Creating an Optimal Grocery List with Budget and Recommended Food Groups 7375 Artifical Intelligence

Andy Vu, Sam Hekman, and Tyler Sutherland

Abstract—The abstract goes here.

Index Terms—Computer Science, Artificial Intelligence, Knapsack, NP Complete, list generation,

1 Introduction

THE knapsack problem is a classic problem that has troubled mathematicians for well over a century [1]. It can be easily be described as given a set of items each with their own weight and value, one is to choose items that maximizes the total value while within the restraints of the knapsack's weight or other limitations. Depending on the application, it is easy to see how widespread this problem can be. Whether it be packing bags for a trip on an airline and attempting to not go over the weight limit, to a thief stealing the most valuable goods from a store and trying to make out with as much money as possible, this problem is difficult to solve.

In a general sense all knapsack problems are similar, there are quite some variations to the problem, although the classic 0-1 knapsack problem is probably the most common and popular. The 0-1 represents either an item being selected (1) or not being selected to put into a bag (0). This means that there is only one of each item and that there can be no more than one of that item. To an extension, there is the bounded knapsack problem where there are a certain amount of duplicates of items and in contrast there is the unbounded knapsack problem where there are no bound or limitations of items. In other words there are unlimited copies of each item available. In addition to these there is also a fractional knapsack. The fractional knapsack allows the ability to pick fraction of items instead of an item whole. An example would be selecting ½ of an item although in most scenarios this is not possible.

Since this problem is one that has troubled mathematicians for quite some time, there are a variety of conventional approaches to the problem. As can be imagined, increasing the variables at play allows more complexity to the problem. The problem itself is defined as NP complete. In a classic knapsack problem the time complexity is O(N*W) where N is the number of items available and W denotes the capacity of the knapsack. This time complexity is obtained using dynamic programming. Other conventional solutions such as brute force results in creating every permutation possible and then selecting the most optimal knapsack. This results in a time complexity of $O(2^n)$. As can be seen, increasing the number of items will cause more options to be selected and thus the number of possible combinations increases.The user now has more items that they may consider to be selected into the knapsack. On the other hand, increasing the allowances of the knapsack results in a similar siutation. By allowing a greater number of items into the knaspack, the problem again is an increase in the amount of items that are avaiable for selection due to this higher limit. Increasing these two variables even or adding a third variable will further cause the knapsack problem to be more complicated.

In this paper, we are approaching the knapsack problem from a different perspective than conventionally. Firstly, instead of a typical knapsack we are generating a grocery list with price being our constraint similar to weight. In addition to this, instead of value of an item, the grocery item will be weighed, thus more weight of groceries will coincide with a higher value. To add a layer of complexity, instead of generating a grocery list to maximize weight and budget constraints, there is another dimension to this problem: food group recommendations. The objective of this paper is to generate a grocery list from a conventional grocery store that satisfies the constraints of a typical knapsack however it adds another dimension. Food group recommendations is to ensure that the grocery list generated does not select only the most cost effective and weighty item but to ensure variety in the optimal list of objects. In other words, in this paper, we explore a multidimensional unbounded knapsack problem within a grocery store.

• *Department Computer Science, Kennesaw State University, Marietta, GA, 1100 South Marietta Pkwy SE 30060.*
git: https://github.com/vespenegas/Artificial_Intelligence_Project

March 28, 2022

March 28, 2022

2 Approach

For every knapsack problem there must be a list of selectable items with appropriate weights, and values associated. Initially, our attempt at creating a life like grocery store was to find an appropriate dataset free online. Looking through websites such as Kaggle and other databases, there were no suitable datasets available. Every dataset lacked food categorization. In our project we are attempting to create a grocery shopping list with one of the criteria being limitations by food groups. Thus, because of this, all of the available datasets did not contain a categorization of the six food groups. This expectation was quite low, but the conclusive evidence showed our predictions were correct after going through many datasets. Secondly another issue with datasets that grocery stores often used was that the items the stores carried were not limited to foods. Any walk into a grocery store would show items such as plates, utensils, cooking ware and so on. These items are not within the scope of our project. The datasets found often had tens of thousands of these items. The last issue with using a pre-created dataset was that often there are many overlapping products that only differ by brand and price thus also not within the scope of our project. An example would be white bread. There are many different brands of white bread and thus all of their prices differed depending on brand.

With these issues with available datasets at hand, we formulated our own dataset of grocery food items. For our custom dataset, we used the online shopping utility available on Kroger.com. From here, we sort by all departments shopping and selected grocery food items that are definable within food groups. This meaning, items that were complete meals or that combined food groups were omitted. Examples of this would be a can of ravioli as there are plentiful grains as well as meats within the item. We prioritized items that were singularly within one food group such as raw meat, vegetables, or grains. Other criteria that is selected is the price, and weight. Of course some items are priced by unit instead of by weight thus to solve this issue, a quick google search of the average weight was used. An example of this would be an apple that Kroger sells for one dollar per apple. We would google the average weight of an apple and add it to our database as price and weight. An example of our dataset can be seen below in table 1. After nearing 70 pages of Kroger's shopping database, and selecting certain items, as of now our database sits at 190 entries. Limitations of brand and size were also challenges that we faced. Entry in our database such as white bread, was selected as the generic Kroger brand white bread with its associated price and weight. All other forms of white bread and brands were omitted as to avoid confusion and complexity in our database. As time progresses and if there arises a necesity, the dataset will be refined with either the addition or subtraction of items.

Groceries			
Food	Category	Price(dollars)	Weight(lbs) #
chicken breast	meat	11.78	4.7
80% lean ground beef	meat	5.99	1
banana	fruit	.23	.41
strawberries	fruit	2.5	1
cucumber	veggie	.69	.75
large raw shrimp	meat	13.98	2
shredded cheddar cheese	dairy	2.29	.5
hot dog buns	grain	1.49	.6875
white bread	grain	2.75	1.25

TABLE 1
Custom dataset collected from Kroger showing food, food group, price and weight or calculated weight. This example table is limited to ten selected entires from the dataset.

3 Preliminary Results

To serve as a baseline, we have implemented a brute force algorithm that works on our dataset to select items that results in the heaviest shopping cart while staying under budget. From the runtimes, it is clear that brute force is extremely slow. We truncate our grocery dataset to collect runtimes with a budget of 100 dollars. Initially with a list length of 10 items, the run time is relatively quick at 0.0015 seconds. However, this exponential increase in time complexity is seen once the grocery list is increased. 20 items results in a run time of 1.57 seconds and then finally 28 items results in a runtime of 531 seconds. Any larger list of items were not included due to the time complexity clearly showing exponential growth. In addition to this, attempts at 30 item list length result in run times far too long and hence we decided to stop at 28 items. The data for these run times can be seen in table 2. In addition to this, the runtime chart and data can be seen plotted in figure 1.

Brute Force Run Time	
length of list	runtime (seconds)
10	0.0015
13	0.011
15	0.047
17	0.185
20	1.57
21	3.31
25	61.14
26	131.35
27	268.46
28	531.08

TABLE 2
Runtime data for brute force algorithm with a truncated dataset. Dataset with more than 28 items were resulting in extremely long runtimes.

4 Discussion and Future Works

As of the writing of this paper and this middle update, two base line conventional algorithms have been developed for the project: brute force and dynamic programming. These two methods are tried and true however their limitations in time complexity as well as being resource dependent show that their usage is limited. For datasets that are small as seen in the tables in the preliminary results section, they have satisfactory performance, however, for larger datasets they are not optimal and are not ideal. For this reason, we are currently implementing a heuristic and nonconventional algorithm.

Genetic algorithms: write about them here, using sources and citation

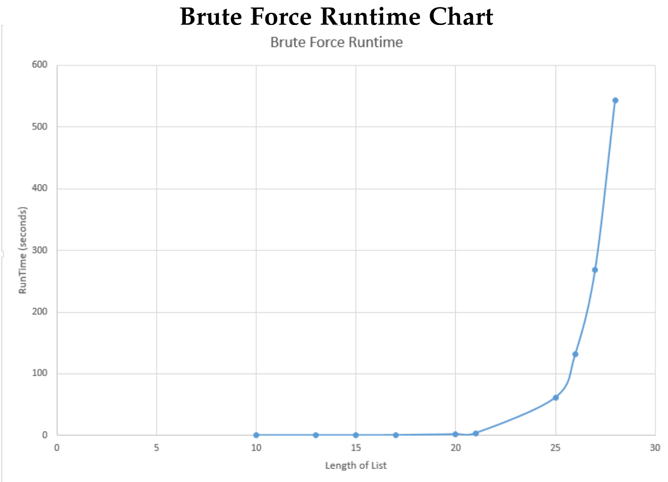

Fig. 1. Plotted runtime chart for the brute force algorithm

In addition to this, we are facing design issues with our project implementation. As can be seen in our brute force and dynamic programming we are utilizing these algorithms as 0-1 knapsacks without consideration of food groups which was a third dimension in our original design. The reason for this is that a conventional grocery store would behave as an unbounded knapsack. Of course there are not infinite amounts of inventory, however, the grocery store inventory is substantial enough that to a typical shopper, it would not be feasible to empty the stock of a certain item.

With this being said, designing the knapsack as unbounded would result in less choices that the algorithm would choose. It would simply pick an item that resulted in the maximum weight to price ratio and then select that item until the budget is met. An example would be selecting water melons which on average weight roughly 25 pounds. In the custom groceries dataset, watermelon is listed as 22 pounds with a price of 6.99. It would be too trivial for the algorithm to select watermelons until whatever budget is met thus resulting in the most efficient shopping cart. Introducing our assigned ratios of food groups would simply extend this issue. By forcing the algorithm to select a certain ratio of grains, meats, and dairy and so on, it would select items on a similar premise. For fruit, it would only select watermelons, for meats it would select an item that has the most efficient weight to price item as well. These would repeat until the budget is met and all food groups are to be accounted for. With that being said, there are some design choices that need to be made and reconsidered for the future of this endeavor.

5 Conclusion

The conclusion goes here.

how in text citation. use refrence.bib file asdf [1]

References

[1] G. B. Mathews, "On the Partition of Numbers," *Proceedings of the London Mathematical Society*, vol. s1-28, no. 1, pp. 486–490, Nov. 1896. [Online]. Available: http://doi.wiley.com/10.1112/plms/s1-28.1.486