + (13-3.09)^2 + (4-3.09)^2 + (3-3.09)^2 + (3-3.09)^2)) = 4.46

Using the same formula, the t-value is now:

t = (GA mean - ACO mean) / sqrt((GA standard deviation^2 / n) + (ACO standard deviation^2 / n))

where n is the sample size (which is 11 in this case).

t = (0.7 - 3.09) / sqrt((1.46^2 / 11) + (4.46^2 / 11)) = -2.73

Using a t-table with 10 degrees of freedom, the critical value for a two-tailed test at a significance level of 0.05 is 2.228.

Since the calculated t-value (-2.73) is less than the critical value (-2.228), we can reject the null hypothesis and conclude that there is a significant difference between the mean runtimes of the two algorithms.

Therefore, we can conclude that the ACO algorithm has a significantly higher mean runtime than the GA algorithm.

A CRITICAL ANALYSIS OF THE RESULTS

The performance of both algorithms is evaluated based on the percentage of optimality achieved compared to the optimal solutions provided, as well as the time in seconds that is required for the algorithm to obtain the best result.

Overall, the results show that both the GA and ACO were able to achieve 100% optimality on all instances of the knapsack problem. However, it is worth noting that the ACO algorithm was not able to obtain the optimal solution for the one instance (knapPI_1_100_1000_1) and achieved a 92% optimality. In contrast, GA was able to obtain the optimal solution in all instances.

The GA required less time (in seconds) compared to ACO in order to obtain an optimal solution in most instances. However, in some instances, such as knapPI_1_100_1000_1 and f8_l-d_kp_23_10000, which are larger instances of the knapsack problem, the ACO algorithm took longer in comparison to the GA. GA can be applied to large-scale problems due to its parallel nature, while ACO is not scalable to large problems because the number of pheromones required to guide the search increases exponentially with problem size

GA is an exploration-based algorithm that uses a population-based approach to generate new candidate solutions while ACO is an exploitation-based algorithm that uses a pheromone trail to guide its search towards the optimal solution. GA requires careful tuning of its parameters such as population size, crossover, mutation rate, and selection operators to achieve optimal performance. ACO also requires parameter tuning, but the parameters easier to tune.

In conclusion, both GA and ACO algorithms were able to achieve optimal solutions in all instances of the knapsack problem. However, GA was able to achieve the optimal solution in lesser time in most instances. This could be due to the fact that we have 500 ants that need to explore solutions, in contrast with the much smaller initial population of 200 for the genetic algorithm. The results also suggest that the ACO algorithm might not be as robust as the GA, as it was not able to converge to the optimal solution as quickly as the GA. GA is more robust than ACO because it can handle noise in the fitness function and can still converge to a good solution. ACO, however, is sensitive to noise, and a small change in the fitness function can lead to a completely different optimal solution.