

Logística Urbana para Entrega de Mercadorias

Grupo 45:

- Lucas Sousa - up202004682
- João Teixeira - up201900190
- Jose Pineda - up202111200

Descrição do Problema

Uma Empresa de entregas tem estafetas/carrinhas e encomendas a entregar.

Precisa de uma plataforma para as distribuir tendo em conta alguns cenários:

- Cenário 1: otimização do número de estafetas
- Cenário 2: otimização do lucro da empresa
- Cenário 3: otimização das entregas expresso



Cenário 1 - Formalização

O primeiro exercício consistiu em procurar a melhor estratégia para tentar minimizar o número de estafetas que uma empresa necessita ao longo de um dia, maximizando o número de encomendas que cada estafeta transporta nesse dia.

Neste caso temos de ter em conta o peso e o volume de cada encomenda e as correspondentes capacidades das carrinhas.



Cenário 1 - Descrição dos Algoritmos

Para desenvolver o primeiro cenário, aplicamos a técnica Offline Best Fit Bin Packing (Greedy), que consiste em escolher a melhor opção possível no momento atual, tendo em consideração escolhas futuras.

Para isso, em primeiro lugar, ordenamos o vetor de entregas e o vetor de carrinhas com base no seu peso, e em caso de igualdade o volume, e iteramos pelo vetor de entregas.



Cenário 1 - Descrição dos Algoritmos

Uma vez ordenados os vetores, procurou-se colocar cada entrega na melhor carrinha que ainda tivesse capacidade suficiente, para que cada carrinha usada estivesse o mais carregada possível.

Por fim, quando sabemos a melhor distribuição para cada carrinha, quantas são necessárias para fazer as entregas, deixando os mais pesados para o dia seguinte.



Cenário 1 - Descrição do Algoritmo

```
vector<Combination> cenario1(vector<Van> vans, vector<Delivery> deliveries){  
    vector<Combination> ret, ret2;  
    vector<Delivery> empty;  
  
    //sort deliveries by weight and then volume  
    std::sort(deliveries.begin(), deliveries.end(), sortByPWeightVolume);  
  
    //sort vans by weight and then by volume  
    std::sort(vans.begin(), vans.end());  
  
    //create combinations with all vans so that indices match  
    for(auto van: vans){  
        ret.push_back(Combination(van, empty, 0));  
    }  
  
    for(int i = 0; i < deliveries.size(); i++){  
        //find best van that can accomodate the delivery  
        int v = 0;  
        for (v; v < vans.size(); v++){  
            if(ret[v].getVan().getWeight() >= deliveries[i].getWeight() && ret[v].getVan().getVolume() >= deliveries[i].getVolume()) break;  
  
            //v is now the index for the best van for the item  
            int maxV = ret[v].getVan().getVolume(), maxW = ret[v].getVan().getWeight();  
  
            ret[v].addDelivery(deliveries[i]);  
            //update weight and volume available to the van  
            ret[v].setVan(Van(maxV-deliveries[i].getVolume(), maxW - deliveries[i].getWeight(), 0));  
        }  
  
        //count number of deliveries  
        int cnt = 0;  
        for(auto comb: ret){  
            if(comb.getDeliveries().size() !=0){  
                ret2.push_back(comb);  
            }  
        }  
    }  
    return ret2;  
}
```

Cenário 1 - Análise de Complexidade

Complexidade Temporal: $O(n^2)$

A complexidade do problema seria $O(n^2)$, pois para cada entrega, é necessário determinar qual a carrinha mais adequada para a carregar.

No pior caso, a melhor carrinha que ainda tem espaço para uma dada entrega estará no final do vetor.



Cenário 1 - Resultados

```
|| Delivery:   Volume: 19   Weight: 29   Reward: 1523   Duration: 353
|| Delivery:   Volume: 22   Weight: 29   Reward: 1448   Duration: 487
|| Delivery:   Volume: 22   Weight: 29   Reward: 1500   Duration: 828
Van:  Volume: 53   Weight: 81   Cost: 0
|| Delivery:   Volume: 23   Weight: 29   Reward: 1789   Duration: 900
|| Delivery:   Volume: 23   Weight: 29   Reward: 957    Duration: 387
|| Delivery:   Volume: 23   Weight: 29   Reward: 1232   Duration: 389
|| Delivery:   Volume: 26   Weight: 29   Reward: 267    Duration: 777
|| Delivery:   Volume: 26   Weight: 29   Reward: 610    Duration: 917
|| Delivery:   Volume: 26   Weight: 29   Reward: 1233   Duration: 434
|| Delivery:   Volume: 29   Weight: 29   Reward: 1019   Duration: 326
|| Delivery:   Volume: 29   Weight: 29   Reward: 763    Duration: 971
|| Delivery:   Volume: 29   Weight: 29   Reward: 969    Duration: 833
```

```
number of vans 28
number of deliveries 450
```


Cenário 2 - Formalização

Neste cenário, cada estafeta, para além das restrições em peso e volume, tem um custo associado. De forma semelhante, cada encomenda, para além de peso e volume, tem uma recompensa associada.

Pretende-se distribuir as encomendas pelos estafetas de forma a maximizar o lucro da empresa. (Diferença entre a recompensa total resultante da entrega das encomendas escolhidas e entre o custo total dos estafetas / carrinhas escolhidos

.

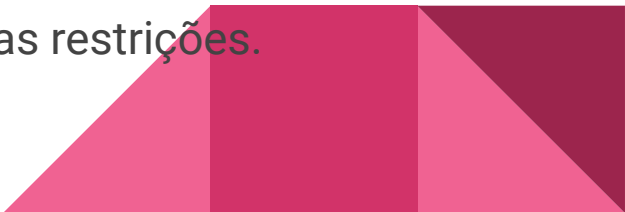


Cenário 2 - Descrição do Algoritmo

De forma a tratar o cenário 2, usamos uma solução mista escolhendo, em cada iteração o estafeta mais barato (Greedy) e determinando quais das encomendas restantes maximizam o seu valor (0-1 bidimensional Knapsack usando Programação Dinâmica).

Assim que o knapsack determina que o maior valor para um dado estafeta é inferior ao seu custo, a execução é terminada.

O algoritmo knapsack cria uma tabela 3d, com os índices das restantes encomendas num eixo, 0 até ao peso e 0 até ao volume da carrinha escolhida nos outros eixos. Popula cada célula com o maior valor dentro das restrições.



Cenário 2 - Descrição dos Algoritmos

```
for(int i = 0; i < vans.size(); i++){
    regardless.clear();
    chosen.clear();

    //0-1 knapsack bidimensional
    table = knapsack(deliveries, vans[i]);

    //get highest value
    int value = table[deliveries.size()][vans[i].getWeight()][vans[i].getVolume()];
    int w = vans[i].getWeight(), v = vans[i].getVolume(), n = deliveries.size();

    //save the chosen deliveries
    for(int k = n; k > 0; k--){
        if(table[k][w][v] != table[k-1][w][v]){
            chosen.push_back(deliveries[k-1]);
            w -= deliveries[k-1].getWeight();
            v -= deliveries[k-1].getVolume();
        }
    }

    //if profit is negative stop
    if(value - vans[i].getCost() <= 0) break;

    //remove deliveries from the poll
    for(auto d : chosen){
        std::remove(deliveries.begin(), deliveries.end(), d);
    }

    //save combination of van and deliveries
    ret.push_back(Combination(vans[i], chosen, value));
}
```

```
vector<vector<vector<int>>> knapsack(vector<Delivery> deliveries, Van van){
    int maxW = van.getMaxWeight(), maxV = van.getMaxVolume(), n = deliveries.size();
    int current, last;
    //start the cube all as 0s
    vector<vector<vector<int>>> table(n+1, vector<vector<int>>(maxW+1, vector<int>(maxV+1)));

    for(int i = 1; i <= n; i++){
        for(int w = 0; w <= maxW; w++){
            for(int v = 0; v <= maxV; v++){
                //if delivery i is heavier than the chosen weight, the cell will stay the same as the previous cell
                if(deliveries[i-1].getWeight() > w) table[i][w][v] = table[i-1][w][v];
                //if delivery i has higher volume than the chosen volume, the cell will stay the same as the previous cell
                else if(deliveries[i-1].getVolume() > v) table[i][w][v] = table[i-1][w][v];
                else{
                    last = table[i-1][w][v]; //previous value
                    current = table[i-1][w - deliveries[i-1].getWeight()][v - deliveries[i-1].getVolume()] + deliveries[i-1].getReward();
                    table[i][w][v] = std::max(last, current);
                }
            }
        }
    }

    return table;
}
```

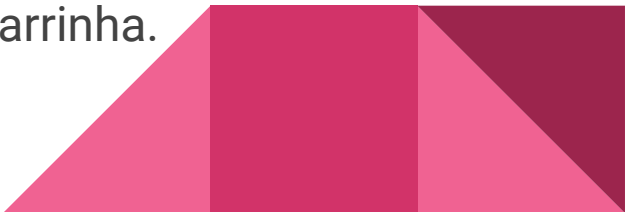
Cenário 2 - Análise de Complexidade

Complexidade Temporal: $O(n^4)$

Devido ao facto de termos de popular uma tabela com 3 eixos, para cada estafeta, a complexidade temporal é, no pior caso, $O(n^4)$.

Complexidade Espacial: $O(n^3)$

Com este algoritmo teremos sempre uma tabela cujas dimensões aumentam com o número de encomendas, peso e volume de cada carrinha.



Cenário 2 - Resultados

	Delivery:	Volume: 29	Weight: 16	Reward: 1051	Duration: 169
	Delivery:	Volume: 3	Weight: 16	Reward: 501	Duration: 692
	Delivery:	Volume: 22	Weight: 16	Reward: 982	Duration: 229
	Delivery:	Volume: 25	Weight: 28	Reward: 1370	Duration: 535
Van:	Volume: 328	Weight: 398	Cost: 13956		
	Delivery:	Volume: 23	Weight: 29	Reward: 1232	Duration: 389
	Delivery:	Volume: 16	Weight: 16	Reward: 706	Duration: 408
	Delivery:	Volume: 29	Weight: 23	Reward: 1148	Duration: 789
	Delivery:	Volume: 16	Weight: 29	Reward: 1002	Duration: 441
	Delivery:	Volume: 6	Weight: 22	Reward: 666	Duration: 504
	Delivery:	Volume: 9	Weight: 9	Reward: 427	Duration: 447
	Delivery:	Volume: 26	Weight: 26	Reward: 1261	Duration: 1009
	Delivery:	Volume: 13	Weight: 23	Reward: 916	Duration: 1026
	Delivery:	Volume: 20	Weight: 16	Reward: 823	Duration: 524
	Delivery:	Volume: 16	Weight: 29	Reward: 1083	Duration: 521
	Delivery:	Volume: 26	Weight: 22	Reward: 1065	Duration: 616
	Delivery:	Volume: 9	Weight: 25	Reward: 849	Duration: 1008
	Delivery:	Volume: 16	Weight: 13	Reward: 671	Duration: 726
	Delivery:	Volume: 16	Weight: 13	Reward: 654	Duration: 720
	Delivery:	Volume: 26	Weight: 29	Reward: 1233	Duration: 434
	Delivery:	Volume: 16	Weight: 20	Reward: 816	Duration: 1079
	Delivery:	Volume: 19	Weight: 19	Reward: 873	Duration: 118
	Delivery:	Volume: 23	Weight: 16	Reward: 868	Duration: 608
	Delivery:	Volume: 3	Weight: 19	Reward: 585	Duration: 827

number of vans 15
total value 426927
price 203033
profit 223894

Cenário 3 - Formalização

O cenário três é relativo aos pacotes com serviço expresso. As entregas são todas realizadas por um único carro, em viagens com pacotes unitários, isto é, apenas uma entrega por viagem. Nossa única restrição aqui é o tempo, cada pacote tem uma duração de viagem para ser entregue e temos um horário de funcionamento onde temos de “encaixar” as entregas.

A otimização aqui é focada no tempo médio de entregas por dia, tentando fazer o máximo de entregas com curtas durações.



Cenário 3 - Descrição do Algoritmo

Dado que o cenário três apresenta apenas uma restrição, lidar com ele torna-se simples, e para tal foi seguido o raciocínio de *first-fit-increasing*, comum em problemas de *bin-packing*, onde ordenamos as entregas através de suas durações em ordem crescente e encaixamos uma por uma, em ordem, até não conseguirmos mais.



Cenário 3 - Descrição do Algoritmo

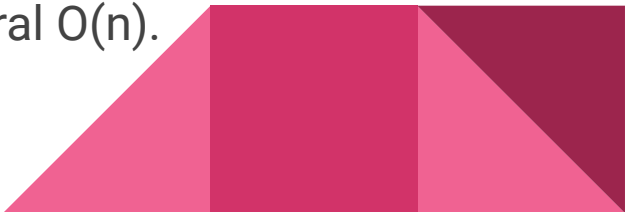
```
vector<Delivery> firstFitIncreasingCen3(vector<Delivery> &deliveries) {  
    unsigned time = 28800;  
    bool done = false;  
    vector<Delivery> result;  
    std::sort( first: deliveries.begin(), last: deliveries.end(), comp: compareTime);  
    auto delPtr :iterator<...> = deliveries.begin();  
    while (!done) {  
        unsigned thisDuration = delPtr->getDuration();  
        if (thisDuration < time && delPtr != deliveries.end()) {  
            time -= thisDuration;  
            result.push_back(*delPtr);  
            delPtr++;  
        } else {  
            done = true;  
        }  
    }  
    return result;  
}
```


Cenário 3 - Análise de complexidade

Complexidade Temporal: $O(n)$

A complexidade temporal é relacionada à quantidade de pacotes aguardando entrega n na forma de $O(n)$, onde o pior dos casos é verificarmos todos os pacotes e nenhum, ou apenas o último, puder ser entregue durante o horário de trabalho.

A estrutura de dados utilizada para guardar as informações dos pacotes nesse cenário foi apenas um vetor contendo n elementos, onde n é o número de pacotes a serem verificados, tendo complexidade temporal $O(n)$.



Cenário 3 - Resultados

Avaliando o terceiro cenário conclui-se que em um problema como esse onde temos uma restrição unidimensional e nosso objetivo é minimizar o tempo médio, uma solução gananciosa onde tentamos a cada iteração adicionar às entregas o pacote de menor duração dará-nos sempre o resultado óptimo e prova-se extremamente simples de se implementar (*first-fit-increasing*).

```
Made 124 deliveries today.
```



Algoritmo em Destaque

Cenário 2

- Utilização de métodos Greedy e Programação Dinâmica
- Maior Complexidade
- Rapidez quando comparado com suas implementações anteriores
- Utilização de knapsack para um problema com várias restrições



Principais dificuldades e contributos

- Pesquisar soluções para problemas de *bin-packing*.
- Encontrar o equilíbrio entre optimização da solução e complexidade de um algoritmo “perfeito” de *bin-packing*.
- Pesquisar soluções para problemas knapsack com várias restrições.
- Encontrar o equilíbrio entre optimização e complexidade de soluções de problema de *knapsack*.

Contributos:

Lucas Sousa 40%

João Teixeira 30%

Jose Pineda 30%

