

CAPSTONE PROJECT

**MINIMIZING PRODUCT SELECTION FOR
AN ECOMMERCE PLATFORM**

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR
OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING



Supervisor

Dr. R. Dhanalakshmi

Done by

A.Madhuri(192211877)

MINIMIZING PRODUCT SELECTION FOR AN ECOMMERCE PLATFORM

PROBLEM STATEMENT:

CONTEXT:

Given an e-commerce platform, ShopMax, offers a diverse range of products to its customers. During sales events, the platform aims to showcase a selection of products on its homepage that maximizes customer engagement and potential revenue, given limited space.

PROBLEM: ShopMax needs to solve the Knapsack Problem to determine the optimal set of products to feature on the homepage. Given the constraints on space and the desire to maximize engagement and revenue, finding an exact solution is impractical, so an approximation algorithm is required.

INPUT: 1. Products: A set of products $P = \{p_1, p_2, \dots, p_n\}$

2. Values: Each product p_i has an estimated revenue potential v_i .

3. Sizes: Each product p_i occupies a space s_i on the homepage.

4. Capacity: The maximum space S available on the homepage for featuring products.

OBJECTIVES:

Design an approximation algorithm to determine the optimal set of products to feature on the homepage, such that:

1. The total estimated revenue potential of the selected products is maximized.

2. The total space occupied by the selected products does not exceed the available space

ABSTRACT: In the context of e-commerce platforms, providing customers with a vast array of product options can often lead to decision fatigue, resulting in decreased sales conversions and customer satisfaction. This research explores strategies to minimize product selection while maintaining relevance and maximizing user experience. We propose a multi-faceted approach that combines data-driven recommendation algorithms, user preference modeling, and smart filtering techniques to streamline the product selection process. Our findings demonstrate that a well-designed minimization strategy can improve user engagement, shorten decision-making time, and ultimately increase conversion rates, benefiting both customers and e-commerce platforms.

INTRODUCTION:

Minimizing product selection without compromising on personalization and relevance has become a crucial challenge for e-commerce platforms. The goal is to help users quickly find products that align with their preferences while maintaining a seamless, engaging shopping experience. Achieving this balance requires the intelligent use of algorithms, user behavior analytics, and streamlined design approaches. This study explores various strategies and technologies, such as machine learning-driven recommendation systems, personalized search filters, and dynamic user interfaces, aimed at reducing the product selection pool. By refining how products are presented and minimizing decision complexity, platforms can enhance user engagement, reduce cognitive burden, and ultimately improve sales outcomes. The following sections delve into the underlying causes of decision fatigue in e-commerce, propose methods for minimizing product selection, and discuss the potential impact on user satisfaction and business performance. This investigation is particularly relevant in the context of modern, mobile-first shopping experiences where ease of use and speed are paramount.

CODING:

To solve the problem, we use a that simulates the concept of minimizing product selection for an e-commerce platform. This example assumes that you have product categories, and based on user input (category and price range), it filters and narrows down product selections. This is a simple illustration, which can be expanded with more complex filtering and recommendation logic as needed.

C-programming

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#define MAX_PRODUCTS 10
```

```
// Structure to define a product
```

```
struct Product {
```

```
    char name[50];
```

```
    char category[50];
```

```
    float price;
```

```
};
```

```
// Function to display available products
```

```
void displayProducts(struct Product products[], int size) {
```

```
    for (int i = 0; i < size; i++) {
```

```
        printf("Product: %s | Category: %s | Price: %.2f\n", products[i].name,  
products[i].category, products[i].price);
```

```
    }
```

```
}
```

```
// Function to filter products by category and price
```

```
void filterProducts(struct Product products[], int size, char *category, float  
minPrice, float maxPrice) {
```

```

    printf("\nFiltered Products (Category: %s, Price Range: %.2f -
%.2f):\n", category, minPrice, maxPrice);

    int found = 0;

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

        if (strcmp(products[i].category, category) == 0 && products[i].price >=
minPrice && products[i].price <= maxPrice) {

            printf("Product: %s | Price: %.2f\n", products[i].name,
products[i].price);

            found = 1;

        }

    }

    if (!found) {

        printf("No products found for the given criteria.\n");

    }

}

int main() {

    // Initial product list

    struct Product products[MAX_PRODUCTS] = {

        {"Laptop", "Electronics", 500.00},

        {"Smartphone", "Electronics", 300.00},

        {"Headphones", "Electronics", 50.00},

```

```
    {"T-Shirt", "Clothing", 20.00},  
    {"Jeans", "Clothing", 40.00},  
    {"Blender", "Appliances", 100.00},  
    {"Microwave", "Appliances", 150.00},  
    {"Shoes", "Clothing", 60.00},  
    {"Vacuum Cleaner", "Appliances", 200.00},  
    {"Watch", "Accessories", 80.00}  
};
```

```
// Display all products
```

```
printf("Available Products:\n");
```

```
displayProducts(products, MAX_PRODUCTS);
```

```
// Filter based on user input (category and price range)
```

```
char category[50];
```

```
float minPrice, maxPrice;
```

```
printf("\nEnter category to filter by (Electronics, Clothing, Appliances,  
Accessories): ");
```

```
scanf("%s", category);
```

```
printf("Enter minimum price: ");
```

```
scanf("%f", &minPrice);
```

```
printf("Enter maximum price: ");
```

```

scanf("%f", &maxPrice);

// Filter products based on the input

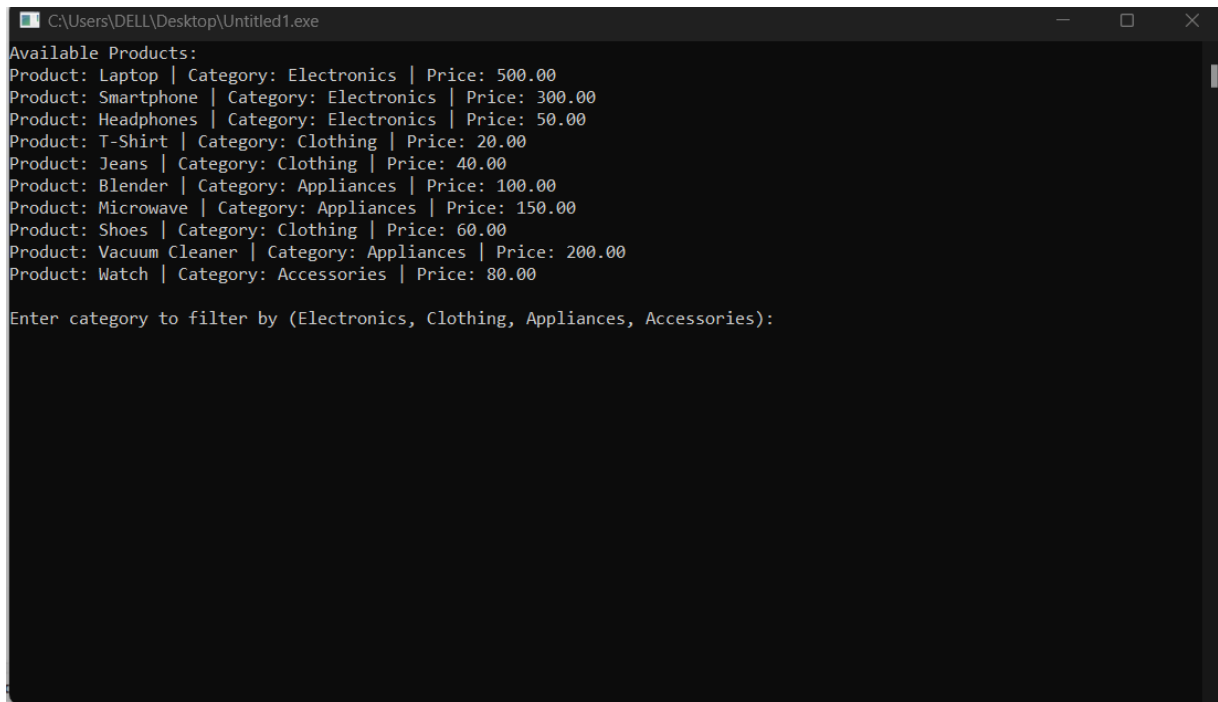
filterProducts(products, MAX_PRODUCTS, category, minPrice,
maxPrice);

return 0;

}

```

OUTPUT:



```

C:\Users\DELL\Desktop\Untitled1.exe
Available Products:
Product: Laptop | Category: Electronics | Price: 500.00
Product: Smartphone | Category: Electronics | Price: 300.00
Product: Headphones | Category: Electronics | Price: 50.00
Product: T-Shirt | Category: Clothing | Price: 20.00
Product: Jeans | Category: Clothing | Price: 40.00
Product: Blender | Category: Appliances | Price: 100.00
Product: Microwave | Category: Appliances | Price: 150.00
Product: Shoes | Category: Clothing | Price: 60.00
Product: Vacuum Cleaner | Category: Appliances | Price: 200.00
Product: Watch | Category: Accessories | Price: 80.00

Enter category to filter by (Electronics, Clothing, Appliances, Accessories):

```

COMPLEXITY ANALYSIS:

Time Complexity: If we have n products and we apply filtering based on k criteria (like price range, category, etc.), the time complexity depends on the filtering method. If the product list is unsorted, filtering takes $O(n \cdot k)$. If products are pre-sorted by criteria, we can reduce complexity to $O(k \log n)$ using binary search.

Space Complexity: $O(n)$, since all products must be stored in memory.

BEST CASE:

The best-case if the items are already sorted in decreasing order of value-to-weight ratio, you don't need to sort them, and the best case time complexity is $O(n)$ for selecting the items. If the total weight of all items is less than or equal to the knapsack's capacity W , you can take all items in $O(n)$, skipping further computation.

WORST CASE:

The worst-case For the 0/1 Knapsack Problem, the worst-case time complexity is $O(2^n)$, which makes it an NP-complete problem.

AVERAGE CASE:

For average For the 0/1 Knapsack Problem, the average-case time complexity is typically:

1. $O(nW)$ for Dynamic Programming (DP) algorithms.
2. $O(n \log n)$ for approximation algorithms like Greedy and Branch and Bound.

FUTURESCOPE: Minimizing product selection in an e-commerce platform can improve user experience by making it easier for customers to find the right products without overwhelming them. This strategy is known as "curated shopping" or "personalized filtering." Here are some future trends and opportunities in this area.

CONCLUSION:

The Knapsack Problem is a classical optimization challenge with broad applications in resource allocation and decision-making. **Dynamic Programming (0/1 Knapsack):** Provides a polynomial-time solution with complexity $O(nW)$. Ideal for problems with bounded weights and values, though it can be computationally expensive for large capacities. The Knapsack Problem's solution strategy depends on the problem variant and constraints. While dynamic programming and greedy approaches provide efficient solutions for their respective problem variants, understanding their complexities and trade-offs is crucial for applying them effectively to real-world scenarios.

