PROJECT Nº2

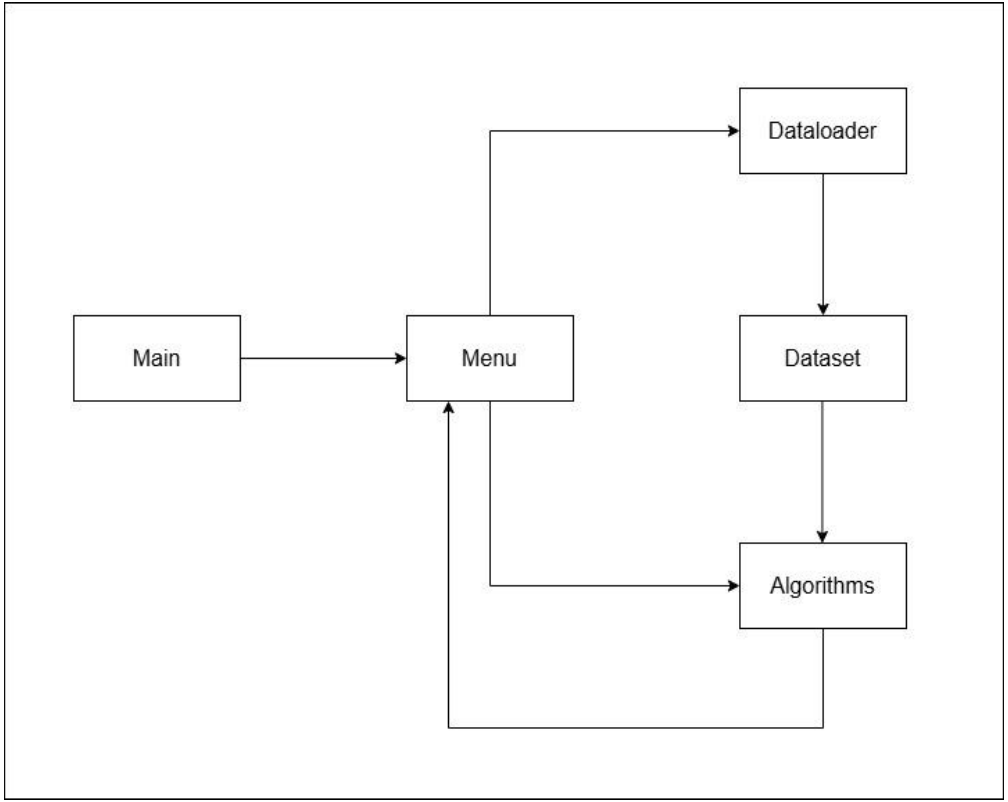
Algorithms Design

Project Overview

Project Goals

- **Objective**: Optimize the selection of pallets to load into trucks without exceeding their maximum weight.
- **Problem type**: 0/1 Knapsack Problem (maximize value with weight constraints).
- **Data**:
    - Input from **.csv files** (pallets and trucks).
    - **Values** = profit; **Weights** = pallet weights

Class Diagram



Reading the Dataset

Overview of the data loading process:

- The program reads data from **two CSV files** using **ifstream**.
- **load_data_pallets()** parses a **pallet dataset**, extracting **pallet ID**, **profit**, and **weight** from each row, converting them to integers, and storing them as **Pallet objects** in a vector.
- **load_data_trucks()** reads **truck capacity** and **max pallet count** from another CSV and returns a **Truck object**.
- Both functions skip headers and handle file opening errors gracefully.

```cpp
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include "dataset.h"
#include "data_loader.h"
using namespace std;

vector<Pallet> load_data_pallets(const string& filename){...}

Truck load_data_trucks(const string& filename){...}
```

Brute Force (Exhaustive Search)

Function: KnapsackBF()

- **Approach:**
  - Tests all 2^n combinations of items.
  - Keeps track of the best solution considering:
    - Highest total value.
    - Fewest items.
    - Lowest sum of pallet IDs.
- **Complexity:** $O(2^n)$
- **Pros:** Guarantees optimality.
- **Cons:** Very slow for large inputs.

```
unsigned int knapsackBF(unsigned int values[], unsigned int weights[], unsigned int n, unsigned int maxWeight, bool usedItems[]) {
    bool curCandidate[4097];  // current solution candidate being built
    unsigned int maxValue = 0;  // value of the best solution found so far
    unsigned int bestNumItems = n + 1; // Initialize to a value greater than the max number of items (n)
    unsigned int bestSumPallets = UINT_MAX;  // Initialize to a large number
    bool foundSol = false;

    // Prepare the first candidate (no items selected initially)
    for (unsigned int i = 0; i < n; i++){...}

    // Iterate over all the candidates
    while (true){...}

    // Output the selected items (pallets)
    for (unsigned int i = 0; i < n; i++){...}

    return maxValue;
}
```

Greedy

Function: KnapsackGreedy()

- **Approach**:
    - Sorts pallets by **profit-to-weight ratio**
    - Adds items until **truck is full**
- **Pros**:
    - Very **fast**
    - Useful for **approximate solutions**
- **Cons**:
    - Does not guarantee **optimality**
- **Use case**: Good for **real-time decisions** or **large datasets**

```cpp
unsigned int knapsackGreedy(vector<Pallet> pallets, unsigned int n, unsigned int maxWeight, bool usedItems[]) {
    sort(pallets.begin(), pallets.end(), compare);
    unsigned int maxValue = 0;
    unsigned int idx = 0;

    while (maxWeight > 0 && idx < n) {
        if (pallets[idx].weight <= maxWeight) {  // check if pallet fits
            maxValue += pallets[idx].profit;
            maxWeight -= pallets[idx].weight;
            usedItems[pallets[idx].pallet] = true;
        }
        idx++;
    }

    cout << "Selected pallets IDs:" << endl;
    for (unsigned int i = 0; i < n; i++) {  // print starting at 0
        if (usedItems[i]) {
            cout << i + 1 << endl;
        }
    }

    return maxValue;
}
```

Dynamic Programming (2 versions)

Dynamic Programming (2 versions):

- **Functions**: KnapsackDP(), KnapsackDP1()
- **Approach**:
  - Bottom-up table-based solution.
  - Builds a DP table **maxValue[i][k]** where:
    - i = item index.
    - k = current weight capacity.
  - **Backtracking**: Determines which items were selected.
- **Complexity**: O(n × **W**)
  (n = number of items, W = maxWeight)
- **Improvement over BF**: Handles larger datasets efficiently.

```
unsigned int knapsackDP(unsigned int values[], unsigned int weights[], unsigned int n, unsigned int maxWeight, bool usedItems[]){...}

unsigned int knapsackDP1(unsigned int values[], unsigned int weights[], unsigned int n, unsigned int maxWeight, bool usedItems[]){...}
```

ILP

Branch and Bound (ILP-inspired)

- **Function**: knapsackILP()
- **Approach**:
  - Branches into **"include"** or **"exclude"** decisions for each item.
  - Computes **upper bound** on max achievable value.
  - Prunes branches that cannot beat the **best solution**.
- **Node structure**: Tracks level, value, weight, decisions.
- **Complexity**: Depends on pruning efficiency.
- **Advantages**: Often much faster than brute force with similar optimality guarantees.

```cpp
unsigned int knapsackILP(unsigned int values[], unsigned int weights[], unsigned int n, unsigned int maxWeight, bool usedItems[]) {
    struct Node {
        int level;
        unsigned int value;
        unsigned int weight;
        bool decisions[20];
        double bound;
    };

    auto computeBound = [&](Node& node){...};

    Node bestNode;
    bestNode.value = 0;
    bestNode.weight = 0;
    bestNode.level = 0;
    bestNode.bound = 0;
    for (unsigned int i = 0; i < n; i++) bestNode.decisions[i] = false;

    std::vector<Node> stack;
    Node root;
    root.value = 0;
    root.weight = 0;
    root.level = 0;
    for (unsigned int i = 0; i < n; i++) root.decisions[i] = false;
    root.bound = computeBound([&] root);
    stack.push_back(root);

    while (!stack.empty()){...}

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

    return bestNode.value;
}
```

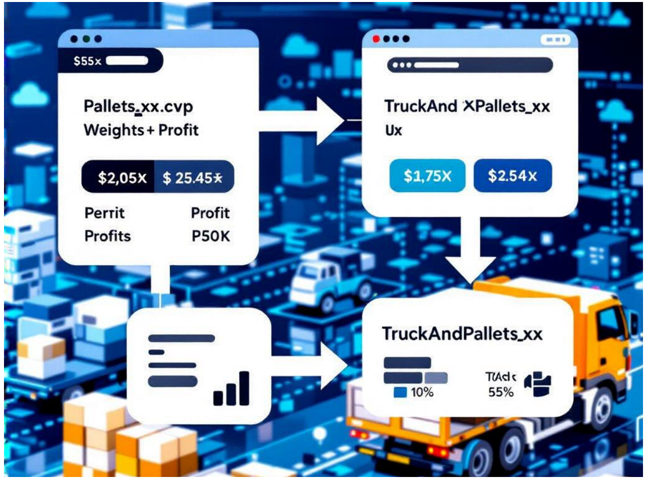
Algorithms Comparison

Algorithm	Optimal Solution	Speed	Space Complexity
Brute Force	✅ Yes	❌ Very Slow	❌ Exponential (`O(2^n)`)
Dynamic Prog.	✅ Yes	✅ Fast	⚠️ Medium (`O(n·W)`)
Greedy	❌ No (Approximate)	✅ Very Fast	✅ Low (`O(n)`)
ILP (B&B)	✅ Yes	⚠️ Medium	⚠️ Medium (`O(n)` stack)

Dataset Integration

Overview of dataset integration processes:

- **Reads .csv files** for pallets and truck data.
- **Files**:
  - **Pallets_xx.csv**: Each pallet's weight and profit.
  - **TruckAndPallets_xx.csv**: Truck capacity and relevant pallet IDs.
- Uses **dynamic filenames** for different datasets.



User Interface & Example of Use

Interface Description:

- Console-based menu interface, allowing users to:
  - Select a dataset (**Pallets.csv** & **TruckAndPallets.csv**)
  - Choose between four algorithmic strategies to solve the **knapsack problem**.
- **Example of Use:**
- User selects dataset 3.
- Program loads:
  - A list of pallet weights and profits.
  - The truck's maximum capacity.
- User selects "**ILP Approach**".
- Program outputs the best combination of pallets to maximize profit while staying within capacity.

```
Welcome to the menu...
Please select dataset: 3
Please select algorithm:
1. Brute-force
2. Dynamic Programming
3. Approximation
4. ILP
```

Highlighted Functionality

What we’re most proud of:

- **The ILP-Inspired Branch and Bound Algorithm**
    - Custom-built **ILP-like algorithm** using Branch and Bound.
    - Efficiently handles large datasets by pruning suboptimal branches using **fractional upper bounds**.
    - Achieves **near-optimal results** in significantly reduced time compared to **brute-force**.

- **Key Highlights**:
    - Uses a custom **Node structure** for tree-based decisions.
    - Smart pruning based on a computed **upper bound (bound)** → avoids unnecessary computation.
    - Tracks best solution across all paths using **bestNode**.

Difficulties & Teamwork

Difficulties & Teamwork

- **Main Difficulties Faced**:
  - ILP algorythm
  - Understanding how to efficiently implement the Branch and Bound technique and upper bound estimation.

- **Team Member Participation**:
  - Everyone contributed equally to:
    - Designing the main logic and data structures.
    - Implementing and testing each algorithm.
    - Debugging and improving the ILP solution.
    - Preparing the interface and menu navigation.