COS 314 Assignment 2 Report

Algorithm Configurations

All the algorithms are coded in Java and use custom classes which I have created to handle the logic of the knapsack algorithm.

There are two classes:

- **Item** - which contains the weight of the item and the value of the item. This can be summarized in a tuple pair of two Doubles
- **Sack** – which contains a max weight to represent how much the knapsack can carry, as well as having an ArrayList of Items to carry, which is managed by internal functions in order make sure the sack cannot carry too many items beyond its maximum weight
    - This sack can be summarized into a Boolean array of the same size as the dataset of items which it is meant to collect (false meaning that it is not in the sack and true meaning it is in the sack) – this Boolean array will be the state of the Sack and can be retrieved using **getState(ArrayList<Item> dataset)**

There are other functions which would manage the sack and items, as well as functions to run the GA and ACO algorithms.

- **Sack.GetStateValue(boolean[] st, ArrayList<Item> dataset, double maxWeight):double** – takes a state of a set and tries to get the total value of all items in that set (if it is valid). If the set is not valid, this will return 0
- **Sack.IsValidState(boolean[] st, ArrayList<Item> dataset, double maxWeight):boolean** – returns true if the state is valid (as in the items do not cross the max weight of the sack)
- **StateToString(Boolean[] st):String** – returns a string of the Boolean array to be printable by the program
- **randomState(int size):boolean[]** – returns a random state of a sack of any dataset size, regardless of if it is valid or not. This is used with the algorithms to get a random starting point
- **booleanArrayToInt(boolean[] arr):int** – converts a state into an integer since a boolean array can be represented by an integer very easily. This is similar to converting a bitstring into an integer
- **intToBooleanArray(int num, int len):boolean[]** – converts an integer into a Boolean array used to represent the states. This can take in the length to get the length of the array to fill up any 0s or omit any extra bits not needed, however this is mainly determined by the dataset of items

Genetic Algorithm Config

The genetic algorithm in this case uses the state of the knapsack as the gene for each population entity, therefore each entity will be represented by just a Boolean array that is a state of that sack based on the dataset of items. The environment of the population is built to suit mutation of genes, which involves flipping the bits based on certain conditions and partially random chance to control when it can mutate, as well as having cross over reproduction to create new entities which will be evaluated before they are made to enter the rest of the population. Cross over reproduction involves splitting the array in half and swapping the halves to create two new arrays.

Evaluation of entities involves checking if the array is a valid state of the sack, and if not, the state will go through forced mutation of gene's bits by removing a random bit that is 1 (true) in a certain number of attempts. If the entity is not successful in those attempts, the entity will be removed from the population. This evaluation process is followed by the start of the algorithm, and when the children as well as main population is evaluated in every generation.

Note that the difference between mutation and evaluation here is that mutation will try to flip the bits of the gene to 1 in order to increase the fitness, while the evaluation is to attempt to fix the gene in order to make it valid by flipping some bits to 0 until it has been decided that it is too late.

The stopping condition of the algorithm is if the current generation has 0 entities or if the number of iterations set are reached. While it is tournament selection, the GA focuses on trying to grow the population and only selecting the fittest entity (based on the total value of the sack) without focusing on trying to kill off entities each generation.

The genetic algorithm has 4 main configurations apart from being able to accept the dataset and the weight of the knapsack, which are:

- **initPopulationSize : int** – allows the algorithm to create a set number of entities, each with a random gene to start off with before being evaluated in each generation
- **iterations : int** – the number of generations set to run for in case the population does not shrink
- **mutationRate : double** – the probability for running the mutation function on an entity in order to flip some of its bits to 1 in order to improve its fitness
- **crossOverRate : double** – the probability for running the cross over function on two entities to create new children to be evaluated later

This is all represented in the function **geneticAlgo(ArrayList<Item> dataset, double weight, int initPopulationSize, int iterations, double crossOverRate, double mutationRate) : Boolean[]**, where it returns the final result that is the most fit entity and the fitness can be extracted from that state.

The mutation and cross over functions are applied as follows:

- **mutate(Boolean[] st, ArrayList<Item> dataset, double weight):boolean[]** – takes in a gene as a state and checks if the state is valid before it tries to flip a random bit to 1 (true). If the state was invalid, it would try to remove the largest yet least valuable item from the array to keep the fitness as high as possible while validating the gene. This will return a new state which is the result of the mutation.

  If [0,0,1,1] is valid, it can become [0,0,1,1] → [0,1,1,1] as a mutation to improve the bit.
  If [0,0,1,1] is invalid, it can become [0,0,1,1] → [0,0,1,0] as a mutation to fix the bit.

- **crossover(Boolean[] stA, Boolean[] stB):Boolean[][]** – takes two genes as states, splitting them in half and swapping the halves in order to create two new arrays which represent the children of those genes, which is what it returns.

  For genes [0,1,1,0] and [0,0,1,1], they produce children [0,1,1,1] and [0,0,1,0]

For optimization purposes, which may affect the performance of the algorithm, there is a MAX_CAPACITY variable which caps the population size at any given point where the population of that generation exceeds that value. When this is done, the population is sorted by the highest value first and any entity in the population with an index higher than the MAX_CAPACITY will be taken out of the population. This is to optimize for keeping the program running without slowing it down and reaching a heap overflow, at the same time catering for how this will affect the performance of the results.

Ant Colony Optimization Config

This ACO algorithm makes use of Hashmap that maps states as keys, which are nodes in an arbitrary graph that has each state linked to another by having a bit flip i.e. the state [0,0,0,1] is linked to [1,0,0,1] since the first bit is flipped to 1; this also means it is linked to [0,0,0,0] since it has the last bit flipped to 0.

The hashmap uses the states as keys to map to values which as pheromone values which are used to show which state takes priority in how the ants should choose. The higher the pheromone value, the more likely the ant is going to go for that state.

HashMap<Integer, Double> map such that the key is the state changed to an integer for comparison purposes and the value is a double representing the pheromone value of that path.

The ants in this algorithm are represented by an ArrayList of Boolean arrays such that the ant will be represented by a state that tries to change over time based on whether it can change into a state that the map contains with a higher pheromone value, or the ant randomly travels until it reaches a dead end.

An example would be if there were ants with the state [0,0,1,1]. In the hashmap, there would be a key-value pair which is {3 : 20}, which 3 is the integer of [0,0,1,1] and shows that the state has a pheromone value of 20. If the ant moves from [0,0,1,1] to [0,1,1,1] and [0,1,1,1] has a higher pheromone value than 20, the ant will change its state to [0,1,1,1]. Otherwise, the ant will either try to flip another bit at random to compare to see if it can change to another state of higher value, or stay the same state and call that a dead end.

States that the ants are represented by will change their bits and evaluate to be a correct move if:

- the bit change causes the state to be valid and having a higher total value than the last state the ant was in, or
- the bit change causes the state to be a certain state in the hashmap that has a higher pheromone value than the last state it was in

The evaluation of the state in the ant is expected to make the state change, to show the ant moving in the graph. If the state does not change, the ant is assumed to have reached a dead end and the ant is removed from the ArrayList, as well as allowing for the state it was in to have its pheromone value increased by its value.

Every state the ants change to has its pheromone value incremented in the hashmap, however the state that becomes a dead end of the ant will have its pheromone value increased by the total value of the state in order for it to gain priority fairly. This allows it to gain a higher fitness than states which have lower values while not being too high to over take states that have been reached which are of obvious higher fitness based on their total value.

The algorithm runs in a loop, evaluating each state/node the ant is at in the graph until all ants have reached a dead end. The algorithm takes in a parameter of **colonySize:int** which determines the number of ants needed to find the optimum node to have a dead end at in the path from the initial state 0 which they all start from (regardless of the number of items in the dataset).

The algorithm is in the form of **antColonyOpt(ArrayList<Item> dataset, double weight, int colonySize) : Boolean[]**, to which at the end of running the algorithm it chooses the state in the hashmap with the highest pheromone value and returns it, which the fitness value can be determined from that state of the sack.

Miscellaneous

The program is built to take in inputs that allow for inputting parameters when running the code. This will also allow for the program to read through the folder "Knapsack Instances", in which it will read each file one-by-one in order to parse and create a dataset so it can apply the algorithms on it. It records timestamps with a Timestamp object and logs the results in a Result object which will output a set of configurations, along with the results of the algorithms. This will also create a folder that contains a log file to write the results into for checking further analysis, which was used in the experimental setup and analysis below.

Experimental Setup

This setup involves having to manipulate the population sizes and iterations for the GA, while also manipulating the colony size of the ACO algorithm as well for the same benefits. Since the GA and ACO are functions, they each have their own parameters that can be changed to suit the experiment to produce different results.

First set up uses assumed values to work based on initial testing while creating these algorithms to check if they work as expected, while also catering for the largest problem instances.

Second and third setups decrease and increase the values in the parameters respectively, while keeping the rates the same for the GA, so as to observe changes in the results on a linear scale.

The last setup is to benchmark the algorithms for performance, as well as testing for long processing based on the large problem instances in an attempt to find a better fitness that is near optimum.

Note that the configurations of the main in the program can be changed despite the fact that an experimental setup as been presented. You may try to use different values and even change the mutation rates and crossover rates to determine new fitness results. However, due to the nature of affecting probabilities for the rates, it causes observations to be inconsistent.

Setup	initPopulationSize (GA)	iterations (GA)	crossOverRate (GA)	mutationRate (GA)	colonySize (ACO)
1	500	500	0.5	0.5	10000
2	100	100	0.5	0.5	5000

3	250	250	0.5	0.5	7500
4	1000	1000	0.4	0.6	12500

Table of results

The algorithm has been run with the table of parameters in the previous section as a way to cater for the range of results that can arise and in order to reach the optimum values as much as they can in those different conditions.

Filename	**Opt**	**GA 1**	**GA 2**	**GA 3**	**GA 4**	**ACO 1**	**ACO 2**	**ACO 3**	**ACO 4**
f1_l-d_kp_10_269	295	295	295	295	295	161	241	295	201
f2_l-d_kp_20_878	1024	1024	1024	995	1004	830	892	935	977
f3_l-d_kp_4_20	35	35	35	35	35	28	35	35	26
f4_l-d_kp_4_11	23	23	23	23	23	18	18	19	19
f5_l-d_kp_15_375	481.0694	481.069368	481.069368	481.069368	481.069368	408.050783	347.02292	366.835536	365.16007
f6_l-d_kp_10_60	52	52	52	52	52	51	50	47	50
f7_l-d_kp_7_50	107	107	107	107	107	91	74	105	105
knapPI_1_100_1000_1	9147	2229	2229	2229	2229	0	0	0	0
f8_l-d_kp_23_10000	9767	9766	9753	9761	9760	9746	9742	9720	9726
f9_l-d_kp_5_80	130	130	130	130	130	130	118	118	130
f10_l-d_kp_20_879	1025	999	1010	985	1025	782	916	927	826

The runtimes have been logged and put by their mean average, while the best fitness for each algorithm has been logged side to side to compare with each other and the known optimum from the spreadsheet we have been provided.

Filename	**Known Opt**	**Best(GA)**	**Best(ACO)**	**Runtime (GA) in ms**	**Runtime (ACO) in ms**	**Diff GA**	**Diff ACO**
f1_l-d_kp_10_269	295	295	295	6801	9389	0	0
f2_l-d_kp_20_878	1024	1024	977	1129	382021	0	47
f3_l-d_kp_4_20	35	35	35	3528	271	0	0
f4_l-d_kp_4_11	23	23	19	3174	576	0	4
f5_l-d_kp_15_375	481.0694	481.069368	408.050783	3837	32002	0.0000319	73.018617
f6_l-d_kp_10_60	52	52	51	4244	3172	0	1
f7_l-d_kp_7_50	107	107	105	3080	1364	0	2
knapPI_1_100_1000_1	9147	2229	0	5284	131364	6918	9147
f8_l-d_kp_23_10000	9767	9766	9746	2688	89207	1	21
f9_l-d_kp_5_80	130	130	130	1071	218	0	0
f10_l-d_kp_20_879	1025	1025	927	5725	104622	0	98

Statistical Analysis

Based on the results of the algorithms, it can be observed there are large differences in the performances for both algorithms with these configurations given the problem instances and the settings from the program:

- Genetic Algorithm, on average, is much faster than the Ant Colony Optimisation Algorithm. The exceptions would be **f3, f4, f7** and **f9** especially, as they seem to show that the lower the number of items and fitness, the much faster the ACO algorithm can process
- GA always is closest to the known optimum than ACO and most of the time even reaches optimum in this case. Due to the configurations, the GA will have a better fitness for the more iterations and higher population it has, while ACO seems to need vary in settings based on the problem instance in order to come closer to the optimum.
- In some cases, especially in configuration 4, the ACO seems to need a lower colony size in order to gain better fitness, compared to the GA which needs to increase its population size in order to come closer to optimum
- The current experimental configurations are somewhat in effective to the knapPI instance, which the ACO algorithm was unable to gain a suitable result to have a fitness, while GA had reached a deadlock that caused it to only produce a consistent fitness of 2229. This may be due to how the configurations may not suit the complexity of the problem instance. Moreover, the random nature of both algorithms makes it much harder to find a valid state of the sack consistently.

Evaluation

Based on these observations, it can be assumed that the current configuration I have designed for the Genetic Algorithm seems to be more suitable for all problem instances compared to the ACO algorithm configurations.

It should be noted that while these results reflect on the performance of these algorithms in my configurations, I have not tested for adjusting the randomness of algorithms to influence the results. This is due to how, despite the random property being part of the algorithms, it may lead to inconsistent results which make it harder to identify patterns for in each algorithm.

It may be possible for the mutation and cross over rates to be adjusted to increase the fitness and produce an entity that may even reach the optimums of certain instances. The genetic operators can possible be adjusted as well to suit this such that it can produce results that suit these problem instances, however using the GA will need the operators to be as generalized as possible in order for them to be applied in various testing conditions.

In the ACO algorithm, the only heuristic involved that helps the search become more effective is the length of the dataset and the pheromones being updated to allow for the solutions for the ants to lean closer to a better solution. Use of the hash map allows for opportunity to compare local optima with potential global optima. However, as reflected from the results, the ACO algorithm could have used a better configuration or a certain number of ants and specific number of iterations in order to reach the optimum.