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

Andy Vu, Sam Hekman, and Tyler Sutherland

Abstract—Developing an optimal grocery list based on budget and food group restraints causes the nature of the problem to identify as a multidimensional knapsack problem. The knapsack problem has been around for a longtime and still continues to cause mathematicians to struggle. There have been a multitude of conventional and well developed approaches to the knapsack problem as well as novel ideas. In this paper, we explore the prospects of solving the knapsack problem using a genetic algorithm. Although not entirely novel, we propose and explore the idea to introduce another dimension of food groups to add complexity to the knapsack. With this in mind we have devised a brute force and dynamic programming solutions to solve the knapsack problem alongside a custom generated dataset to serve as baselines to the genetic algorithm. The results show that conventional approaches such as brute force and dynamic programming work so long as the input size of the dataset is small but genetic algorithms work much more efficiently if there is a large enough dataset. Although not in the case of our implementation due to the small dataset our genetic algorithm was ten times slower than the dynamic programming but offered a much larger diversity in its output solutions. The idea of a multidimensional knapsack including food groups was found to be far too simple to even be bothered to implement and that datasets and grocery store design is much more simple and complex than perceived to be.

Index Terms—Computer Science, Artificial Intelligence, Knapsack, NP Complete, List generation, Dynamic programming, Bruteforce, Exhaustive search, Genetic algorithms, Heuristic algorithms

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 extent, there is the bounded knapsack problem where there are a certain number of duplicates of items and in contrast there is the unbounded knapsack problem where there are no bounds 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 a 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 situation. By allowing a greater number of items into the knapsack, the problem again is an increase in the amount of items that are available for selection due to this higher limit, thus increasing the dimensionality. 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 the value of an item, the grocery item

• *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

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 [2].

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 a 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 attempt 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 orange juice. There are many different brands of orange juice all with their own respective prices and various sizes.

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 select 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 are 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 189 entries. Limitations of brand and size were also challenges that we faced. Entries in our database such as white bread, were selected as the generic Kroger brand white bread with its associated price and weight. All other forms of white bread and brands were omitted to avoid confusion and complexity in our database as well as redundancy. As time progresses and if there arises a necessity, 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.

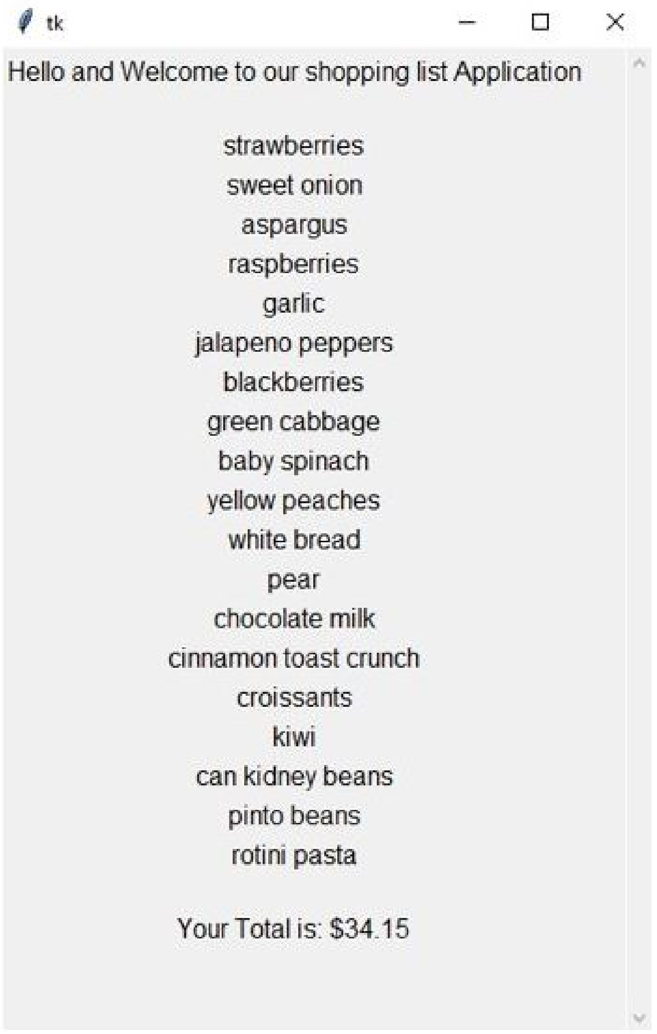
For baseline comparisons to the proposed model, we have implemented both a brute force method and a dynamic programming method to tackle the knapsack. As discussed further in the discussion sections, the current algorithms are set to work as conventional 0-1 knapsacks disregarding food group criteria. These two baseline algorithms serve to demonstrate the time complexity and usability in comparison to the genetic algorithm that will be implemented in the future. Both the brute force and dynamic programming methods take in the budget as a constraint, using weights as the value to maximize and consider items by their price. The brute force algorithm uses itertools combination method to create a list of all possible combinations within the budget constraint, and then chooses the maximum value and list that composes that combination.

Using dynamic programming for our algorithm, we intended to use a top-down memoization approach to compute subproblems for the price limit and weight maximization. This approach works by building a table based on the number of items in the dataset by the limit in integers. For example, if the dataset has 200 items and the price limit is 100, then the table will have a shape of 200X100. Each index stores the datapoint of an item, or combination of items, that yields the maximum possible weight at the current cost. The algorithm then traverses the array to find the index with the highest weight yield at the maximum price limit. Each datapoint in this index is added to a results list that is returned to the user. Just as with the brute force approach, this algorithm is subject to being a 0-1 knapsack. Any result using memoization returns a list filled entirely with the single best price-to-weight ratio. This is not ideal nor realistic for a practical grocery list.

Our logic for selecting the genetic algorithm was that the genetic approach of selecting 'fitness' of data points could prove to be a better fit. Genetic algorithms rely on randomization of selected items for a specific subset size and compares the values of each item. These subsets are stored as 'chromosomes', typically as a binary encoding, that have a length of the number of items. Using a fitness function, we can find the best chromosome in the subsets to fill the knapsack. Each encoded 1 in the chromosome corresponds to a specific item that is added to the knapsack while the 0 items are not added.

Lastly we have developed a user interface (UI) system that allows our program ease of use. An example of it can be seen in figure 1. We decided to create a GUI system using Tkinter, an import within pythons' built-in library, for our algorithm to help users interact with the system more easily. Using its different widgets such as Labels to display text, input areas to allow the user to input their desired budget amount, check boxes to enable or disable certain food groups and lastly a button to submit the users order. Also we used different windows to allow us to create an easy operating experience for the user.

Example User Interface



Hello and Welcome to our shopping list Application

strawberries
sweet onion
aspargus
raspberries
garlic
jalapeno peppers
blackberries
green cabbage
baby spinach
yellow peaches
white bread
pear
chocolate milk
cinnamon toast crunch
croissants
kiwi
can kidney beans
pinto beans
rotini pasta

Your Total is: $34.15

Fig. 1. Example userinterface allow the user to select food restrictions as well as inputting budget

3 Literature Review

"On the Partition of Numbers" by G. B. Mathews explores the multichoice, multi-dimensional knapsack problem back in 1896 as one of the earliest to do so. Although it was not known as the knapsack problem in his work, it does propose a problem of finding a set of non-negative integers that are a multivariate number with N number of assignments. Mathews concludes that the formula solution for some set of integers would be the same solution as used for a different set of integers [1].

Caserta, et al. propose tackling the multiple choice multidimensional knapsack using a lagrangian based scheme. Their technique combines lagrangian relaxation with the corridor method. Their results show the most competitive approaches for the nominal multiple choice multidimensional knapsack however their goal was to investigate the trade off between solution reliability and robustness price in hopes to bring theory and practice in operations research world [3].

"A hybrid genetic algorithm for the multidimensional knapsack problem" by F. Djannaty and S. Doostdar was an interesting read on the development of a genetic algorithm to work with MD knapsack. We looked at this article to see what it would take to create a hybrid GA to solve knapsack. Looked at the constraints they used for mutations and cross-breeding, as well as the functions and formulas they used to create the algorithm [4].

"A hybrid heuristic for the multiple choice multidimensional knapsack problem" by R. Mansi, C. Alves, C. Valerio de, and S. Hanafi. The aim of this article was to see how we could set up constraints to separate our food groups into different classes, so that we could run our algorithms. It was also our aim to see how a heuristic algorithm would operate for this problem. When looking into the testing they add one item from each class into their knapsack. When it came to the heuristic approach they refined the upper and lower bounds of their algorithm. Creating multiple stages in their algorithm to adjust to different inputs. Having cutting, fixation, and reformulation phases as well as multiple others used to calculate their solution [5].

"Neural Knapsack: A Neural Network Based Solver for the Knapsack Problem" by Nomer, Alnowibet, Elsayed, and Mohamed proposes a neural network and deep learning based heuristic for solving the knapsack problem. This is accomplished by keeping the relationships of items to the capacity based on previous attempts and leveraging the ability to utilize the distribution of the data. As such, the proposed model generalizes based on the correlation between values and weights of the items in the set and their history of being selected. The authors find that this method was able to outperform the greedy algorithm approach using their synthetic testing sets [6].

4 Results

To serve as a baseline, we have implemented a brute force algorithm that works on our dataset to select items that result 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 2.

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.

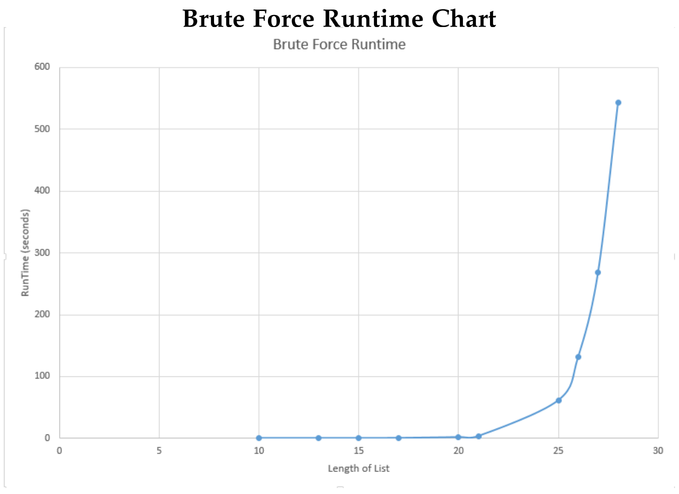


Fig. 2. Plotted runtime chart for the brute force algorithm

Testing using the memoization technique with the dynamic programming showed a linear increase in computation time versus the brute force technique, with a growth order of O(n*m) where n is the number of items in the dataset and m is the price limit of the grocery list. As can be seen in figure 3 the run time is much quicker compared to the brute force. In addition to this, it was able to clear the entire data set while the brute force was not. This can be seen more clearly in table 3. For comparison, the bruteforce runtime for a list of 20 items was 1.57 seconds while the dynamic programming took 0.000997 for the same list length. This is a much more rapid solution.

The genetic algorithm was able to clear the entirety of the custom dataset similarly to the dynamic programming. Despite this, across the board the genetic algorithm had a slower runtime than the dynamic programming by a factor of ten. However, despite the slower performance, the genetic algorithm did not have an increase in runtimes similar to the dynamic programming. As can be seen in table 4, there are certain larger list lengths that have a quicker runtime than smaller list lengths. The time difference between runtimes of varying list lengths did not increase by much while the other two baseline algorithms had a clear increase in runtimes as the dataset size increased. These results and runtimes can be seen in figure 4. There are situations where the output of the genetic algorithm surpasses the amount set by the budget restraint and these are due to the inner workings of the numpy library as well as rounding issues. There are times where the budget restriction is set to a price such as 35, and the total output list is 35.14.

Dynamic (memoization) Run Time	
length of list	runtime (seconds)
20	0.000997
40	0.00199
60	0.00199
80	0.00299
100	0.00399
120	0.00498
140	0.00499
160	0.00598
180	0.00598
189	0.00698

TABLE 3
Dynamic programming runtimes of custom dataset

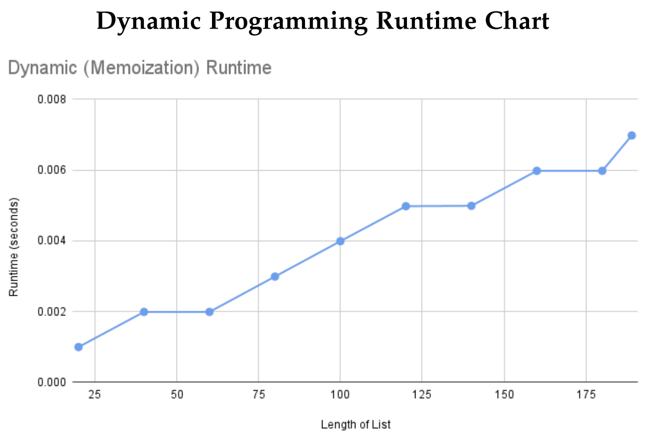


Fig. 3. Plotted runtime chart for the dynamic programming algorithm

Genetic Algorithm Run Times	
length of list	runtime (seconds)
20	0.03191
40	0.04088
60	0.04685
80	0.03191
100	0.04886
120	0.04985
140	0.05384
160	0.05684
180	0.04986
189	0.05483

TABLE 4
Genetic runtimes of custom dataset

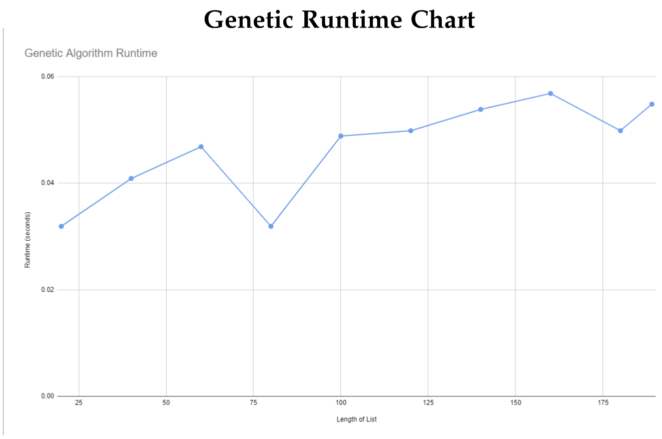


Fig. 4. Plotted runtime chart for the genetic algorithm

5 Discussion and Future Works

Two baseline 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 implemented a heuristic and nonconventional algorithm while considering other unconventional methods such as neural networks or other advanced designes with upper and lower bounds [6] [5].

Looking at the genetic algorithm it is clear that is being given a disadvantage by the small dataset. Because the dataset is only composed of 189 items, the dynamic programming algorithm is able to clear it much quicker than the genetic algorithm. However, if the dataset were to be much larger such as a real world retail grocery store dataset with over 10,000 items, the genetic algorithm would be able to clear the dataset much more quickly. This is due to the advantage of the genetic algorithm to run on a heuristic and only being time limited based on its ability to process its fitness function and generations. On the other hand the conventional dynamic programming algorithm would be forced to create a table of the entire dataset and go through each and every item. Another benefit that the genetic algorithm was able to provide over the other two algorithms was that it introduced a level of diversity. Because it works by introducing random populations of items, the pre-selected items are diversified in their weights, prices, and food types whereas the dynamic and brute force algorithms will output the same results and answers each time.

Plans to increase the dataset size were also considered but not executed due to the limitations and reasoning of creating the custom dataset in the first place. Because of the amount of redundancy and item overlap of traditional large chain grocery stores we were not able to increase the amount of produce and meat or dairy products in our dataset. There are only a certain amount of raw food items that can be considered without taking in the account of brand and sizes. Increasing our dataset to a larger size would result in redundancy of multiple brands and sizes of the same items.

In addition to the dataset problem, there were many other design issues that our project implementation faced. 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, a typical 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 weigh 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. This can be combated by dividing the initial budget into the ratios of the food groups and then running the genetic algorithm in parallel with a reduced budget. Once this is complete the list would be combined. However, we also decided to opt out of this implementation as it is not a true multi-dimensional knapsack with food groups being a third dimension. This simply is just subdividing the budget and then performing the genetic algorithm with a subdivided dataset multiple times over for each food group. This does not introduce a new dimension or complexity to the problem. Another aspect of the food group design is the difficulty to quantify the food items. How many servings is a bag of rice or within one water melon? These aspects make it difficult to divide the food groups into ratios that are to be added to the algorithm. With that being said, there are some design choices that need to be made and reconsidered for the future of this endeavor and that the initial design of the project was met with many overly complicated ideas that were actually too simple to solve.

6 Peer Session Reviews

Each group member picked a presentation and wrote about what they found interesting about the group's project. One of the presentations that we found ourselves interested in was the Solving Sudoku Puzzle using Deep Learning. We looked into this one to see what deep learning technique they were using to accomplish their goal state. When looking at their implementation process we saw that they implemented CNN model in order to solve their puzzle which took about three seconds to complete. It was interesting to see how they implemented CNN in the fact they had to reshape after flattening their model.

Our second presentation that we decided to look into was the Classification of Star project. One of our members did a similar project in another course that also attempted to classify stars by stellar types, although using a different type of classification. Stars are typically classified by two methods of classification. Heil used the Morgan-Keenan scale spectral types, which are seven types from O to M, based on the estimated temperature of the star derived from emission spectra readings. As alluded to in Heil's presentation, another classification type is the Yerkes type based on the evolutionary states of stars, such as dwarfs and giants. Heil used decision tree and SVM models to classify his data. Because the clustering of the data should be pretty well defined, it would make sense to use these models. This is especially the case for SVM as, depending on the type, is a discriminative model for finding the decision boundary of a class. Heil's approach for his data was to use a dataset from Kaggle that had feature engineering already done. This is advantageous as this feature engineering requires in depth knowledge of how professional astronomers use the raw data. For example, it already has temperature defined, instead of being typically derived from the intersection of blue and violet filtered channels of the electro-magnetic spectrum.

Our third presentation that we picked was a presentation about implementing digit recognition, using neural networks as algorithm solver. When looking into their implementation it was interesting to see that they implemented KNN and CNN models. When looking at their results it was interesting to see that KNN and CNN model accuracies were about the same when it came to single digits. But when it comes to multiple numbers it seems from the data results that a CNN model is needed for getting the correct number. It was also interesting to see what dataset they were using to develop their project. Using two different sets of data to look at two potential problems and trying to solve both of them using one algorithm.

7 Conclusion

Our implementation of brute force to approach the knapsack problem aligns with the conventional train of thought. Run times were incredibly slow especially as the size of the dataset increases resulting in an exponential time complexity growth. In addition to this, our dynamic programming approach also matches what can be found with literature. Dynamic programming allows a much more rapid solution to the knapsack in comparison to the brute force algorithm and surprisingly faster than a genetic algorithm given this small dataset size. Unfortunately we were not able to provide a dataset that allowed the genetic algorithm to show its true strengths in comparison to conventional algorithms. With these three we have developed a UI system for ease of use.

We were unable to implement the second half of our project implementation due to various philosophical design aspects. Unbound knapsacks exist and are realistic representations of grocery stores while this knapsack design is not applicable for any of the algorithms that we developed. In addition to this, food groups as a third dimensional is nearly undefinable and face similar problems of the unbound knapsack with a limited amount of selection and diversity. For these multitude of reasons we have decided to alter our implementation and must consider further design choices in the future.

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

[2] M. Agha, "The Binary Multidimensional Knapsack Problem (MKP)," Jul. 2021. [Online]. Available: https://towardsdatascience.com/the-binary-multidimensional-knapsack-problem-mkp-2559745f5fde

[3] M. Caserta and S. Voß, "The robust multiple-choice multidimensional knapsack problem," *Omega*, vol. 86, pp. 16–27, Jul. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305048317305777

[4] F. Djannaty and S. Doustdargholi, "A hybrid genetic algorithm for the multidimensional knapsack problem," *Int. J. Contemp. Math. Sciences*, vol. 3, pp. 443–456, 01 2008.

[5] R. Mansi, C. Alves, J. M. Valério de Carvalho, and S. Hanafi, "A hybrid heuristic for the multiple choice multidimensional knapsack problem," *Engineering Optimization*, vol. 45, pp. 983–1004, Aug. 2013, aDS Bibcode: 2013EnOp...45..983M. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2013EnOp...45..983M

[6] H. A. A. Nomer, K. A. Alnowibet, A. Elsayed, and A. W. Mohamed, "Neural Knapsack: A Neural Network Based Solver for the Knapsack Problem," *IEEE Access*, vol. 8, pp. 224 200–224 210, 2020.