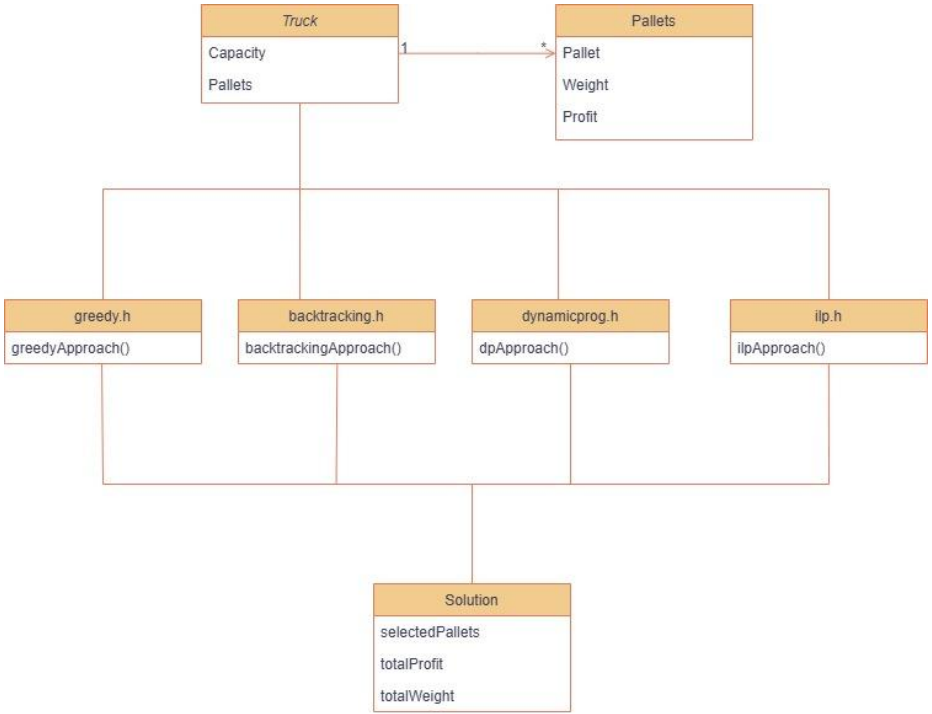
Otimização da Embalagem de Paletes de Camiões de Distribuição

Grupo 7, Turma 1:
Raquel Fernandes - 202207134
Nicolas Ramos - 202304442
Hugo Alves - 202305395

Regência: Pedro Diniz
Docente: Iohan Soares

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Diagrama de Classes



Descrição da Leitura do Conjunto de Dados Fornecido

O *parsing* dos arquivos .csv é feito da seguinte maneira:

- O arquivo é aberto e lido linha por linha (exceto pela primeira). Para cada linha, adicionamos um novo *Pallet* a um vetor contendo os mesmos, preenchendo os dados do *struct*.

Funcionalidades Implementadas: *Greedy*

```cpp
std::pair<int, std::vector<int>> greedyApproach(std::vector<Pallet> &pallets, int capacity) {
    std::vector<std::pair<double, int>> ratios( n: pallets.size());
    for (int i = 0; i < pallets.size(); ++i) {
        ratios[i] = { x: static_cast<double>(pallets[i].profit) / pallets[i].weight, &: i};
    }

    std::sort( first: ratios.rbegin(), last: ratios.rend());

    int totalProfit = 0;
    std::vector<int> selectedPallets;

    for (const auto &ratio : pair<double, int> const & : ratios) {
        int index = ratio.second;
        if (pallets[index].weight <= capacity) {
            selectedPallets.push_back(pallets[index].id);
            totalProfit += pallets[index].profit;
            capacity -= pallets[index].weight;
        }
    }

    return { &: totalProfit, &: selectedPallets};
```

- ➢ Complexidade Temporal: O(n log n)
- ➢ Complexidade Espacial: O(n)

Funcionalidades Implementadas: *Dynamic Programming*

```cpp
std::pair<int, std::vector<int>> dpApproach(const std::vector<Pallet>& pallets, int capacity) {
    int n = pallets.size();
    std::vector dp( n: n + 1,  value: std::vector<int>( n: capacity + 1,  value: 0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 0; w <= capacity; ++w) {
            if (pallets[i - 1].weight <= w) {
                dp[i][w] = std::max(dp[i - 1][w], dp[i - 1][w - pallets[i - 1].weight] + pallets[i - 1].profit);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    std::vector<int> selected;
    int w = capacity;
    for (int i = n; i >= 1; --i) {
        if (dp[i][w] != dp[i - 1][w]) {
            selected.push_back(pallets[i - 1].id);
            w -= pallets[i - 1].weight;
        }
    }

    return { &: dp[n][capacity],  &: selected};
```

Legenda: *n* é o número de paletes e *W* é a capacidade do camião.

- ➢ Complexidade Temporal: O(n * W)
- ➢ Complexidade Espacial: O(n * W)

Funcionalidades Implementadas: *Integer Linear Programming*

```cpp
std::pair<int, std::vector<int>> ilpApproach(std::vector<Pallet> &pallets, int capacity) {
    std::pair<int, std::vector<int>> result;
    std::ofstream file( s: "../src/input.txt");
    file << pallets.size() << '\n' << capacity << '\n';
    for (int i = 0; i < pallets.size(); i++)
        file << pallets[i].weight << ' ';

    file << '\n';
    for (int i = 0; i < pallets.size(); i++)
        file << pallets[i].profit << ' ';

    file.close();
    int ret = system( Command: "../venv/bin/python ../src/knapsack_solver.py ../src/input.txt ../src/output.txt");

    if (ret) {
        std::cerr << "Solver failed\n";
        return {};
    }

    std::ifstream fileIn( s: "../src/output.txt");

    fileIn >> result.first;
    int weight;
    fileIn >> weight;

    int id;
    while (fileIn >> id) {
        result.second.push_back(id);
    }

    return result;
```

No pior caso:

- Complexidade Temporal: O(2^n)
- Complexidade Espacial: O(2^n)

Funcionalidades Implementadas: *Brute Force + Back Tracking*

```
::pair<int, std::vector<int>> backtrackingApproach(const std::vector<Pallet> &pallets, int capacity) {
 std::vector<int> left;
 std::vector<int> right;
 std::vector<Pallet> sorted_pallets = pallets;
 std::sort( first: sorted_pallets.begin(), last: sorted_pallets.end());
 int element = 0;

 if ( element >= pallets.size() || sorted_pallets[element].weight > capacity) {
     return { x: 0, &: left};
 }

 right.push_back(sorted_pallets[element].id);

 std::pair<int, std::vector<int>> resultLeft = backtrackingAux( pallets: sorted_pallets, currentValue: 0, remainingCapacity: capacity, state: left, element: element + 1);
 std::pair<int, std::vector<int>> resultRight = backtrackingAux( pallets: sorted_pallets, currentValue: sorted_pallets[element].profit, remainingCapacity: capacity - sorted_pallets[element].weight, state: right, element: element + 1

 if (resultLeft.first > resultRight.first) {
     return { &: resultLeft.first, &: resultLeft.second};
 }
 return { &: resultRight.first, &: resultRight.second};
```

- Complexidade Temporal: O(2^n) - pior caso
- Complexidade Espacial: O(n^2)

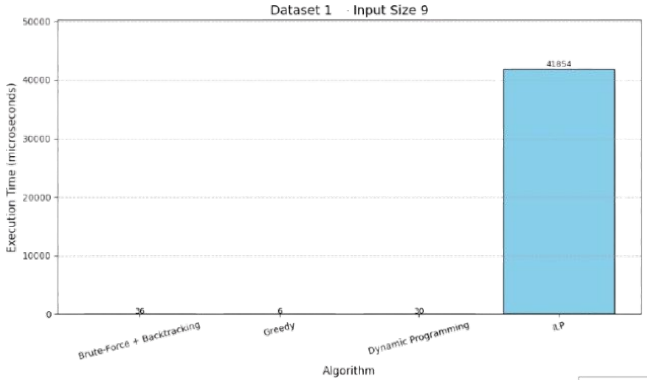
Descrição da Interface do Utilizador

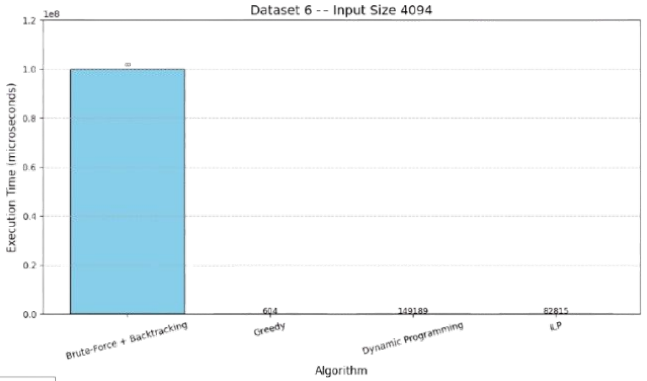
1.
```
Choose an option:
[1] Load dataset
[2] Choose algorithmic approach
[3] See loaded settings
[4] Run Program
[5] Exit Program
1

Insert dataset id:
3

Choose an option:
[1] Load dataset
[2] Choose algorithmic approach
[3] See loaded settings
[4] Run Program
[5] Exit Program
2
```

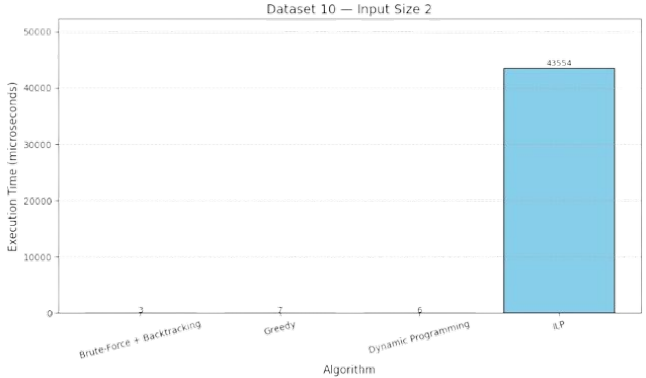
2.
```
Choose an approach:
[1] Brute-Force + Backtracking
[2] Greedy
[3] Dynamic Programming
[4] ILP
2

Choose an option:
[1] Load dataset
[2] Choose algorithmic approach
[3] See loaded settings
[4] Run Program
[5] Exit Program
3
```

3.
```
Loaded truck:
TruckAndPallets_03.csv
Loaded pallets:
Pallets_03.csv

Approach selected:
Greedy

Choose an option:
[1] Load dataset
[2] Choose algorithmic approach
[3] See loaded settings
[4] Run Program
[5] Exit Program
4

Total profit: 40
Pallets: 9 8 7 6
```

Comparação global do algoritmo implementado







Principais Dificuldades

Em geral, o projeto foi mais simples que o primeiro, não havendo muitas dificuldades no foco principal do mesmo (resolução do problema com as *approaches* pretendidas).

A principal dificuldade foi gerar os gráficos que dizem respeito ao tempo de execução de cada *approach* para cada *dataset*.