Vending Machine Project

Analysis Lab

Adham Ali

900223243

Submitted to Dr. May Shalaby
This paper is prepared for CSCE2203, section 02

THE AMERICAN
UNIVERSITY IN CAIRO

Department of Computer Science and Engineering
The American University in Cairo

03/12/2024

Table of contents

Chapter 1

Introduction

Vending machines are integral to modern convenience, offering quick access to products without the need for human assistance. Their efficiency depends on the ability to process payments and provide accurate change reliably.

The change optimization problem in vending machines is a critical aspect of their operation. It involves determining the optimal way to dispense change from a finite set of denominations, considering constraints such as the availability of coins and notes. The solution must balance efficiency, optimality, and customer satisfaction.

This report examines three algorithmic solutions to this problem:

1. **Greedy Approach:** A fast, straightforward solution that may not always find the optimal result.
2. **Dynamic Programming Approach:** A systematic method that guarantees optimality by exploring all possibilities in a structured manner.
3. **Brute Force Approach:** An exhaustive method that guarantees an optimal solution but is computationally expensive.

Chapter 2

Problem Description

The vending machine system solves the following problem:

- **Objective:** Dispense the exact change using the minimum number of coins or bills.
- **Inputs:**
  - Available denominations and their counts.
  - Required change amount.
- **Outputs:** A list of coins or bills used to provide the change.
- **Constraints:**
  - The machine must provide exact change.
  - It should not dispense denominations that are unavailable or exceed the remaining balance.

Additionally, the vending machine maintains a product inventory and currency stock, dynamically updating these after every transaction.

Chapter 3

Code Explanation

This chapter explains the functionality of the main parts of the code and their roles in the overall system.

3.1 Stock and Currency Management

The vending machine's state is preserved between program runs by reading from and writing to external files. For example:

```
void loadStock(vector<pair<string, int>>& products, vector<int>& stock) {
    ifstream stockFile("Stock.txt");
    if (!stockFile) {
        cerr << "Error: Unable to open stock file!" << endl;
        exit(1);
    }

    products.clear();
    stock.clear();
    string name;
    int price, count;

    while (stockFile >> name >> price >> count) {
        products.push_back({name, price});
        stock.push_back(count);
    }

    stockFile.close();
}
```

Explanation:

- `products`: A vector of pairs where each pair contains the product name and price.
- `stock`: A vector that keeps track of the available quantity for each product.
- The function ensures that the data is cleared before loading new information to avoid duplication.

3.2 Display Functions

The display functions provide real-time feedback to the user about the state of the vending machine:

- `displayProducts`: Lists the products available for purchase along with their prices and stock levels.
- `displayCurrency`: Displays the remaining quantity of each currency denomination.
- `displayStock`: Summarizes the remaining stock of all products in the machine.

3.3 Transaction Logic

The transaction logic manages the flow of user interactions with the vending machine. Below is a snippet showing the main transaction loop:

```
while (true) {
    displayProducts(products, stock);
    int productIndex = selectProduct(products, stock);

    if (productIndex == -1) {
        break;
    }

    cout << "Selected product: " << products[productIndex].first
         << " - Price: " << products[productIndex].second << " EGP" << endl;

    int productPrice = products[productIndex].second;

    if (remainingMoney >= productPrice) {
        remainingMoney -= productPrice;
        stock[productIndex]--;
        cout << "Remaining balance used to buy the item. Remaining balance: "
             << remainingMoney << " EGP." << endl;
    } else {
        int amountNeeded = productPrice - remainingMoney;
        cout << "Insert " << amountNeeded << " EGP: ";
        int moneyInserted;
        cin >> moneyInserted;

        addMoneyToCurrency(currency, moneyInserted);

        int change = moneyInserted - amountNeeded;
        remainingMoney = change;
        stock[productIndex]--;

        if (remainingMoney > 0) {
            cout << "Remaining balance: " << remainingMoney << " EGP." << endl
                ;
        }
    }
}
```

Explanation:

- `remainingMoney`: Tracks the balance left after purchasing an item.
- `moneyInserted`: Records the additional money the user inputs if their current balance is insufficient.
- The program updates the product stock and currency inventory dynamically after each transaction.

Chapter 4

Algorithmic Approaches

This chapter presents the implementations and functionalities of the three algorithmic approaches for providing change.

4.1 Greedy Approach

```
bool provideChange_Greedy(map<int, int>& currency, int change)
{
    map<int, int> tempCurrency = currency;
    map<int, int> changeGiven;

    for (auto it = tempCurrency.rbegin(); it != tempCurrency.rend(); ++it)
    {
        int denom = it->first;
        int needed = min(change / denom, it->second);
        change -= needed * denom;
        tempCurrency[denom] -= needed;
        changeGiven[denom] += needed;
    }

    if (change > 0)
    {
        cout << "Sorry, exact change is not possible. Remaining: " << change
           << " EGP." << endl;
        currency = tempCurrency;
        return false;
    }

    cout << "Change given: ";
    for (auto it = changeGiven.begin(); it != changeGiven.end(); ++it)
    {
        if (it->second > 0)
        {
            cout << it->second << " x " << it->first << " EGP, ";
            currency[it->first] -= it->second;
        }
    }
```

```
    cout << endl;

    return true;
}
```

Explanation: The greedy approach aims to provide change using the largest denominations first. It iterates over the denominations in descending order and uses as many coins or bills as possible for each denomination until the remaining change becomes zero.

- **Key Variables:**
  - `denom`: The current denomination being processed.
  - `needed`: The number of coins/bills needed for the current denomination without exceeding the remaining change.
  - `tempCurrency`: A temporary copy of the vending machine's currency to simulate transactions.
  - `changeGiven`: Tracks the coins/bills used to provide change.
- The algorithm works well for denominations that allow optimal solutions when larger denominations are prioritized.
- **Limitations:** The greedy approach may fail to find an exact solution when the denominations require a smaller-denomination combination. For example:
  - Denominations: {1, 3, 4}, Change: 6
  - Greedy output: {4, 1, 1} (3 coins)
  - Optimal output: {3, 3} (2 coins)

4.2 Dynamic Programming Approach

```
bool provideChange_DP(map<int, int>& currency, int change) {

    vector<int> coins;
    vector<int> counts;
    for (auto it = currency.begin(); it != currency.end(); ++it) {
        coins.push_back(it->first);
        counts.push_back(it->second);
    }

    int n = coins.size();
    vector<vector<int>> dp(n + 1, vector<int>(change + 1, 0));
    vector<vector<int>> coinUsage(n + 1, vector<int>(change + 1, 0));

    dp[0][0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= change; j++) {
            dp[i][j] = dp[i - 1][j];
            coinUsage[i][j] = 0;

            if (j >= coins[i - 1] && dp[i][j - coins[i - 1]] > 0 && coinUsage[
                i][j - coins[i - 1]] < counts[i - 1]) {
```

```
                dp[i][j] += dp[i][j - coins[i - 1]];
                coinUsage[i][j] = coinUsage[i][j - coins[i - 1]] + 1;
            }
        }
    }

    if (dp[n][change] == 0) {
        cout << "Sorry, exact change is not possible. Remaining: " << change
           << " EGP." << endl;
        return false;
    }

    int remaining = change;
    cout << "Change given: ";
    for (int i = n; i > 0 && remaining > 0; i--) {
        while (remaining >= coins[i - 1] && coinUsage[i][remaining] > 0) {
            cout << coins[i - 1] << " EGP ";
            currency[coins[i - 1]]--;
            remaining -= coins[i - 1];
            coinUsage[i][remaining]--;
        }
    }
    cout << endl;

    return true;
}
```

Explanation:

- **Tabulation:** This is a bottom-up dynamic programming approach where a table (`dp`) is constructed iteratively. Each entry `dp[i][j]` represents the number of ways to make an amount `j` using the first `i` denominations.
- **Base Case:** The base case is `dp[0][0] = 1`, as no coins are required to make change for 0.
- **Transition:** For each denomination and amount, the formula:

$$dp[i][j] = dp[i-1][j] + dp[i][j - coins[i-1]]$$

  calculates whether using the current denomination provides additional ways to make the amount.
- **Backtracking:** After computing the table, the `coinUsage` array reconstructs the denominations used and updates the machine's currency inventory.

Visualization: Consider the following example:

- Denominations: {1, 2, 3}
- Counts: {5, 3, 2}
- Change: 5

The table evolves as follows:

Amount (j)	0	1	2	3	4	5
Using 0 coins	1	0	0	0	0	0
Using 1 EGP	1	1	1	1	1	1
Using 2 EGP	1	1	2	2	3	3
Using 3 EGP	1	1	2	3	4	5

4.3 Brute Force Approach

```
bool provideChange_Brute(map<int, int>& currency, int change)
{
    vector<int> denominations;
    for (auto it = currency.begin(); it != currency.end(); ++it)
    {
        if (it->second > 0)
        {
            denominations.push_back(it->first);
        }
    }

    sort(denominations.begin(), denominations.end());

    vector<int> bestCombination;
    int minCoins = INT_MAX;

    int totalCombinations = 1 << denominations.size();

    for (int mask = 0; mask < totalCombinations; mask++) {
        map<int, int> tempCurrency = currency;
        vector<int> tempCombination;
        int currentChange = change;

        for (int i = 0; i < denominations.size(); i++)
        {
            if (mask & (1 << i))
            {
                int denom = denominations[i];
                while (currentChange >= denom && tempCurrency[denom] > 0)
                {
                    currentChange -= denom;
                    tempCurrency[denom]--;
                    tempCombination.push_back(denom);
                }
            }
        }

        if (currentChange == 0 && tempCombination.size() < minCoins) {
            bestCombination = tempCombination;
            minCoins = tempCombination.size();
        }
    }
```

```
43
44      if (bestCombination.empty()) {
45          cout << "Sorry, exact change is not possible. Remaining: " << change
                << " EGP." << endl;
46          return false;
47      }
48
49      for (int i = 0; i < bestCombination.size(); i++) {
50          currency[bestCombination[i]]--;
51      }
52
53      cout << "Change given: ";
54      for (int i = 0; i < bestCombination.size(); i++) {
55          cout << bestCombination[i] << " EGP ";
56      }
57      cout << endl;
58
59      return true;
60  }
```

Explanation: The brute force approach explores all possible combinations of denominations to provide change. It uses a bitmask to generate all subsets of the denominations and evaluates each subset.

- **Key Variables:**
  - `denominations`: A sorted list of available denominations.
  - `mask`: Represents a subset of denominations as a binary value.
  - `tempCurrency`: Simulates the current state of the vending machine's currency.
  - `bestCombination`: Stores the optimal combination of coins/bills that minimize the total count.
- The algorithm tries every possible subset of denominations, subtracting them from the remaining change until the exact amount is achieved.
- **Advantages:** Always finds the optimal solution if it exists.
- **Limitations:**
  - Time complexity is $O(2^n \cdot c)$, where n is the number of denominations and c is the change amount.
  - Highly inefficient for large inputs due to its exponential growth.

Visualization: Consider the following example:

- Denominations: $\{1, 2, 5\}$
- Change: 7

The algorithm generates all subsets of the denominations:

- Subset: $\{5, 2\}$, Change left: $7 - 5 - 2 = 0$
- Subset: $\{1, 1, 1, 2, 2\}$, Change left: $7 - 1 - 1 - 1 - 2 - 2 = 0$
- Other subsets are pruned as they either exceed the change or leave an unresolvable remainder.

The optimal solution is found by evaluating all subsets.

Chapter 5

Complexity Analysis

5.1 Greedy Approach

Time Complexity: $O(m + k)$, where:

- m: The number of available denominations (iterations over denominations in descending order).
- k: The number of coins or bills used to make the change.

Space Complexity: $O(m)$ for temporary storage of the currency map.

Advantages:

- Runs in linear time with respect to the number of denominations.
- Simple to implement and efficient for most practical cases.

Limitations:

- Fails to find the optimal solution in cases where smaller denominations are required to minimize the number of coins. For example, with denominations {1, 3, 4} and change = 6, the greedy approach would select {4, 1, 1} (3 coins) instead of the optimal {3, 3} (2 coins).
- Requires denominations to be sorted in descending order for efficient implementation.

5.2 Dynamic Programming Approach

Time Complexity: $O(n \cdot c)$, where:

- n: The number of denominations.
- c: The target amount of change.

Space Complexity: $O(n \cdot c)$, for the 2D DP table and auxiliary arrays.

Advantages:

- Always finds the optimal solution (e.g., minimizes the number of coins required).
- Handles cases where the greedy approach fails, such as the {1, 3, 4} example mentioned earlier.

Limitations:

- Computationally expensive for large c or n due to the size of the DP table.
- Requires additional memory to store intermediate results.
- May not be practical for systems with constrained memory.

5.3 Brute Force Approach

Time Complexity: $O(2^n \cdot c)$, where:

- 2^n: The total number of subsets of denominations.
- c: The target change amount, as reducing c involves a loop over each subset.

Space Complexity: $O(n + c)$, for tracking subsets and intermediate results.

Advantages:

- Guarantees finding a solution (if one exists) as it explores all possible subsets of coins.
- Simple to implement for small inputs or academic purposes.

Limitations:

- Extremely inefficient for large numbers of denominations due to its exponential growth.
- Impractical for real-world vending machines or large-scale systems.

5.4 Comparison of Approaches

- The **greedy approach** is the fastest but fails to find the optimal solution in edge cases.
- The **dynamic programming approach** balances efficiency and correctness, making it the most practical for real-world applications.
- The **brute force approach**, while theoretically exhaustive, is only feasible for small inputs due to its exponential complexity.

Approach	Time Complexity	Space Complexity	Optimal Solution?
Greedy	$O(m + k)$	$O(m)$	No
Dynamic Programming	$O(n \cdot c)$	$O(n \cdot c)$	Yes
Brute Force	$O(2^n \cdot c)$	$O(n + c)$	Yes

Table 5.1: Comparison of Algorithmic Approaches for Change Optimization

Chapter 6

Conclusion

This report explored the change optimization problem in vending machines and implemented three algorithmic solutions. The results highlight:

- **Greedy Approach:** Efficient for cases where larger denominations suffice but fails in complex scenarios.
- **Dynamic Programming:** Balances efficiency and correctness, suitable for real-world systems.
- **Brute Force:** Guarantees an optimal solution but is computationally infeasible for large inputs.

The dynamic programming approach is the most practical for real-world applications, offering a compromise between complexity and optimality. Future work could explore hybrid methods or machine learning to dynamically predict and optimize change scenarios.