ADDIS ABABA INSTITUTE OF TECHNOLOGY

Artificial intelligence group Assignment

The knapsack and traveling salesperson problem

Report #1 Knapsack problem

Group members

1. Hayat Tofik ---------- UGR/ 2987/12
2. Paulos Dessie -------- UGR/6912/12
3. Tsega Yaekob -------- UGR/8485/12

Submitted to – Mr. Ammanuel

Contents

1. Genetic Algorithm to solve Knapsack problem

To solve the knapsack problem using the genetic algorithm we used to classes the first class is used to initialize constructors to specify the properties of the item. Which is name, size and value. The second class is the GeneticAlgorithm class and almost all the basic works are done here. Since we are writing an algorithm for the knapsack problem, we also initialized some variables which we are going to use later, in this class. These are the crossover threshold, mutation threshold, iterationLimit which are explained later and we assigned a value for each. Then we initialized a variable ‘sizeLimit’ which will specify the size that the bag can hold, an ‘items’ list to store the items that are going to be placed inside the bag and ‘best’ list to store added items name.

Then after we initialized everything, we needed we are ready to go. So first we defined a function “additem” that will simply add the items to this list but if the item is not initialized it raises an exception “not initialized”. So, it is based on this list that everything here below starts.

Then we defined a function “getRandom” so this function first defines a variable ‘tempSize’ which is initially zero and ‘randomSelection’ list to store the final selected items. Then using for loop, it tests if ‘tempSize’ is in the range of the items list length. If that condition is fulfilled, then an item is randomly selected from the list and stored in a variable called selection. Then if the sum of tempSize and the selection size is greater than the sizeLimit then it will break out of the loop and if that is not the case it will add the selection size to tempSize because we should let it know that we took an item and the size has increased so that it can’t be more than the sizeLimit as the loop continues. Then it appends the item which is stored in selection to the randomSelection list and finally the function returns that randomSelection list which stores the randomly selected items.

Then there is a function ‘getSize’ that is used to show size or weight of a combination of items. It takes one argument which is combination that shows the combination of the items which are selected. So, if an item is in the combination of the selected items, then we add the items size to a variable ‘size’ which we used to store how size of an item appears which is in the combination of selected items. Then the function returns this variable size. So, we can see that if an item is selected twice then the function tells us that the item is placed twice in the bag.

The ‘getpopulation’ function defines a specific range which is 16 and we used a loop where the loop continues until i is out of that range. So, the function will call the getRandom function to get the randomly selected items since the getRandom returns the items we selected randomly from the list. Then it stores those variables inside variable a and a is appended to a list called ‘temp’ and temp is again assigned to another variable bestCombo. This function returns nothing but the randomly selected items are found here which are stored in bestCombo.

The ‘betterSolution’ function basically compares the fitness or profit of two combinations of selected items. First, it takes two arguments which we combination1 and combination2, then by calling the ‘getFitness’ function it calculates or finds the profit which we called value of each combination and stores them in their respective variable combo1value and combo2value then using if conditional it compares the two values and returns the one that is the highest.

The ‘mutation’ function takes a single argument which is one combination of possible items. Here we are going to use the mutation threshold that we initialized earlier. So, it generates a random number in the range from 0.0 to 1.0 for each indexed item in the combination and if the generated number is less than the mutation threshold then that specific indexed item is changed with any random indexed item from the combination, it is basically swapping of that indexed item from the combination with another indexed item in that combination. Finally, the function returns combination that is created after mutation.

The ‘crossOver’ function takes two arguments combination1 and combination2 since cross over is done between two possible combinations. So, this function compares the length of the two combinations and uses the one that has less length. Then stores that length in a variable called ‘length’ which is initially zero. Then inside a loop, we generated numbers between 0.0 and 1.0 for each item in the combination, which is stored in a ‘crossoverChecker’. We are going to use the crossOverTreshold that we initialized at the beginning of the code. And if the generated number is less than the crossover threshold, then the indexed item of the first combination is crossed with the indexed item of the second combination and the indexed item of the second combination is crossed with the indexed item first combination. The loop continues for each item of the combinations, If the cross over is possible meaning the generated number is less than the crossover threshold. But if the generated number is greater than the crossover threshold then we won't perform a cross over. Finally, the new created off springs are returned after the possible crossover is done.

The 'BestCombiation' function takes one argument which is popList which is a list that we are going to use to store the items that we will pop later. The function first calculates the fitness of each item using the 'getFitness' function. Then each value is stored in a list called 'val'. We created a variable called 'li' that is a copy of popList and which is a temporary storage that can be manipulated the popList content is not affected. Then the function takes the one with maximum fitness from the 'val' list and that value's index is stored in a variable called 'index' then we take that index to find the value inside the 'li' list and append that value to another list called 'minPopulation' which stores the items fitness which are relatively larger.

Then if the function continues and looks for larger fitness or value, then it ends up finding the same value so to avoid this, the larger fitness that we found first needs to be popped and stored somewhere else for the function to continue and find the rest seven larger fitness's. So, it pops the first larger value from the fitness list 'val' and also from 'li' looks for the next larger value and pops it from the list 'val' and also from 'li' and appends the that relatively larger value to the minPopulation list and this process goes on and on until we find the 8 largest fitness value of 8 items. Finally, the minPopulation list that holds the 8 relatively larger fitness's of 8 items is returned.

The 'findOptimum' function takes no arguments. First, we used a loop that continues until variable i is in the range of 'iterationLimit' that we assigned a value of 100 at the beginning when initialized the lists and variables we needed. So, the function basically starts by taking 8 items from the bestCombo variable which we defined earlier in the 'getpopulation' function. And as it is described bestCombo stores 16 variables that are assumed to be randomly selected using getRandom function. So, after we took 8 items, each are defined as parents. Then by taking a pair of parents from the 8 items, each pair creates another pair by using the ‘crossOver’ function which are then stored in a consecutive parent variable. For example, we created parent9 and parent10 by recombining parent1 and parent2.

Then each new off springs are mutated using the function ‘mutate’. Then the bestCombo, which stores the 16 randomly selected items is updated with these 16 parent items (the first 8 parent items and the next 8 created by recombining and then mutating the first 8 items) that are supposed to give the best profit. Then by using the ‘BestCombiation’ function on the bestCombo list it will find the best combination. Finally, the function returns the best items, the fitness or value and the size.

Finally, we have a function called ‘GATester’. This function finally shows us the final output and it does that by running the code 10 times and calculates the average time, value and size of the best combination. After running the code, the result, we get is provided below in the table.

Final Output

No. of times it runs	Average Time	Average Value	Average Size
10 times	30.238630005624145	6342	60.95
15 times	56.56396000413224	9973	83.22
20 times	66.28185999579728	11917	93.84

as it can be seen from our final outputs, the time gradually increased as the number of items increased. We believe that this is due to the iterations in the algorithm. There are many iterations in the algorithm that use the total number of items and this has affected the average time taken for the specified number of items. The other major thing to note here is the average value found by the algorithm for each number of items. The average value for ten number of items was found to be 6342.

This is the same number that we got when this item were run in the simulated annealing algorithm. This means that both the simulated annealing algorithm and genetic algorithm work almost perfectly for 10 and less number of items. But when the number of items increase, the two algorithms start to show difference. The simulated algorithm was found to be better in accuracy than the geometric algorithm. There is no need to talk about the hill climbing algorithm, as both genetic and simulated annealing algorithms provide way better value. But, just because the value provided is better doesn't mean that the simulated annealing algorithm is better. It is important to note that, both algorithms are optimization algorithms, they don't give as the exact value but they make a repeated and improved guess. Which means we mainly use them to solve matters that are very costy to find a perfect answer for and finding something very close to the perfect answer is enough. So with that in mind, both of the algotihms give a fairly close output/value to the what is assumed to be a perfect value, but they have a major difference in the time they take to get this answers. Compared to the simulated algorithm, we have found that the Genetic algorithm is much much faster. As it can be reffered from the final output tables for both simulated annealing and genetic algorithm, the genetic algorithm on average was found to take upto 0.03 – 0.07 seconds for 10-20 items while the simulated annealing was found to take upto 2-7 seconds on average. This is a major gap in time that gets even wider and wider as the number of items increase. So, in conclusion from our outputs, we have come to understand that although simulated annealing is a little bit more accurate, the genetic algorithm is much faster and gives enough indication where the perfect value lays around and hence, is more famous than the simulated annealing.

2. Simulated annealing to solve Knapsack problem

To solve the knapsack problem using the simulated algorithm we used similar methods like the genetic algorithm except the core of the algorithm.

Here again we used two classes, the first class is used to define constructors that specify the properties of the bag or sack that will hold the items that are supposed to give maximum profit. The constructor includes name, the size and the value for the bag or the profit that we gain from placing the profit maximizing items or combination of items.

The second class is the class where all the work is done. First, we initialized some basic variables and lists like we did in the genetic algorithm. And here we created an items list to hold the items that we will place in the bag and best list to store items name.

Then we have a function called ‘additem’ that will basically add or append each item to the items list we initialized before. Then comes the ‘getValue’ function its work is basically the same as before.

The ‘getValue’ function takes one argument which is combination. Combination is the sequence of items that are assumed to give maximum profit. So, the function takes each item in the combination and adds their value to a variable called value which is initially zero. Finally, that value is returned that stores the total value or profit sum from a combination of list. Where the value is the profit of each item.

The ‘getRandom’ function creates or generates a combination where the items are selected randomly. First, we created a list called ‘randomSelection’ which we are going to use later to store the randomly selected items. To select the items, we first need to have a list where we will find the items that we are going to select to manipulate and they are found in the list ‘items’ that we created above. Then we added each item we find into this list. So, the work done is basically the same before. Using a loop until we reach the size of the bag or the bags holding capacity an item is selected from the items list randomly and stored in a variable ‘selection’.

Then we have tempSize variable that is used to hold the list size temporarily after we added an item. Then if the sum of tempSize and selection size or weight is more than the sizeLimit which is the holding capacity of the bag then it breaks out if that is not the case then we add the selected items size to the temporary size holding variable which is tempSize. Then that selected item is appended to the randomSelection list. Finally, we return the randomSelection list which holds the items which we selected randomly.

The ‘acceptanceProbability’ function takes three arguments a value, newvalue and temperature. So, basically this function comes in handy later in the ‘findOptimum’ function. But what this function does is compare the two arguments value and newvalue. First ‘value’ is the value or profit that we will gain from the first combination of random list of items and newvalue is the value of the new created combination of list of items. So, the function compares the two values and if the newvalue is greater than the original value the it returns 1 if not the basic calculation of simulated annealing is computed which is $e^{\text{(newvalue – value)/temperature.}}$

Then comes the ‘findOptimum’ function that finds the optimum value from a combination of items by repeatedly creating a combination and calculating their value. We first need the temperature that we initialized at the beginning as START_TEMPRATURE and assigned it to a value of ‘100’ then we also need a cooling rate which is calculated as a quotient of the cooling rate we initialized at the start first and the length of items. Here the function uses while loop and the condition that must be satisfied or the thing that allows the code below to be executed is for the temperature to greater than the END_TEMPERATURE. S, the function then creates two combinations of items using the ‘getRandom’ function which as explained earlier generates a combination of items that are selected randomly. After the combinations are created, their respective value is computed using ‘getValue’ as described earlier this function calculates the profit of a given combination of items.

Then as the loop goes on this best value is updated. And using the acceptanceProbability function we compare the accepting probability of the current and the next value given the temperature. Then after the comparison if the value which is greater than the two values (current and the next value) is greater than the randomly generated number from 0 to 1 the function sets the next combination as the current combination.

Then if the value of the current combination is greater than the best combination then we set the current combination as the best combination. And we decrease the temperature by the product of the temperature and the cooling rate. Then inside this function using a for loop we specify the name of the selected items, the value or profit we gain, the size or weight of the items in the combination. Finally, we return profit maximizing list of items, the value and its size.

Then the function 'SATester' This function finally shows us the final output and it does that by running the code 10, 15 and 20 times and calculates the average time, value and size of the best combination. After running the code, the result, we get is provided below in the table.

Final Output

No. of times it runs	Average Time(ms)	Average Value	Average Size
10 times	2364.03917999912322	6342	60.95
15 times	4649.0309299995934	10164.2	84.469
20 times	8451.678280002594	12178.8	93.161

This outputs were briefly discussed at the genetic algorithm's report(right below the final outputs of genetic algorithms) as it was easier for us to understand their differences and which one to use in certain cases. This algorithm's final outputs were found to be the biggest in value and hence the most optimal, but the time taken to find these results was seen as a little bit of a let down.

3. Hill Climbing to solve Knapsack problem

Since we are solving the same problem which is knapsack problem in all the three solutions some parts of the algorithm are the same. Hill climbing algorithm also shares some parts of the previous algorithms. Here again we used two classes in the first class we have a constructor that defines the properties of the bag that will be holding the items which at the end are supposed to give maximum profit. These are the name which specifies the name of the items that are added in the bag, the size which tells us the size of the bag and value is the profit gained from the combination of items stored in the bag.

The second class is HillClimbing class that defines every work. First, we initialized items, visited, current and bestCombo lists whose work are explained later. And we have sizeLimit variable that tells us the size of the bag or its capacity.

Then there is the ‘additem’ function adds items to a list called items. Its work is basically the same as how we used it in the above two algorithms it will simply add the items to this list but if the item is not initialized it raises an exception “not initialized”.

The ‘getValue’ function also does the same work which is taking each item in the combination and adds their value to a variable called value which is initially zero. Finally, that ‘value’ is returned that stores the total value or profit sum of a combination of items.

The ‘define Selector” function is basically used to create a biased selection. It used the value per weight of each item to determine the number of times a specific item is added to the domain list from which random items are selected from. This was, the items with higher values will have a higher chance of selection.

The ‘betterSolution’ function works the same way as its used above. First, by taking two arguments which we combination1 and combination2, then by calling the ‘getFitness’ function it calculates or finds the profit which we called value of each combination and stores them in their respective variable combo1value and combo2value then using if conditional it compares the two values and returns the one that is the highest.

Then comes the ‘getLocalMaxima’ function, which is the main function that coordinates all the other functions to find the local maxima. It initiates the starting combination to a random(but yet biased) selection of items. It then chooses the next combination randomly as each of the remaining combinations can come next to the current one. The values of these two combinations are compared using the function ‘better solution’ and if the new combination was found to be better thatn the current one, the current will be updated to the new combination and the process will repeat. Otherwise, the process halts and returns the current value.

The ‘hillTester’ function works the same way as GATester. This function finally shows us the final output and it does that by running the code 10 times and calculates the average time, value and size of the best combination. After running the code, the result, we get is provided below in the table.

Final Output

No. of times it runs	Average Time(ms)	Average Value	Average Size
10 times	0.17200999427586794	4781.5	60.95000000000001
15 times	0.2948800043668598	9481	62.82000000000001
20 times	0.3098200017120689	10356.3	88.493

From this output we can understand that hill climbing algorithm just returns the local maximum. Sometime the local maximum might be the global maximum but at other time it is way of it. That is why the average value for the hill climbing algorithm is way lower that those for simulated and genetic algorithms. But , on the contrary , the hill climbing algorithm is found to be relatively faster that the previous 2 as it takes less than 0.1 milli-seconds to find its maxima. It is important to note that this may not always be true if the randomly chosen starting combination of items has a very low value and the following combinations start to gradually increase in value. This means the current combination will keep on updating to the next and hence, taking more time. But in our case, since we didn't find a way to order the list of combination with different local maxima (the next is also a randomly generated combination), it usually halts very quickly making it faster than the previous 2 algorithms.

For different setting(hyper parameters)

We tested 2 of our algorithms with different setting.

Simulated annealing

we were using an initial value of 100 for temperature and it gradual decreased at a rate of 0.0002 until it reached a final temperature of one. we modified the starting temperature for the simulated annealing 4 times and tested the algorithm again to see the difference in results. A list of 20 items was used for all initial temperature values so that the only difference found could be dues to the initial temperature. The results found were as below:

Initial temprature	Average Time(ms)	Average Value	Average Size
10	2211.3600200100336	12046.9	92.95
5	1744.6810799883679	12020.4	92.72900000000001
1	0.20601999713107944	9001.9	85.55799999999999
0.1	0.18905000761151314	9817.8	89.367

As it can be seen from above result, decrease in initial temperature made the algorithm faster but at a cost of accuracy. The optimal value started to decrease as the temperature decreased. We believe this behavior was due to the limitation on the number of iteration caused by the decrease in the intial value. As initial value decreases the time it takes to reach its final value decreases as well limiting the the number of iteration, and hence the journey to a better result.

Genetic Algorithm

we initially used a population of size 16 to calculate the optimal result. We then tested this algorithm with different population size. We tested this algorithm with population size of 10, 100 and 500 the results were found to be as shown below:

Population size	Average Time(ms)	Average Value	Average Size
10	69.515	11834	94.30
100	262.77	12227	93.35
500	2200.76	12227	93.35

From this results, it can be clear seen that increasing the population size has increased the optimality of the result of the genetic algorithm but the time consumed to give the output increased at each increase of population. Which means both simulated annealing and genetic algorithm depend on the way they are implemented. If the initial temperature of a simulated annealing is too high or its cooling rate is too low it outputs an optimal result but the time taken increases dynamically as well. The same is true for using large sized populations in genetic algorithm. But we still think, that the genetic algorithm edges the simulated algorithm just a little bit as it provides a wholly evolved population rather than a single best and also the time taken still looks better than the simulated annealing although for large population size.