Professor Bhowmick
4110.002
Zachary Bordelon, Malena Reyes

Project (Due December 9 2019)

Problem Statement. Consider that you are working for a delivery service, where you can go to the warehouse and pick items to deliver. Each item i has a cost c_i and weight c_i, as well as the address to where to deliver. You have a limit W on how much weight your van can carry. For every item you deliver, you earn 10% of its cost

You are also provided with a map of the town with all the possible of addresses of the items in the warehouse, and the distances between them. Every mile you travel costs you $1.

For simplicity you can consider the items are packaged and cannot be divided further. Also there is a direct path between all pairs of addresses, although the distances may differ. Each address receives exactly one item.

1. Develop an algorithm, where given the list of items, their weights, costs and delivery ad- dresses, you can select the items to pick for 1trip, that brings you maximum profit. Clearly explain the algorithm in english (Step1, Step2,..etc) and demonstrate how it works for an example problem (20+20=40)

   The following algorithm is a combination of a Knapsack 0/1 problem and a dijkstras search for the shortest distance problem. The following algorithm uses them both as a knapsack bottom-down approach is taken to find the maximum value with a given weight constraint. As we are calculating the max value at a particular weight with a given list of items we have 2 choices. Either we do not use that item or we use the ith item and reap the benefits of M[i-1][currcol-currweight] rewards. In addition to looking at the max of M[i-1][currcol] and M[i-1][currcol-currweight]+ Mi, we must also calculate the distances between these paths from the warehouse.

```
Part1(M[][]){
   val =  M[i-1][currcol-currweight];
   If(M[i-1][currcol-currweight] == 0){
        Mile = dist[0][curr.address];

        return val-Mile;

    }else{
        Int I = row_we_are_in;

        While(M[(i-1)][currcol-currweight] != 0]){
            Add(vertex.M[i][currcol-currweight].address);
            //adding previous vertexes
            i--;
        }
        Mile = Dijkstras(Curr.address, (vertex.M[i+1][currcol-currweight].address))
        //shortest path from warehouse to all all of the added vertexes

        return val-Mile;
    }
}
```

```
Part2(M[][]){
    val = M[i-1][currcol-currweight]+ Mi;
    If(M[i-1][currcol-currweight]){
        Mile = dist[0][curr.address];
        return val-Mile;

    }else{
        Int I = row_we_are_in;

        While(M[(i-1)][currcol-currweight] != 0]){
            Add(vertex.M[i][currcol-currweight].address);
            //adding previous vertexes
            i--;
        }
        Mile = Dijkstras(Curr.address, (vertex.M[i+1][currcol-currweight].address))
        //shortest path from warehouse to all all of the added vertexes

        return val-Mile;
}



}
```

```
packageKnap(){

    if(currcoll ==0){
        M[i][currCol]=0
    }if(i ==0){
        M[i][currCol]=0
    }else if(currweight <= currcol){
        M[i][currcol]= Max(Part1(M[][]), Part2(M[][]);

    }

}
```

Items	Weights	Cost	address
ID1	3	30	A
ID2	1	20	B
ID3	2	20	C

	WH	A	B	C
WH	0	5	10	5
A	5	0	5	10
B	10	5	0	5
C	5	10	5	0

		0	1	2	3
0		0	0	0	0

1	ID1
	Value:30
	Weight:3
		0	0
			-cant
				0
				-cant
					25
					Max(M[1-1][3-1]+30=M[0][3]+30=0+30=30
					-mile:
					If(M[0][3] == 0){
					Mile =
					dist[0][ID1.address]=dist[0][1]=
					5
					30-5=25
					
					}
					
					Or
					M[1-1][3]=M[0][3]=0
					
					If(M[0][3] == 0){
					Mile =
					dist[0][ID1.address]=dist[0][1]=
					5
					0-0=0
					}
					
					) = 25

2	ID2
	Value:20
	Weight:1
		0	10
			
			Max(M[2-1][1-1]=M[1][0]=0+20=20
			-mile:
			If(M[1][0] == 0){
			Mile =
			dist[0][ID2.address]=dist[0][2]=
			10
			20-10=10
			}
			
			Or
			M[2-1][1]=M[1][1]=0) = 20
			
			If(M[1][0] == 0){
			Mile =
			dist[0][0]=dist[0][0]=
			0
			0-0=0
			}
			
			) = 10
			
			-
				10
				
				Max(M[2-1][2-1]+20=M[1][1]+20=0+20=20
				-mile:
				If(M[1][1] == 0){
				Mile =
				dist[0][ID2.address]=dist[0][2]=
				10
				20-10=10
				}
				
				Or
				M[2-1][2]=M[1][2]=0) = 20
				If(M[1][2] == 0){
				Mile =
				dist[0][0]=dist[0][0]=
				0
				0-0=0
				}
				
				) = 10
					25
					
					Max(M[2-1][3-1]+20=M[1][3]+20=0+20=20
					
					-mile:
					If(M[1][3] == 0){
					Mile =
					dist[0][ID3.address]=dist[0][3]=
					10
					20-10=10
					
					}else{
					Int I = row_we_are_in;
					
					While(M[(i-1)][currcol-currweight] != 0]){
					
					Add(vertex.M[i][3].address);
					//adding ID1 and ID2
					i--;
					}
					Dijkstras(ID3.address, (vertex.M[i+1][3].address))
					//shortest path from warehouse to all would be WH-A-B=10
					}
					
					Or
					M[2-1][3]=M[1][3]=25
					-mile:
					If(M[1][3] == 0){
					Mile =
					dist[0][ID3.address]=dist[0][3]=
					
					}else{
					Int I = row_we_are_in;
					
					While(M[(i-1)][currcol-currweight] != 0]){
					
					Add(vertex.M[i][3].address);
					//adding ID1 and ID2
					i--;
					}
					Dijkstras(ID3.address, (vertex.M[i+1][3].address))

					//shortest path from warehouse to all would be WH-A-B=10
					25-10=15
					max(15, M[i+1][3])
					}
					) = 25

3	ID3
	Value:20
	Weight:2
		0	20
			
			M[3-1][1]=M[2][1]=0) = 20
			
			-mile:adopt from top value 10
			20-10=10
			or
			
			-mile:do nothing
			0-0=0
				15
				
				Max(M[3-1][2-1]+20=M[2][1]+20=0+20=20
				
				-mile:
				If(M[2][3] == 0){
				Mile = dist[0][ID3.address]=dist[0][3]=
				10
				20-5=15
				}else{
				Int I = row_we_are_in;
				
				While(M[(i-1)][currcol-currweight] != 0]){
				
				Add(vertex.M[i][2].address);
				i--;
				}
				Dijkstras(ID3.address, (vertex.M[i+1][2].address))
				
				}
				
				Or
				M[3-1][2]=M[2][1]=20
				-mile:
				If(M[2][1] == 0){
				Mile = dist[0][ID3.address]=dist[0][3]=
				
				}else{
				Int I = row_we_are_in;
					20
					
					Max(M[3-1][2-1]+20=M[2][1]+20=10+20=30
					
					-mile:
					If(M[1][3] == 0){
					Mile = dist[0][ID3.address]=dist[0][3]=
					10
					
					}else{
					Int I = row_we_are_in;
					
					While(M[(i-1)][currcol-currweight] != 0]){
					
					Add(vertex.M[i][3].address);
					//adding ID2 and ID3
					i--;
					}
					Dijkstras(ID3.address, (vertex.M[i+1][3].address))
					//shortest path from warehouse to all would be WH-C-B=10
					30-10=20
					}
					
					Or
					M[3-1][1]= M[2][3]=25
					
					-mile:
					If(M[2][3] == 0){
					Mile = dist[0][ID3.address]=dist[0][3]=
					
					}else{

				While(M[(i-1)][currcol-currweight] != 0]){
				
				Add(vertex.M[i][2].address);
				//adding ID3 and ID2
				i--;
				}
				Dijkstras(ID3.address, (vertex.M[i+1][2].address))
				//shortest path from warehouse to all would be WH-C-B=10
				20-10=10
				}
				
				) = 15
					Int I = row_we_are_in;
					
					While(M[(i-1)][currcol-currweight] != 0]){
					
					Add(vertex.M[i][3].address);
					//adding ID3, ID2 and ID1
					i--;
					}
					Dijkstras(ID3.address, (vertex.M[i+1][3].address))
					//shortest path from warehouse to all would be WH-A-B-C=15
					25-15=10
					max(10, M[i+1][3])
					}
					
					) = 25

2. Implement the algorithm and try it on 5 different set of inputs. Each input consists of a list of items and a complete graph of addresses, along with their distance. There should be at least 200 items on the list and 200 addresses. Submit your code and your results (20+20=40)

```
Malenas-MacBook-Pro:4110Project malenareyes$ g++ -std=c++11 temp.cpp
Malenas-MacBook-Pro:4110Project malenareyes$ ./a.out
Which file of item input to map txt?
maptm.txt
Which file of item input to test?
tm.txt
What is the max weight, W, that the truck can carry?
3
Maximum profit is : 25Malenas-MacBook-Pro:4110Project malenareyes$ cat maptm.txt
WH,A,B,C
WH,0,5,10,5
A,5,0,5,10
B,10,5,0,5
C,5,10,5,0Malenas-MacBook-Pro:4110Project malenareyes$ cat tm.txt
ID1,3,30,A
ID2,1,20,B
ID3,2,20,CMalenas-MacBook-Pro:4110Project malenareyes$
```

3. Write a short (3 page 11point font report) on how you developed the algorithm, its usefulness, the results, discussion about the advantages and drawbacks of your method. Also discuss whether such algorithms are available publicly, and what future enhancements can be made. Include references as appropriate (20)

Initially when we designed the algorithm we began to break apart the problem into two sorts of algorithmic questions. One involving the 01 knapsack problem and the other a Dijkstras sorting algorithm. The reasoning behind this was simply because the dilemma of having to come up with a maximum profit with a constraint of different weight values presented itself, as the driver having to come up with the maximum value for a given set of items to put into his van with a given weight constraint. Considering this area alone we would have been able to simply multiply the values by 10% to receive the assortment of items that would result in the maximum value. However, since this was not the only constraint, we found the need for a distance sorting algorithm. With the driver being charged one dollar per a mile, we knew we needed to consider the cost of travel to subtract it from the possible max value for a more accurate reading of profitable value.

We initially generated a small test case set to identify different decisions made during a knapsack sort. We created a map table that was inclusive of the distances between items address points to the warehouse and to each other. This map created a simple square structure with diagonals twice the size of distances of its adjacent nodes. Doing this allowed us to focus in on potential edge cases which may have arisen with different types of situations. Having the diagonals twice the size of the adjacent, allowed us to test for the instance where a possible distance could be a determining factor for choosing a particular item over another. In other words an instance where the mileage of one item would be higher causing the profit to be lower due to the subtraction from the potential profit. This example is depicted in the algorithm section where we were in between items1 and item2 combo that had a total value profit of 40 and choosing item 1 with had a value of 30. The algorithm using only the 0/1 knapsack problem would have resulted in choosing item 1 and item 2. However, when we kept track of the traveling distances it was clear that the direction of travel that the driver were to take would make a difference in profit. For instance, if the driver were to travel from the warehouse to address B then C, then the cost of travel would have resulted in $15, with an overall profit of $25, which

would have been the same profit of just having to do one stop with A. Versus if we had went from the warehouse to address C then to address B we would have amounted to a mileage cost of $10 resulting with a profit of $30. As we will later discuss, this was also a point where we noticed incorporating an MST would have been constructive in deciphering between two similar outcomes.

We also went under the assumption that the driver was able to pack up and leave after their last stop of deliveries effectively ending the need to calculate more miles. In addition to the Map table we generated a small items table that included the items weight, cost, and address, with their associated item ID. Initially when we designed the algorithm, we had only taken into account the items to be delivered, the cost, and the weight per item.

This helped us identify the primary primary resource of the 0/1 Knapsack problem. In traversing down the knapsack matrix we realize that when making the choices between items via the knapsack algorithm that this would be the ideal place to compare the distances between two possible maximum values.Specifically we found that we needed to compare the value of the element's above value distance from the warehouse if it was the first stop in which that value was possible, or traverse back to its original first destination point. If in the event the path was inclusive of multiple items then these items addresses would be added to a graph to sort through using Dijkstra's algorithm. We would use this sorting algorithm to find the minimum path to reach to the item and all of its added nodes. In retrospect we believe that this particular instance might have been better implemented via a minimum spanning set algorithm. This could have been used to Identify multiple different paths between the nodes in question. Adding these possible MST sets we later then could have ran a Dijkstra's algorithm on the set of possible MST's, result in a more thorough analysis. However, for our test purposes analyzing initially with dijkstras would suffice it enough. We were able to use the path generated by dijkstras to then calculate the distance of the path to subtract from the potential values to have a better understanding of what the driver would actually be making with a particular set of items along with their weights and total route distance.

Another notable asset to the development phase was in the development of our test cases. In two particular cases we generated a random set of distances between certain ranges that formed a linear graph where we could expect only one single package to be deliverable with the given constraints. For one test case we implemented this deliverable package at the furthest distance from the warehouse. However, given that it was the only package that fell within the weight constraint of the van, it was expected to be the only package that could be delivered, also making it's cost the maximum value if and only if the profit subtracted from the distance was greater than not having had left the warehouse at all. The same type of test case was ran but with only package that fell within the constraint being closest to the

warehouse. This test case was helpful in testing a base case for the algorithm. It helped us identify that if the weight of the item exceeded the max limit we could further cut down our test set. This however was not able to be implemented do to time constraints. Instead the program we had still attempted to view all of the items rather then immediately omitting them from the test set in the knapsack problem. In the future we would enhance this to request the weight constraints prior to reading in the test files in order to sift through irrelevant data.

Resources that proved to be helpful for this algorithm were the 0-1 Knapsack Problem | DP-10 posted on the GeeksforGeeks website. This algorithm helped depict a good bottom up approach to base our algorithm off of. Another useful website was onlinerandomtools.com for generating random numbers and strings. It allowed us to create a random set of values between a set minimum and maximum values. This sped up creation of test files and allowed us to test a large number of variations for our algorithm. Resources for Dijkstra were also helpful and provided by Dijkstra's shortest path algorithm | Greedy Algo-7. This algorithm was useful in our design process as it helped us identify the shortest path between a given number of nodes to be searched through in the knapsack algorithm. The total path was summed together inorder to return. The return value could then be used to subtract from a discovered profit value. This would indicate the profit minus the cost of the milage. Also, resources involving vectors and maps were important. We used vectors for holding our item classes as well as for reading in the values from a comma separated values file. They were extremely useful structures due to their nature of being able to resize on their own. Maps were useful when constructing our addresses and their distances.

One of the drawbacks that occured when using both the knapsack and dijkstras algorithm was that when the algorithm ran against larger sets of data the recursion seemed to act volatile. We settled on the notion that due to memory utilizing both algorithms would not be effective on larger test cases but was more precise with smaller test cases. We generated another algorithm using only knapsack to compare the results and found that while the knapsack algorithm only worked on larger data sets, the profit calculation had a fair amount of true discrepancies. The knapsack only algorithm though was still able to get a fair enough reading though in regards to making an overall decision on item selections with our particular given test cases.

Finally the total cost could then be multiplied my .10 to represent the actual value of the profit. This being done towards the end seemed to be a more simpler task in order to keep the data we recorded consistent when traversing.

This algorithm would be extremely useful in real world applications. Companies like Uber would benefit as drivers can see which passengers would give them the most profit compared to the distance of the both the current location of the driver to

the current location of the passenger as well as the starting location to the drop off location. A variation for this situation might also take into account if the destination has more potential passengers in the area as well. This algorithm could also be used similarly to the initial problem, for any kind of shipping company. Companies like Amazon could use this algorithm to determine shipping routes to optimize both time and money for gas. In this instance, a variation of the algorithm might be used to prioritize packages with express shipping or Amazon Prime members over regular shipping or non-Prime users when determining which routes would be best.