Report

Design Process

Approach 1:

In the development process I tried several Approaches to come up with the optimal maximum profit. My initial intuition was simply to Incorporate the profit into the graph, in the adjacency matrix: such that the cost of traveling between link would be subtracted from the profit at the destination address: graph[i, j] = Profit-delivering-item-at-destination(j) - direct-cost(i, j) and the reciprocal would be graph[j, i] = Profit-delivering-item-at-destination(i) - direct-cost(j, i). I would then use dijkstra's to find the maximum profit MST instead of the minimum cost MST. The dijkstra's Algorithm instead of finding the minimum cost node to include would instead include the maximum profit node and compute the Maximum cost spanning tree. The Dijkstra's Computation could start at the Maximum Profit Node and compute the paths from this Node to every other node and take the most Profitable route. When it arrives at the most Profitable destination would compute Dijkstra's again from this route to the next Profitable Route. It would keep traveling routes till it can deliver items and the maximum capacity isn't filled.

Limitations:

This approach does not provide the optimal results however because the Profit is always included for all the intermediary nodes even when no item is being delivered to these destinations. To handle this you could deliver at all items in the path this would however not find the most optimal route and would be delivering the items in the intermediary nodes instead of delivering at all the maximum Profit nodes. There is the issue of algorithm traveling back and forth at the same path, From source to maximum profit destination and from maximum profit destination to source. This issue can be solved by setting the Profit of every node you travel to 0 but then this would cause you to get the incorrect cost when traveling between routes, You can instead then subtract the profit from the travel cost of each link in the path you take, this would give you negative links. There were various Complications with the approach so this approach was ultimately abandoned.

Approach used:

Instead of the algorithm stated above I chose a simpler approach that uses two algorithms and keeps a logical separation between the subproblems of solving the larger problem of finding the optimal maximum profit. This approach uses knapsack to find the maximum profit items to include and uses dijkstra's to compute the minimum cost trip. The knapsack computes the most profitable items for the most profit through dynamic programming approach and builds upon subproblems that are interdependent these problems are stored in a table of items x weights. Each weight column is broken down into units of weight from 0 to

weight capacity and the rows represent the computed weight for each item included. The cost for each weight column is computed for each column when the item is considered for inclusion and then the optimal value for each item in each weight is computed dynamically using the equation: $V[i, w] = \max(V[i - 1, w], P_i + V[i - 1, w - w_i])$. The algorithm takes the maximum of two values: the optimal value in the previous row or the computed value by including the item: which looks into the previous row optimal value of the weight column subtracting the weight of the item being included plus the profit which would compute the optimal value for including the item. If including the item increases the optimal value for the weight column the item is included otherwise it is not. The last row of the last column gives the optimal value. To get the items included in the optimal value you must reverse engineer what items were included. You look inside the table starting from the optimal value in the last row and the last column and check if the value is the same in the previous row, if it is the same then this can be interpreted as the item was not included to cause the change however if the item optimal has a change that means the item was included and caused the change in value you then subtract the item weight from the weight column being considered and go to the column that comes before including the item and there you can repeat the process and see if in that weight column the optimal value is the same in the previous row and check if the item was included. Through reverse engineering the knapsack table you can check for inclusion and return a set of items included to compute the optimal value. This set of items can then be used for computing the trip routing since each item has an associated address.

I used dijkstra's to compute the minimum cost route between each destination to get a single source minimum path to another destination till we have no more destinations to reach. The dijkstra's algorithm also dynamically computes the minimum distance from source to each destination initializing each minimum distance at infinity and then computing the route minimum route by traveling to a minimum distance node and using this minimum distance node to compute shorter routes every other node. I used my own approach for implementing dijkstra's through a table, the table of nodes included to compute and a row of nodes for which the minimum distance is being computed. I came up with the approach by computing dijkstra's on paper and noting the steps performed to compute the minimum distances. I used the output to create a minimum spanning tree. This approach works but there was a much more efficient approach online that has a similar order of operations however has a lower space complexity. The table method that I was using had some errors with computing from a source node other than the starting node so I used the algorithm I found online at geekforgeek.com. This algorithm computes using a single minimum distance array for each node and a boolean array that stores if the node has been added in computing the algorithm and then the algorithm returns a minimum distances to each node from the single source. We can trace the path taken through both algorithms, since both store the parent node used to reach any particular node. I run the algorithm between each destination to always compute a minimum distance from one route to the next. I record the distance/cost for the whole trip and subtract the total-distance from the optimal-cost computed in knapsack to get the total profit.

Limitation:

The limitation to this approach is that the algorithm still computes an approximation of the optimal profit from the optimal delivery trip. Since the two problems of computing the best items to include and computing the route are logically separated, the algorithm doesn't completely guarantee an optimal route. The problem of computing the most optimal profit: the knapsack doesn't factor in the cost to generate the profit this can mean that the algorithm could potentially have included an item that may have a lower profit margin because the route taken to achieve the profit may be more costly and the item with a lower profit value and a lower cost value may have a comparative advantage and this item would be excluded in the algorithm. This is the limitation of the algorithm in achieving the goal of obtaining a maximum profit delivery trip. The Algorithm has a complexity of $O(N^2)$, the algorithm can be made more efficient by using come form of greedy approach or any other form of efficient programming techniques that can reduce or exclude subproblems that don't need to be computed.

Approach 3:

There was another approach that could also have been used is the traveling salesman problem(TSP). TSP however computes cyclical routes and where the algorithm must always stop at the starting node whereas for this problem the algorithm doesn't have to start at a specific starting node and end its trip at the same node where it started. TSP also doesn't guarantee the best outcome because it only computes the approximate values because there are no efficient algorithms that can solve this problem and this algorithm cannot be solved without using an exhaustive approach to the problem solving. The only that compute efficiently are approximate so this approach suffers from the same limitations as the ones previously mentioned.

Results:

For the five input problems of size 200 randomly generated input I have the cost for each item ranging from 10 - 190, the profit for which is 10% so the profit for each item ranges from 1 - 19, the weight for each item ranges from 1 - 9 and the knapsack maximum capacity ranges from 50 - 100, the direct cost between each address is between 1 - 10. The results computed from these random inputs is the following. the average max profit size = 459.6, the average cost from dijkstra's = 70.4, the average optimal value from knapsack = 530, the average order of operations = 1444164.2. From the results we see that there is a significant profit margin. The order of operation complexity for this algorithm is $O(N^2) + O(NC)$. My estimation of operations was 130,000.

Implementation:

The solution is implemented in java, the algorithm is broken down into classes and I modularized my code in order to have logical division between separate modules of the problem

solution. I first solved the problem for a small example problem for which I could verify the results and then I scaled to larger problems of size 200.

Suggestion:

The algorithms can be found at various open source forums and online source and their implementation can be found in various different programming languages. There is a site called rosetta code that has implementations of these common algorithms in almost every language, and in some cases many versions of the language and in many different approaches to solving the algorithmic problem.

For the borrowed code for dijkstra's algorithm from geekforgeek.com, my suggestion for the algorithm would be more modularity in the code so that the code is more readable. I would also reduce the if statement condition by using a getmin() function instead. Although I didn't borrow the knapsack problem from anywhere. I just used what was explained in class to construct the algorithm. My suggested improvement for computing the knapsack is to compute the algorithm using a single array instead of the table to reduce the space complexity. The new algorithm can be computed using this equation:

$$V[w] = \max(V[w], P_i + V[w - w_i])$$

References

https://www.geeksforgeeks.org/printing-paths-dijkstras-shortest-path-algorithm/