TAWANDA JIMU

COS 314 Assignment 2

U20494166

BACKGROUND

The Knapsack problem is a popular optimization challenge which is similar to the 1-dimensional bin problem. It involves filling a knapsack with a collection of items, each with a weight and value, while keeping the total weight of the items in the knapsack within a given limit and maximizing the total value of the selected items. The problem can be defined as selecting a subset of n items with weight wi and value vi that fit into a knapsack of capacity W. The Knapsack problem is categorized as NP-hard, which implies that there is no polynomial-time algorithm that can solve it optimally for all scenarios. However, there are various efficient algorithms and meta-heuristics that can provide acceptable approximate solutions. The Knapsack problem has numerous practical applications, such as in logistics, finance, scheduling, and resource allocation.

In Assignment 2 we I'm going to campare the effectiveness of applying meta-heuristics to solve instances of the knapsack problem name a Genetic Algorithm and Ant Colony Optimisation.I developed a GA and ACO algorithm to solve the given 12 problem instances

SYSTEM ENVIROMENT:

The performance of a search algorithm can depend on a variety of factors, including the hardware and software environment of the computer on which it is executed.This assignment was completed on the following system:

OS Name Microsoft Windows 10 Home Single Language
System Model HP Pavilion Aero Laptop 13-be0xxx
System Type x64-based PC
Processor AMD Ryzen 7 5800U with Radeon Graphics, 1901 Mhz, 8 Core(s), 16 Logical Processor(s)
Installed Physical Memory (RAM) 8,00 GB
Physical Disk Type Solid State Drive 512.00 GB

Algorithm 1: Genetic Algorithm

I implemented a genetic algorithm based on the structure described in the "Search and Metaheuristics.pdf" document provided by Dr. Thambo Nyathi (referenced in the resources folder of my submission).

To begin, I imported all the problem instances into my program using the Java libraries *java.io.BufferedReader* and *java.io.FileReader*. I stored the contents of the problem instances in a 2D array named *items*, which holds the weights and values for each problem instance.

The first step of the genetic algorithm is to randomly generate an initial population of individuals. The genes of each individual are represented using a binary encoding scheme, as it provides a natural and intuitive representation for the knapsack problem. Each gene corresponds to an item, and its value (0 or 1) represents whether the item is included in the knapsack. This binary representation aligns well with the binary nature of the problem, where items are either chosen or not chosen.

The size of the population is determined by the population size parameter, which is obtained from the problem instance file (the first number in the dataset represents the number of objects). The *generateInitialPopulation()* function in the *GeneticAlgorithm.java* file handles the initial population generation.

After generating the initial population, I evaluated the fitness of each individual. The fitness represents how well an individual solves the problem, and in the case of the knapsack problem, it is calculated as the total value of the items included in the knapsack. The *evaluateFitness()* function in the *GeneticAlgorithm.java* file handles the fitness evaluation. For each selected item (where *genes[i]* is *true*), the value of the item at index *i* is added to the total value, and the weight of the item at index *i* is added to the total weight. If the weight of the selected items exceeds the knapsack's capacity, the fitness is set to 0 to penalize such solutions.

I set the termination criteria for the genetic algorithm to be a maximum number of generations, and I chose 750 as the maximum number of generations. This range allows the genetic algorithm to perform a sufficient number of iterations to explore the search space adequately.

During each iteration of the genetic algorithm, the following steps are performed:

1. *Selection*: Fitter individuals are selected using the tournament selection method. In tournament selection, a fixed number of individuals (in my case, 2) are randomly selected from the population, and the individual with the best fitness among the selected individuals is chosen. The *tournamentSelection()* method is called, passing the population as a parameter. Within the method, an initial best individual is set to *null*. Then, in each iteration, a random individual is selected from the population using *random.nextInt(population.size())*. If the selected individual has a higher fitness value than the current best individual, the selected individual becomes the new best individual. After the iterations, the best individual from the tournament (the one with the highest fitness value) is returned as the selected individual.

2. *Crossover*: Crossover, also known as recombination, is used as the genetic operator to generate new offspring by combining genetic material from parent individuals. The *crossover()* method is called, passing two parent individuals (*parent1* and *parent2*) as parameters. Within the *crossover()* method, arrays of boolean values representing the genes of the parent individuals are accessed (*parent1Genes* and *parent2Genes*). A random crossover point is determined using *random.nextInt(NUM_ITEMS)*, where *NUM_ITEMS* is the total number of items. A child array is initially set to the genes of *parent1* up to the crossover point. Then, the remaining genes are copied from *parent2* into the child array. The resulting child individual is returned.

3. *Mutation*: Mutation introduces random changes in an individual's genetic material to maintain diversity in the population and prevent premature convergence. The *mutate()* method is called, passing an individual (*child*) as a parameter. Within the *mutate()* method, each gene of the individual is iterated. For each gene, a random number *mutationRate* is generated using *random.nextDouble()*. If *mutationRate* is less than the specified mutation rate (0.01), the gene is flipped (from *true* to *false* or vice versa) using the *!* operator. The mutated individual is then returned.

4. *Population Replacement*: In this step, the new offspring replaces a certain number of individuals in the current population. I used a generational replacement strategy, where the entire population is replaced by the offspring. The *replacePopulation()* method is called, passing the offspring population as a parameter. Within the method, the current population is cleared, and the offspring individuals are added to the population.

The genetic algorithm iterates through these steps until the termination condition is met. In my implementation, the termination condition is reached when the maximum number of generations (750) is reached. At that point, the best individual (the one with the highest fitness value) is returned as the output.

Overall, the genetic algorithm based on the provided document and implemented in my program demonstrates a search and metaheuristic approach to solving the knapsack problem. The algorithm iteratively evolves a population of individuals by applying selection, crossover, and mutation operations to search for better solutions. The termination condition ensures that the algorithm reaches a stopping point, and the best individual found during the process represents the solution to the problem.

Algorithm 2: Ant Colony Optimisation

I used the Structure of the ACO from the *Swarm Intelligence.pdf* which was provided by Dr. Thambo Nyathi on clickup. I have attached the file in the resources folder of my submission. For the ACO, I created a *ACOKnapsackSolver* class and I initialized all the parameters. The parameters are *capacity*, *items*, *numAnts*, *alpha*, *beta*, *evaporationRate*, *initialPheromone*, *maxIterations*, *noImprovementThreshold*. Here's a description of each parameter and its purpose:

1. *capacity*: Represents the maximum capacity of the knapsack. It determines the constraint on the total weight of the selected items.
2. *items*: This is a 2D array that represents the items in the problem. Each row corresponds to an item, and the columns represent the item's value and weight, respectively.
3. *numAnts*: Specifies the number of ants used in the ACO algorithm. Each ant constructs a solution by selecting items iteratively.
4. *alpha*: Determines the importance of the pheromone level when calculating item selection probabilities. A higher value gives more weight to pheromone levels.
5. *beta*: Controls the importance of the heuristic information (item value and weight) in the item selection process. A higher value emphasizes the importance of the item's value or weight.
6. *evaporationRate*: Specifies the rate at which pheromone levels evaporate after each iteration. It influences the exploration and exploitation trade-off of the algorithm. A higher rate leads to faster pheromone decay.
7. *initialPheromone*: Determines the initial pheromone level on each edge. It affects the exploration of the search space. A higher value encourages more exploration.
8. *maxIterations*: Defines the maximum number of iterations for the algorithm. Once this limit is reached, the algorithm terminates.
9. *noImprovementThreshold*: Represents the number of iterations without improvement required to trigger termination. If the best solution does not improve for this number of iterations, the algorithm terminates early.

The termination criteria is based on the *noImprovementThreshold* parameter. I set the *noImprovementThreshold* to a constant of 50 and the *maxIterations* to 500 due to the relatively small number of items in most of the given *problemInstances*. If the algorithm fails to find an improved solution for a certain number of iterations, the algorithm terminates early.

In the *solve()* method, there is a loop that iterates *maxIterations* times or until the termination criterion is met. Within each iteration, the best solution found by the ants is compared to the current best solution. If the best solution found in the current iteration is better (higher total value) than the current best solution, the *bestValue* and *bestSolution* variables are updated. However, if the best solution remains the same for *noImprovementThreshold* consecutive iterations, the

noImprovementCounter is incremented. If the *noImprovementCounter* reaches or exceeds the *noImprovementThreshold*, the loop is terminated, and the algorithm stops executing further iterations. This termination criterion indicates that the algorithm has reached a point where it is not finding any further improvements and is likely converged or stuck in a suboptimal solution.

Whilst the termination condition is not met I:

1. Constructed ant solutions. Ant solutions are constructed in the *constructSolution()* method. This method constructs a single ant solution by iteratively selecting items to include in the knapsack until the capacity constraint is reached or no more items can be added. Here's an overview of how the ant solutions are constructed in the code: a. Create an empty solution list to store the indices of selected items. b. Initialize a boolean array called *visited* to keep track of visited items. Initially, all items are marked as not visited. c. Set the initial *remainingCapacity* to the total capacity of the knapsack. d. Enter a loop that continues until the capacity constraint is reached or no more items can be added. e. Inside the loop, call the *selectNextItem()* method to select the next item to add to the knapsack. This method uses a probabilistic rule based on pheromone values and item characteristics to make the selection. If no more items can be added (i.e., *selectNextItem()* returns -1), break out of the loop. f. Add the selected item index to the solution list. g. Mark the selected item as visited by setting the corresponding index in the *visited* array to true. h. Update the *remainingCapacity* by subtracting the weight of the selected item. i. Repeat steps e to h until the capacity constraint is reached or no more items can be added. j. Return the solution list as the constructed ant solution.
2. My solution doesn't use a local search step after constructing the ant solutions. The algorithm focuses on the construction of ant solutions using the Ant Colony Optimization (ACO) framework but does not incorporate a local search phase. I got this from the *Swarm Intelligence.pdf* provided by the lecturer. I added the document for reference in my submission.
3. After constructing the ant solutions, I update the pheromones. The pheromone update is performed in the *updatePheromones()* method. This method updates the pheromone values based on the ant solutions generated in each iteration. Here's an overview of how the pheromones are updated in the code: a. Initialize the pheromone values by evaporating a certain percentage of the existing pheromones. This is done by multiplying each pheromone value by the *evaporationRate*. b. Iterate over each ant solution in the *antSolutions* list. c. Calculate the value of the current ant solution using the *calculateTotalValue()* method. This represents the total value of the selected items in the knapsack for the current ant solution. d. For each item in the ant solution, increase the pheromone value on the corresponding position in the *pheromones* matrix. This is done by adding *1.0 / solutionValue* to the pheromone value of the item. The *solutionValue* represents the total value of the current ant solution calculated in the previous step.

COMPARATIVE ANALYSIS

Problem Set	Algorithm	Best Solution	Known Optimum	Elapsed Time ns
f1_l-d_kp_10_269	ACO	252	295	24715800
	GA	294		4329900
f2_l-d_kp_20_878	ACO	972	1024	100324100
	GA	963		1405300
f3_l-d_kp_4_20	ACO	35	35	3062300
	GA	35		961300
f4_l-d_kp_4_11	ACO	23	23	2875500
	GA	23		995200
f5_l-d_kp_15_375	ACO	359.999389	481.07	28326500
	GA	419.083223		1332300
f6_l-d_kp_10_60	ACO	52.0	52	12874400
	GA	52.0		2258800
f7_l-d_kp_7_50	ACO	105.0	107	7479000
	GA	107.0		1534100
f8_l-d_kp_23_10000	ACO	9752.0	9767	69567300
	GA	9741.0		1924600
f9_l-d_kp_5_80	ACO	130.0	130	5141600
	GA	130.0		1712100
f10_l-d_kp_20_879	ACO	967.0	1025	96389300
	GA	935.0		1643000
knapPI_1_100_1000_1	ACO	8631	9147	27518300
	GA	8762		3414300

From the problem set and the results shown, it is possible to make a comparative analysis of GA and ACO on the Knapsack problem. Here are some observations:

- GA tends to find solutions more quickly than ACO. In all instances, GA achieved a solution faster than ACO. For example, for problem instance f1_l-d_kp_10_269, GA found a solution in 4.3 ms, while ACO took 24.7 ms to find a solution.
- ACO appears to be more accurate than GA in some instances. In instances where the optimum is known, ACO found the optimal solution for problems f3_l-d_kp_4_20, f4_l-d_kp_4_11, and f9_l-d_kp_5_80, while GA found the optimal solution for f3_l-d_kp_4_20 and f4_l-d_kp_4_11.
- For some problem instances, the difference in performance between ACO and GA is negligible. For example, both algorithms found the optimal solution for f3_l-d_kp_4_20 and f4_l-d_kp_4_11.
- GA tends to outperform ACO in larger problem instances. For example, for problem instance f8_l-d_kp_23_10000, GA took 1.9 ms to find a solution, while ACO took 69.6 ms.

- ACO appears to have better scalability in some instances. For example, for problem instance knapPI_1_100_1000_1, ACO found a solution in 27.5 ms, while GA took 3.4 ms. However, it should be noted that the difference in performance is not significant.

Overall, the choice between ACO and GA would depend on the specific problem instance and the trade-offs between solution quality and computational resources. If a quick solution is required, GA would be a good choice, while ACO may be more appropriate when accuracy is more important than speed. In larger problem instances, GA is likely to outperform ACO, while ACO may be more scalable for smaller instances.