Logística Urbana para Entrega de Mercadorias

Desenho de Algoritmos, LEIC, 2º Ano

Fábio Araújo de Sá, up202007658@fe.up.pt
Francisco João Gonçalves Calado Araújo, up201806326@fe.up.pt
Marcos William Ferreira Pinto, up201800177@fe.up.pt

Porto, Abril de 2022

Descrição do problema

Uma empresa de entregas de mercadorias pretende ver o seu negócio maximizado segundo os seguintes cenários:

1. Otimização do número de estafetas;
2. Otimização do lucro da empresa, ao entregar encomendas usando carrinhas de menor custo;
3. Otimização das entregas expresso, melhorando o tempo de entrega médio;

O seguinte projeto implementa soluções para cada um dos casos usando alguns algoritmos estudados, como *greedy*, *bin-packing*, *knapsack* e *job scheduling*.

Cenário 1 - Formalização

Input:
Conjunto de Carrinhas, C, em que:
- $C_v[i]$ é a capacidade de volume da carrinha i, em unidades arbitrárias de volume, $i \in [0..N]$
- $C_p[i]$ é a capacidade de peso da carrinha i, em unidades arbitrárias de peso, $i \in [0..N]$

Conjunto de Encomendas, E, em que:
- $E_v[j]$ é o volume da encomenda j, em unidades arbitrárias de volume, $j \in [0..N]$
- $E_p[j]$ é o peso da encomenda j, em unidades arbitrárias de peso, $j \in [0..N]$

Output:
Subconjunto $C' = \{C'_1, C'_2, \ldots, C'_N\}$ de Carrinhas. A Carrinha C'_i, $i \in [0..N]$, é usada para transportar um subconjunto de $E' = \{E'_1, E'_2, \ldots, E'_T\}$, as encomendas entregues.

Restrições:

$$\forall c \in C', c_V \geq 0 \wedge c_P \geq 0 \wedge \sum_{j=0}^{T} c[j]_V \leq c_V \wedge$$

$$\wedge \sum_{j=0}^{T} c[j]_P \leq c_P \wedge \forall e \in E', e_V \geq 0 \wedge e_P \geq 0$$

Objetivo: min (C'), se E' = E $\vee$ max (E'), se E' ≠ E

Cenário 1 - Algoritmos

Algoritmo **duplamente *greedy***, com ordenação dos carros por ordem decrescente e as encomendas por ordem crescente. Assim há garantia que se a encomenda **i** não cabe num determinado carro, a encomenda **i + 1** também não cabe. Esta abordagem transforma um algoritmo que poderia ser O(ExC) em **O(E+C)**.

A ordenação inicial é feita **pelo peso ou pelo volume**, mediante o parâmetro onde as encomendas tenham **menor desvio absoluto máximo**. Esta otimização garante redução de até um carro no resultado final.

```
if (desvioAbsMaxPeso >= desvioAbsMaxVolume) {
    this->trucks.sort(byMaxVolumeDesc);
    this->packages.sort(byVolumeAsc);
} else {
    this->trucks.sort(byMaxWeightDesc);
    this->packages.sort(byWeightAsc);
}

list<Package> notDelivered = {};
list<Truck>::iterator j = trucks.begin();
for (list<Package>::iterator i = packages.begin(); i != packages.end(); i++
{
    if ((*j).addPackage(*i)) continue;
    else if (j == trucks.end()) {
        (*i).addPriority();
        notDelivered.push_back(*i);
    } else {
        j++;
        i--;
    }
}
```

Cenário 1 - Complexidade

Seja C o número de Carrinhas e E o número de Encomendas, onde E ≥ C:

$$\mathbf{T(C, E)} = O (E + C \log C + E \log E + E + C + E) \approx \mathbf{O (E \log E)}$$

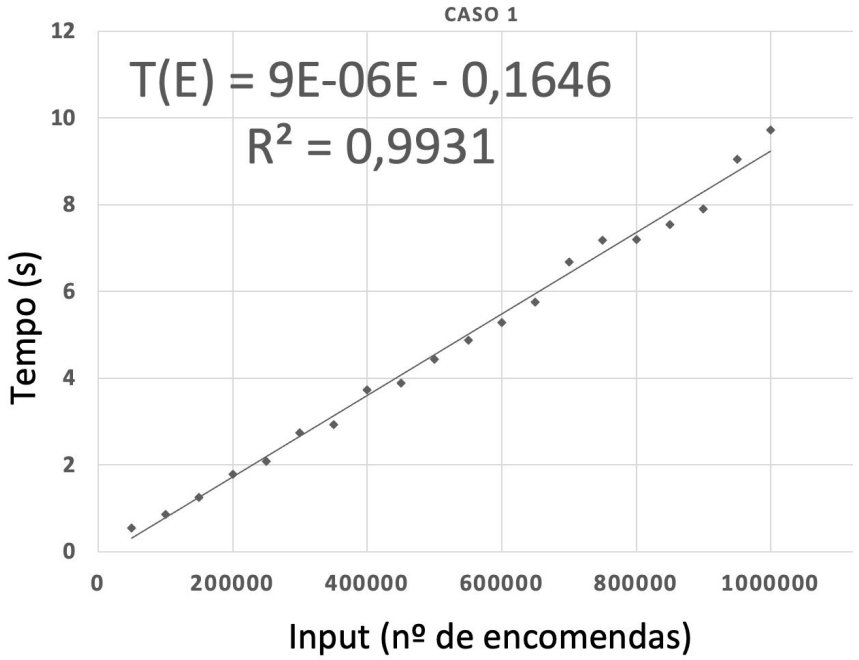
- E: Determinação do desvio absoluto máximo (peso, volume)
- $C \log C + E \log E$: Ordenação das duas listas, usando o algoritmo *IntroSort* da STL
- E: Inserção de encomendas nas carrinhas (*binpacking* modificado)
- $C + E$: Reescrita em ficheiros das encomendas que não puderam ser entregues

$$\mathbf{S(C, E)} = \mathbf{O (E)}$$

- E: Lista auxiliar da encomendas que não puderam ser entregues

Cenário 1 - Avaliação empírica

Input, E (número de encomendas)	Tempo de Execução, T (segundos)
50000	0,548
100000	0,865
150000	1,254
200000	1,782
250000	2,077
300000	2,749
350000	2,923
400000	3,735
450000	3,886
500000	4,442
550000	4,869
600000	5,287
650000	5,759
700000	6,683
750000	7,185
800000	7,203
850000	7,534
900000	7,906
950000	9,052
1000000	9,723



Cenário 2 - Formalização

Input:
Conjunto de Carrinhas, C, em que:

- $C_v[i]$ é a capacidade de volume da carrinha i, em unidades arbitrárias de volume, i ∈ [0..N]
- $C_p[i]$ é a capacidade de peso da carrinha i, em unidades arbitrárias de peso, i ∈ [0..N]
- $C_D[i]$ é a despesa da carrinha i, em unidades arbitrárias monetárias, i ∈ [0..N]

Conjunto de Encomendas, E, em que:

- $E_v[j]$ é o volume da encomenda j, em unidades arbitrárias de volume, j ∈ [0..N]
- $E_p[j]$ é o peso da encomenda j, em unidades arbitrárias de peso, j ∈ [0..N]
- $E_R[j]$ é a recompensa da encomenda j, em unidades arbitrárias monetárias, j ∈ [0..N]

Output:
Subconjunto C' = {C'_1, C'_2, …, C'_N} de Carrinhas. A Carrinha C'_i , i ∈ [0..N], é usada para transportar um subconjunto de E' = {E'_1, E'_2, …, E'_T}, as encomendas entregues.

Restrições:

$$\forall c \in C', c_V \geq 0 \wedge c_P \geq 0 \wedge c_D \geq 0 \wedge \sum_{j=0}^{T} c[j]_V \leq c_V \wedge \sum_{j=0}^{T} c[j]_P \leq c_P \wedge$$
$$\wedge \forall e \in E', e_R \geq 0 \wedge e_V \geq 0 \wedge e_P \geq 0$$

Objetivo:

$$max \left(\sum_{i=0}^{N} \left(\left(\sum_{j=0}^{T} C'[i][j]_R \right) - C'[i]_D \right) \right)$$

Cenário 2 - Algoritmos

É percorrido a lista de carros utilizando o algoritmo de Knapsack para cada um.

O algoritmo de Knapsack usa a técnica de ¨memoization¨ para guardar qual a melhor solução até um determinado momento. Após percorrer a lista de encomendas o melhor resultado é guardado na estafeta.

O algoritmo é interrompido quando um carro não gera lucro, descartando a entrega dele e passando todas as encomendas restantes para o dia seguinte.

A lista de carros é ordenada no início de forma a usar sempre os carros de menor custo.

```
list<Package>::iterator pIt = currentPackages.begin();
// Computes the maximum value that can be reached varying the number of elements and the capacity
for (unsigned int n = 1; n <= currentPackages.size(); n++)
{
    for (unsigned int capacity = 1; capacity <= currentTruck.getMaxWeight(); capacity++)
    {
        if (pIt->getWeight() <= capacity)
        {
            if (((int)((*pIt).getWeight()) + knapsack[n - 1][capacity - (int)((*pIt).getWeight())]) > knapsack[n - 1][capacity])
                knapsack[n][capacity] = pIt->getReward() + knapsack[n - 1][capacity - pIt->getWeight()];
            else
                knapsack[n][capacity] = knapsack[n - 1][capacity];
        }
    }
    advance(pIt, 1);
}
```

```
for (int i = currentPackages.size(); i > 0 && res > 0; i--)
{
    --pIt;
    if (res == knapsack[i - 1][weight])
        continue;
    else
    {
        if ((*pIt).getVolume() + currentTruck.getActualVolume() <= currentTru
        {
            currentTruck.addPackage(*pIt);
            selectedIt.push_back(pIt);
            packagesGain += (int)((*pIt).getReward());

            res -= pIt->getReward();
            weight -= pIt->getWeight();
        }
        else
            break;
    }
}

if ((packagesGain - (int)currentTruck.getCost()) >= 0)
{
    for (auto it : selectedIt)
        currentPackages.erase(it); // TEMPORARY - deleteing items from list
    return true;
}
else
{
    currentTruck.clearPackages();
    return false;
}
```

Cenário 2 - Complexidade

Seja C o número de Carrinhas e E o número de Encomendas, onde E ≥ C:

T(C, E) = O (E + C log C + E log E + C*C*E + E) ≈ **O (C*C*E)**

Determinação do desvio absoluto máximo (peso, volume)

Ordenação das duas listas, usando o algoritmo *IntroSort* da STL

Percorrer a lista de carrinhas

Algoritmo de Knapsack

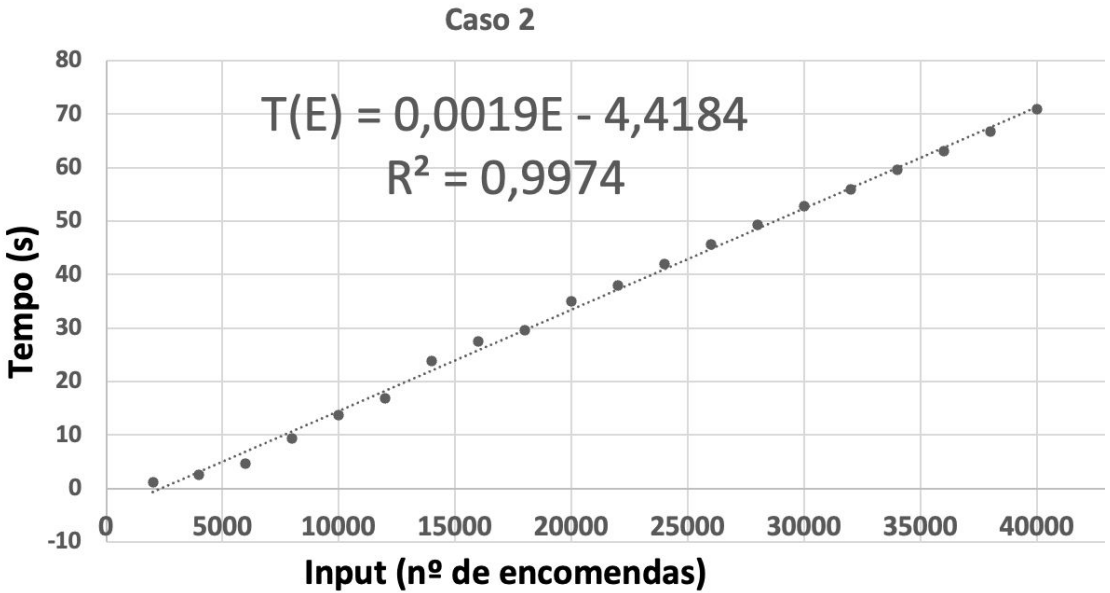
Reescrita em ficheiros das encomendas que não puderam ser entregues

S(C, E) = O (?) ≈ **O (E*C)**

Tamanho do Array 2D usado para fazer knapsack

Cenário 2 - Avaliação empírica

Input, E (número de encomendas)	Tempo de Execução, T (segundos)
2000	1,189
4000	2,539
6000	4,587
8000	9,442
10000	13,669
12000	16,953
14000	23,788
16000	27,441
18000	29,536
20000	34,987
22000	37,938
24000	42,07
26000	45,679
28000	49,237
30000	52,778
32000	55,988
34000	59,581
36000	63,156
38000	66,769
40000	70,941



Cenário 3 - Formalização

Input:

Intervalo de tempo, T, em segundos

Conjunto de Encomendas, E = {E_1, E_2, …, E_N}, em que:

- E_D[j] é a duração da entrega da encomenda j, em unidades arbitrárias de tempo, j ∈ [1..N]

Output:

Subconjunto E' = {E'_1, E'_2, …, E'_N}, as encomendas entregues no intervalo de tempo T.

Restrições:

$$\sum_{j=1}^{N} E'[j]_D \leq T \wedge T \geq 0 \wedge \forall e \in E', e_D \geq 0$$

Objetivo:

$$min \left(\sum_{j=1}^{N} \sum_{i=1}^{j} E'[i]_D \right)$$

Cenário 3 - Algoritmos

Algoritmo ***greedy***, que garante a solução ótima, com ordenação das encomendas expresso por ordem ascendente de duração de entrega.

Ao entregar as **encomendas de menor duração primeiro**, garante-se uma maior quantidade de entregas efetuadas num determinado período T e, para esse período, uma **minimização da média do tempo de entrega**.

```
this->expressoPackages.sort(byDurationAsc);

int timeLeft = T;
unsigned int averageTime = 0;
unsigned int numberOfPackagesDelivered = 0;
list<Package> notDelivered = {};
for (auto p : expressoPackages) {
    if ((timeLeft - p.getDuration()) >= 0) {
        file << "\tPackage " << p;
        timeLeft -= p.getDuration();
        averageTime += p.getDuration();
        numberOfPackagesDelivered++;
    } else notDelivered.push_back(p);
}
```

Cenário 3 - Complexidade

Seja E o número de Encomendas:

$$\mathbf{T(E)} = O(E \log E + E + E) \approx \mathbf{O(E \log E)}$$

Ordenação da lista de encomendas, usando o algoritmo *IntroSort* da STL

Cálculo do tempo médio das entregas

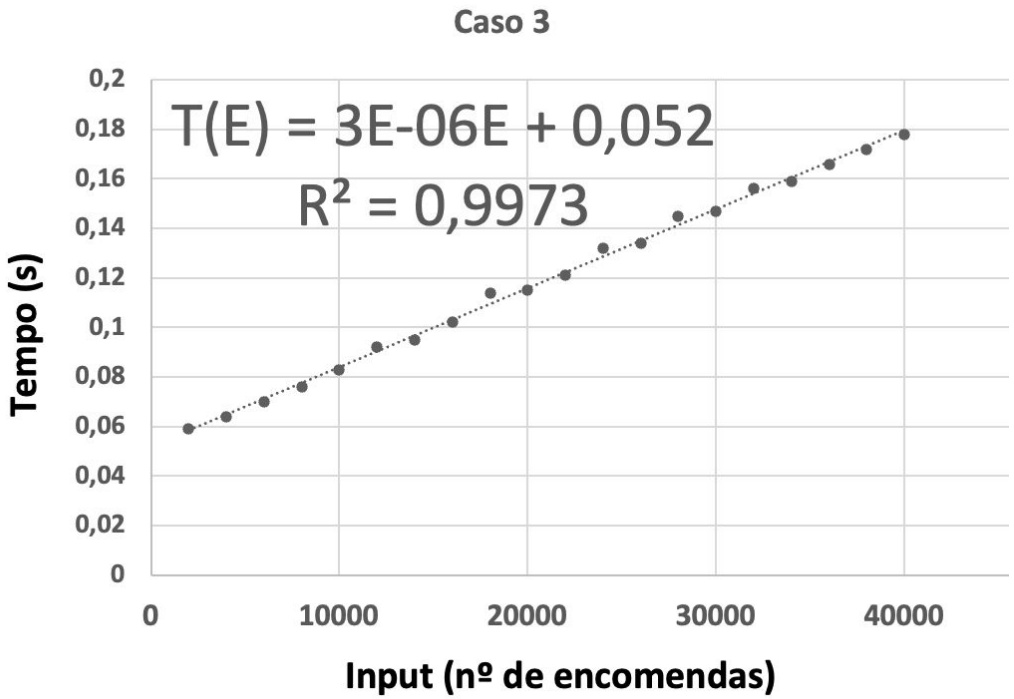
Reescrita em ficheiros das encomendas que não puderam ser entregues

$$\mathbf{S(E)} = \mathbf{O(E)}$$

Lista auxiliar da encomendas que não puderam ser entregues

Cenário 3 - Avaliação empírica

Input, E (número de encomendas)	Tempo de Execução, T (segundos)
2000	0,059
4000	0,064
6000	0,07
8000	0,076
10000	0,083
12000	0,092
14000	0,095
16000	0,102
18000	0,114
20000	0,115
22000	0,121
24000	0,132
26000	0,134
28000	0,145
30000	0,147
32000	0,156
34000	0,159
36000	0,166
38000	0,172
40000	0,178



Funcionalidades extra

- Aumento da prioridade da encomenda quando não pode ser entregue;
- Guardar o ficheiro de resultados com data e horário no nome;
- Medir a eficiência de cada operação realizada;
- Restrição do tempo de entrega das encomendas expresso;
- Escolher nome do ficheiro de output;

Solução Algorítmica a destacar

Tratamento dos dados iniciais para optimização da solução do problema através de uma análise estatística, visando a melhor solução e desempenho para a obtenção dos resultados.

A ordenação inicial é feita **pelo peso ou pelo volume**, mediante o parâmetro onde as encomendas tenham **menor desvio absoluto máximo**. Esta otimização garante redução de até um carro no resultado final.

Dificuldades encontradas

- Além de calcular a melhor solução para uma estafeta (utilizando o algoritmo de Knapsack) como guardar as encomendas que fazem parte de solução;

Autoavaliação

- Fábio Araújo de Sá, **33,(3)%**
- Francisco João Gonçalves Calado Araújo, **33,(3)%**
- Marcos William Ferreira Pinto, **33,(3)%**