CS5800 ALGORITHMS

PROJECT REPORT

TITLE: OPTIMIZING SHOPPING EXPERIENCE

Team members:

Rahul Chandak

Darshan Mahendra Bhokare

Ganavi Jayaram

Shreya Ale

Project Link:

https://www.youtube.com/watch?v=zgUHf56CnxI&feature=youtu.be&ab_channel=Darshan Bhokare

Contents

1. Introduction

Problem Statement:

Shopping can be challenging for many people, as they often struggle to stick to their budget or carrying capacity or the nutritional value of the contents they buy. Our algorithm aims to address this issue by providing personalized recommendations on which items to purchase, prioritizing the items of the highest value (weight or nutritional value) according to the user's budget. By following these recommendations, users can maximize the efficiency of their shopping trip, ensuring they purchase the most important items first.

Why we chose this topic:

1. *Rahul:* This topic, in particular, was important to me because I have found myself in situations where only after billing do I realize that the weight of the items I bought is too much and carrying them back won't be easy. Or at the last moment, I find myself eliminating items from my cart which leads to me missing out on something important. Therefore I needed an application that can shortlist the maximum desired items within my budget and pre-inform me how much stuff I can expect to carry.

2. *Darshan:* As a student, shopping for groceries and other household items can be a real hassle for me. Not only do I have to stick to a tight budget, but I also have to carry everything myself, as I don't own a car. This means that I need to carefully consider the weight and size of each item I purchase, and I'm often limited in how much I can carry at once. To make matters worse, finding the time to shop can be a challenge with my busy schedule, and navigating the crowded aisles of the grocery store can be overwhelming. And even after I've made my purchases, I still need to figure out how to transport everything back to my dorm or apartment. It can be a real headache! That's why I think a personalized shopping recommendation algorithm that takes into account the cost and carrying capacity of items could be a game-changer for me. With such a tool, I could more easily prioritize my purchases, ensuring that I get the most important items within my budget and carrying capacity first. This would save me time, money, and effort, and make the whole shopping experience much more manageable.

3. *Ganavi:* The algorithm would have been extremely useful when I had a job. As a busy professional with limited time and resources, the algorithm would have helped me maximize my shopping efficiency by providing personalized recommendations on which items to prioritize. It would have saved me time and hassle from browsing through countless aisles and would have helped me stick to my budget by ensuring that I only purchased essential items. The algorithm would have also taken into account my limited carrying capacity, recommending items that were easy to carry or that could be divided into multiple trips. Overall, the algorithm would have been a valuable tool in simplifying my shopping experience and helping me make informed decisions that met my individual needs and preferences.

4. *Shreya:* As a student, this project is valuable as it solves a common shopping problem by providing personalized recommendations on which items to purchase based on value and weight preferences. It maximizes shopping efficiency, crucial for individuals on tight budgets or limited carrying capacity like students. I can save time,

money, and effort by using this algorithm. Participating in this project helps me develop my research, analytical, and problem-solving skills. Contributing to a team that develops real-world solutions is exciting and rewarding. Overall, this project benefits individuals who face similar challenges while shopping, making it an exciting opportunity to be a part of.

2. Analysis

What is the current scenario? What are the shortcomings of this?

- In today's world, offline shopping has become more of an intuitive task than a planned activity. Many people enter stores with a vague idea of what they need or want to buy, without any specific plan or budget in mind. As they browse through the store, they may be tempted by various products, leading them to make impulsive purchases.
- These unplanned purchases can quickly add up and cause shoppers to overspend, often realizing their mistake only after they see the final bill. In addition, they may end up buying items that were lower on their priority list, or even unnecessary. To address these issues, it's important for shoppers to prioritize their needs and wants before entering a store. This could involve making a list of essential items and sticking to them, as well as setting a budget for each shopping trip. It may also be helpful to consider factors such as the weight of the final bag or the nutritional value of the items being purchased.
- By taking a more planned and thoughtful approach to offline shopping, individuals can make better use of their time and money, and ultimately be more satisfied with their purchases.

What would be the benefits if this problem was solved?

- If priority is the weight of the cart and spending limit, before leaving home the user will have a rough idea of how many pounds/ kgs of stuff they might have to carry back and can decide the mode of transportation to take.
- If the priority is the nutritional value, the user will get an idea of what items to buy first that will maximize the nutritional value of their cart within their spending limit.
- If they have a predefined list of items ready, it will save a lot of time for the user eliminating the need to manually shortlist items.

Use cases!

- Students: While shopping for groceries they could use the algorithm to prioritize which food items to purchase based on their weight value. Students usually don’t own private vehicles so they can maximize the weight of the cart and carry it back in a ‘single’ taxi trip.

- Someone who is moving to a new city and needs to furnish their apartment on a limited budget could use this algorithm to prioritize which items to purchase first based on their relative importance, value, and weight.
- A family that is grocery shopping on a tight budget could use this algorithm to prioritize which food items to purchase based on their nutritional value.
- Holiday shopping: The algorithm would help them to prioritize which gifts to purchase based on their value and weight, ensuring that the total cost of the purchased items does not exceed their budget limit and that they can carry the purchased items home.

3. How do we plan to achieve this solution?

- We plan to apply the knapsack algorithm with various parameters like weight, nutritional value, etc. to optimize the shopping experience for the user.
- The plan is to demonstrate by comparing a naïve approach with the optimized solution involving dynamic programming.
- In the course, we learned about dynamic programming in Module 8 and Module 9. To be specific knapsack algorithm was covered in Module 8: 8.4.
- Initially, we will cover the solution with a recursive approach and then optimize it with a dynamic programming approach concluding why the latter is better than the two.

Assumptions:

1. User has pre-decided the list of items from which they wish to buy.
2. The approximate cost of the items entered by the user is close to the actual price of the item in the store.

Approach 1: The naïve approach (0/1 Knapsack using recursion)

Input:

1. List of item weights (w[1], w[2], ..., w[n])
2. List of item values (v[1], v[2], ..., v[n])
3. Knapsack weight capacity (W)

Algorithm:

1. Define a recursive function knapsack(i, W) that takes two arguments:
    a. i: the index of the item being considered
    b. W: the remaining weight capacity of the knapsack

2. If i is zero or W is zero, return zero, since we cannot select any items when either the list of items or the knapsack capacity is empty.
3. If w[i] is greater than W, return knapsack(i-1, W), since we cannot select the current item if its weight exceeds the remaining capacity.
4. Otherwise, return the maximum of the following two values:
    a. knapsack(i-1, W), which represents the maximum value obtained by excluding the current item and selecting items from the first i-1 items with weight <= W.
    b. v[i] + knapsack(i-1, W-w[i]), which represents the maximum value obtained by including the current item and selecting items from the first i-1 items with weight <= W-w[i].
5. Call the function knapsack(n, W), where n is the total number of items.
6. Return the result obtained from step 5.

Explanation:

This algorithm is a recursive implementation of the classic Knapsack problem. The Knapsack problem is a combinatorial optimization problem that seeks to find the optimal combination of items to pack into a knapsack (or backpack) of limited capacity, such that the value of the packed items is maximized.

The input to the algorithm includes two lists of item weights and values, respectively, and the weight capacity of the knapsack. The algorithm defines a recursive function called knapsack, which takes as arguments the index i of the item being considered and the remaining weight capacity W of the knapsack.

The knapsack functions first checks if either the list of items or the knapsack capacity is empty. If either of these conditions is true, the function returns zero, since there are no items that can be selected.

If the weight of the current item is greater than the remaining capacity of the knapsack, the function returns the maximum value obtained by excluding the current item and selecting items from the first i-1 items with weight <= W.

Otherwise, the function returns the maximum of two values: (1) the maximum value obtained by excluding the current item and selecting items from the first i-1 items with weight <= W, and (2) the maximum value obtained by including the current item and selecting items from the first i-1 items with weight <= W-w[i].

The algorithm then calls the knapsack function with the total number of items n and the knapsack capacity W, and returns the result obtained from the function call.

Overall, the algorithm is a simple recursive implementation of the Knapsack problem and can be used to find the optimal combination of items to pack into a knapsack of limited capacity, such that the value of the packed items is maximized.

Time Complexity:

The recursive function knapsack is called 2^n times, where n is the number of items. Each call to knapsack performs a constant amount of work (i.e., comparison and addition operations). Therefore, the time complexity of the recursive algorithm is O(2^n).

Space Complexity:

The space used by the call stack is proportional to the maximum depth of the recursion tree, which is n. Therefore, the space complexity of the recursive algorithm is O(n).

Approach 2: Optimized approach (0/1 Knapsack using dynamic programming)

Problem statement: Given a set of items, each with a weight and a value, determine the items to include in a collection such that the total weight is no more than a given limit and the total value is as large as possible.

Inputs:

items: a list of tuples where each tuple represents an item and has the form (value, weight)
limit: the maximum weight that the knapsack can hold

Output:

A tuple (max_value, items_in_knapsack) where max_value is the maximum value that can be obtained by filling the knapsack and items_in_knapsack is a list of the items that should be included in the knapsack to achieve this maximum value.

Algorithm:

1. Create a table with dimensions (n+1) x (limit+1) where n is the number of items and limit is the maximum weight that the knapsack can hold. Initialize all entries to
2. Iterate over each item i from 1 to n.
3. Iterate over each weight w from 1 to limit.
4. If the weight of item i is less than or equal to w, set the value of the current cell to the maximum of:
5. The value of the previous row and same column (corresponding to not including the current item)
6. The value of the cell in the previous row and column w minus the weight of item i (corresponding to including the current item)
7. If the weight of item i is greater than w, set the value of the current cell to the value of the previous row and same column.
8. The maximum value that can be obtained by filling the knapsack is the value in the cell (n, limit).
9. To determine which items were included in the knapsack, start at cell (n, limit) and iterate backward through the table:
10. If the value in the current cell is not equal to the value in the cell above it, then item i was included in the knapsack. Add item i to the list of items_in_knapsack and subtract the weight of item i from the current weight w.

11. Repeat for the previous row until reaching the first row.
12. Reverse the order of items_in_knapsack since we added items in reverse order

Code:

```
import java.util.*;


public class knapsackBudget {

 static int[][] dp;

 static int[] cost;

 static int[] weight;

 static String[] name;

 static int n, C;


 public static void main(String[] args) {

   Scanner sc = new Scanner(System.in);

   System.out.print("Enter your budget for today: ");

   C = sc.nextInt();

   System.out.print("Enter the number of items you plan to purchase: ");

   n = sc.nextInt();


   name = new String[n + 1];

   cost = new int[n + 1];

   weight = new int[n + 1];

   dp = new int[n + 1][C + 1];


   System.out.println("Enter the item name, cost per unit and weight (in
lb) of each item:");

   for (int i = 1; i <= n; i++) {

     name[i] = sc.next();
```

```
    cost[i] = sc.nextInt();

    weight[i] = sc.nextInt();

  }



  int max_value = knapsack();

  List<String> items = getSelectedItems(name);



  System.out.println("For your budget, in a single trip, find the
details below : ");

  System.out.println("The maximum weight you can carry is: " +
max_value + " lbs");

  System.out.println("The items within this weight and budget will be:
" + items);

 }



 static int knapsack() {

  for (int i = 0; i <= n; i++) {

    for (int j = 0; j <= C; j++) {

      if (i == 0 || j == 0) {

        dp[i][j] = 0;

      } else if (cost[i] <= j) {

        dp[i][j] = Math.max(dp[i-1][j], weight[i] +
dp[i-1][j-cost[i]]);

      } else {

        dp[i][j] = dp[i-1][j];

      }

    }

  }

  return dp[n][C];

 }
```

```
  static List<String> getSelectedItems(String[] name) {
    List<String> items = new ArrayList<>();
    int j = C;
    for (int i = n; i >= 1; i--) {
      if (dp[i][j] > dp[i-1][j]) {
        items.add(name[i]);
        j = j - cost[i];
      }
    }
    Collections.reverse(items);
    return items;
  }
}
```

Test Case

```
Enter your budget for today: 7
Enter the number of items you plan to purchase: 5
Enter the item name, cost per unit and weight (in lb) of each item:
a 2 1
b 3 2
c 4 2
d 2 3
e 1 5
For your budget, in a single trip, find the details below :
The maximum weight you can carry is: 10 lbs
The items within this weight and budget will be: [b, d, e]
```

Explanation

This algorithm is implementing the 0/1 Knapsack problem to solve a shopping problem, where a user gives a budget and a list of items with their cost and weight, and they have to select the items that they can carry within the given budget and maximize the total weight of the selected items. The algorithm uses dynamic programming to solve this problem.

The input of the algorithm includes the budget 'C' and the number of items 'n'. Then it takes 'n' lines of input, each containing the name, cost, and weight of an item. After taking the input, it initializes three arrays - 'name', 'cost', and 'weight' - to store the name, cost, and weight of each item respectively. It also initializes a 2D array 'dp' to store the maximum weight that can be carried within the given budget for all the subproblems.

The 'knapsack' function is used to calculate the maximum weight that can be carried within the given budget. It uses a nested loop to iterate over all the subproblems. For each subproblem, it checks whether the current item can be included in the knapsack or not. If it can be included, then it calculates the maximum weight that can be carried either by including or excluding the current item. The maximum weight is stored in the 'dp' array. Finally, the function returns the maximum weight that can be carried within the given budget.

The 'getSelectedItems' function is used to retrieve the selected items from the 'dp' array. It iterates over the 'dp' array in reverse order and checks whether the current item is included in the knapsack or not. If it is included, then it adds the name of the item to a list 'items' and updates the remaining budget. Finally, it reverses the 'items' list and returns it.

In the given test case, the user has a budget of 7 and 5 items to choose from. The algorithm calculates the maximum weight that can be carried within the given budget as 10 lbs and the items that can be carried within the budget and this weight is 'b', 'd', and 'e'. These items have a total price of $6, which is within the given budget.

Time Complexity

The time complexity of the "knapsack" algorithm in the provided Java code is O(n*C), where n is the number of items and C is the maximum budget allowed. This is because the algorithm uses a nested loop structure to iterate through all possible combinations of items and budgets to determine the maximum weight that can be carried within the given budget.

The outer loop iterates over all items from 0 to n, and the inner loop iterates over all possible budgets from 0 to C. Within each iteration of the inner loop, the algorithm performs a constant amount of work to determine the maximum weight that can be carried, based on whether or not the current item is included in the knapsack. Therefore, the time complexity of the algorithm is proportional to the product of the number of items and the maximum budget, resulting in a time complexity of O(n*C).

Space complexity

The space complexity of the "knapsack" algorithm in the provided Java code is O(n*C), where n is the number of items and C is the maximum budget allowed. This is because

the algorithm uses a 2D array called "dp" of size (n+1) by (C+1) to store the maximum weight that can be carried for each possible combination of items and budgets.

During the execution of the algorithm, the "dp" array is populated with values based on the items' costs and weights, and the maximum budget allowed. This means that the entire "dp" array needs to be stored in memory, resulting in a space complexity of O(n*C).

Additionally, the algorithm uses several auxiliary arrays of size n+1 to store the cost, weight, and name of each item, which contribute to the overall space complexity. However, since these arrays have a maximum size of n+1, they do not contribute significantly to the space complexity of the algorithm.

4. Application:

One potential use case for a knapsack solution that optimizes a shopping experience for a fixed budget and maximum nutritional value is in the development of a personalized meal planning and grocery shopping application. The application could allow users to input their dietary preferences, restrictions, and a fixed budget. The application would then use a knapsack algorithm to suggest a list of grocery items that the user can purchase while maximizing the nutritional value within their budget.

The problem arises when we have to define what we mean by ‘maximizing the nutritional value’. In the above algorithm, where we use dynamic programming, we input the weight of the items as a single number. But the nutritional value in itself cannot be represented by a single number. It may contain various factors like the number of calories, proteins, sugar, etc. and hence determining based on a single value could result in unfair decisions.

According to a 2011 poll by Consumer Reports Health, 90 percent of Americans believe that they eat a healthy diet. The reason is common people, don’t understand what a ‘healthy diet’ consists of. One such attempt to simplify this term was given by Dr. Joel Fuhrman. To illustrate which foods have the highest nutrient-per-calorie density, Dr. Fuhrman created the aggregate nutrient density index or ANDI. It lets you quickly see which foods are the most health-promoting and nutrient dense.

In our proposed solution, instead of the weight, we used the ANDI score as a metric that will help users purchase food items within their budget with the maximum nutritional value of all. There are various single-number nutrition rating systems like the NuVal Nutritional Scoring System, HSR (Health Star Rating), FSA Score (Food Standards Agency Score), NRF (Nutrient Rich Foods Index), etc.

5. Limitations and Weaknesses

- We assume that the user knows the exact price or weight of the product. This information provided by the user may or may not be accurate every time.
- It is limited to one-time shopping. We assume that the customer is buying items only once. Many customers go shopping multiple times, which means the algorithm needs to be adapted to handle ongoing shopping experiences.
- It also doesn't account for dependent products. For example, a user may need to buy a specific accessory or component to use a particular product. This might result in an incomplete or inefficient shopping experience.

6. Future Scope

- Shopping in general is an activity that can be influenced by many factors.
- To cover the above limitations, one approach is to use a multidimensional knapsack algorithm to optimize the shopping experience based on multiple criteria, such as price, quality, and nutritional value.
- Another enhancement to this project can be to track the availability of the concerned product in the store. Based on availability, the algorithm can give recommendations of the items the user plans to buy.
- A far-fetched approach can be to suggest to the users what items of what brand they should buy based on their needs or preferences. This can be achieved by using K-means clustering which can be used to group similar products together. Attributes for clustering can be price, quality, calories, etc.

7. Conclusion

Shopping is something which is done on various levels by everyone very frequently and recursively. Through this project a small effort was made towards optimizing this daily task and simplifying the shopping experience for users, helping them make informed decisions that meet their individual needs and preferences. If we think about it, people of all age groups, from juniors to seniors, individuals to families, and students to professionals can benefit from this in some way. The Knapsack algorithm with dynamic programming proves to give us an optimized solution. This algorithm can be extended with the addition of more parameters which allows more choices for the user in a long run.

8. What did we learn from this?

1. *Rahul* - It was a great opportunity to think about how I can provide value to customers who want to make the best use of their budget while shopping. Learning about the knapsack algorithm and its applications has intrigued interest in me as to how these theoretical concepts can be put into practice to solve common everyday problems. Learning about the naive recursive approach first and then proceeding step by step toward the DP solution allowed me to think from multiple angles.
2. *Darshan*: This project taught us the challenges faced by shoppers and the benefits of personalized recommendation algorithms. Prioritizing items based on weight, nutritional value, etc can help shoppers maximize their time and money. We learned about applying algorithms such as knapsack and dynamic programming, and the importance of comparing approaches for effective solutions. This algorithm has the potential to improve the shopping experience for students, families, and holiday shoppers, making a significant impact on their daily lives.
3. *Ganavi*: Through this project, we have gained a deeper understanding of the challenges that shoppers face when trying to make informed and efficient purchasing decisions. By analysing the shortcomings of the current shopping scenario, we identified the potential benefits of implementing a personalised recommendation

algorithm that prioritises items based on weight, nutritional value, and other parameters. We have learned how to apply various algorithms, such as the knapsack algorithm and dynamic programming, to optimise the shopping experience for users. Additionally, we gained experience in comparing different approaches and evaluating their effectiveness to select the best solution.By testing and comparing a greedy algorithm approach with an optimised dynamic programming approach, we learned how the latter could provide better results in terms of maximising the value of the cart while staying within the user's budget.

4. *Shreya*: This project has been a great learning experience for me. The Knapsack algorithm with dynamic programming has helped me save time, money, and effort while shopping. I've also enhanced my problem-solving skills and learned to extend the algorithm with more parameters for greater choice. It's been an excellent opportunity to apply my knowledge and skills to real-world problems and grow both personally and professionally.

7. References

- https://www.sciencedirect.com/science/article/abs/pii/S0377221715009212
- https://inviarobotics.com/blog/knapsack-problem-and-warehouse-optimization/#:~:text=The%20Knapsack%20Problem%20is%20an,most%20value%20into%20the%20knapsack
- https://apmonitor.com/me575/index.php/Main/KnapsackOptimization
- https://www.drfuhrman.com/blog/128/andi-food-scores-rating-the-nutrient-density-of-foods