Padeiro em Época de Covid

CAL Parte 1 - Turma 6 Grupo 9

Afonso Duarte de Carvalho Monteiro - up201907284@edu.fe.up.pt
João Ferreira Baltazar - up201905616@edu.fe.up.pt
Joel Alexandre Vieira Fernandes - up201904977@edu.fe.up.pt

2020/2021

Conteúdo

1 Descrição do Problema

Pretende-se otimizar um serviço de entregas de pão numa localidade de modo a minimizar o tempo total do itinerário e equilibrar o tempo de atraso nas entregas. É dada uma tolerância de atraso e de antecipação no tempo de entrega e uma duração do processo de entrega. Sobre cada cliente conhece-se a morada e hora preferencial de entrega. O problema base tem os seguintes pressupostos:

1. Disponibilidade de apenas um veículo de entrega;
2. Capacidade ilimitada de pães.

Deste modo, o objetivo da solução **1** passa, então, por obter o itinerário ideal com estas restrições. As soluções seguintes relaxam-nas gradualmente:

A solução **2** considera um número arbitrário de carrinhas de entrega.

A solução **3** considera um número arbitrário de carrinhas de entrega e condutores com duração de entrega específica.

A solução **4** considera um número arbitrário de carrinhas de entrega com capacidade limitada e condutores com duração de entrega específica.

2 Formalização do problema

2.1 Dados de entrada

- Frota de carrinhas F:
  - Identificador *id*;
  - Capacidade total de pães *cap*.

- Grafo *G(V, E)* - rede de distribuição:
  - Vértices V - Pontos de entrega:
    - Identificador *id*;
    - Arestas adjacentes $Adj \subseteq E$;
    - Hora preferencial de entrega (Cliente) *t_pref*;
    - Tolerância de atraso da entrega *t_atr*;
    - Tolerância de antecipação da entrega *t_ant*;
    - Quantidade de pães requesitados *qp*.
  - Arestas E - Deslocações ótimas entre os pontos de entrega:
    - Identificador *id*;
    - Tempo-distância *t_dist* que representa o tempo gasto no deslocamento entre os vértices ligados por esta.

- Vértice de origem e fim (padaria) *pad*;

- Conjunto de condutores C:
  - Identificador *id* (por simplificação, igual ao da carrinha a que corresponde);
  - Tempo de entrega médio *t_entr*.

2.2 Dados de Saída

- Um grafo *G(V, E)* semelhante ao de entrada;

- Um itinerário definido para cada condutor, com cada caminho e percurso associado (*itinerary*) a outros dados relevantes como capacidade, tempo de itinerário (*itinerary_time* e atraso total (*total_delay*).

2.3 Restrições

- Todos os identificadores $id \geq 0$ e não repetidos;
- A padaria é um vértice com $id = 0 \wedge t\_pref = 7h$;
- $|F| > 0$ - Tem de haver pelo menos 1 carrinha a fazer entregas;
- $\forall carr \in F,\ carr.cap \geq 0$ - Todas as carrinhas da frota têm capacidade não negativa;
- $\forall cli \in V,\ cli.t\_atr \geq 0$ - A tolerância de atraso de cada cliente tem de ser não-negativa;
- $\forall cli \in V,\ cli.qp \geq 0$ - A quantidade de pães a entregar tem de ser não negativa;
- $\forall cli \in V,\ cli.t\_ant \geq 0$ - A tolerância de antecipação de cada cliente tem de ser não-negativa;
- $|C| = |F|$ - Há tantas carrinhas como condutores;
- $pad \in V$ - A padaria é um dos pontos de interesse do percurso;
- $\forall cli \in V,\ cli.t\_pref \geq 7h$ - Os clientes só podem pedir entregas a partir das 7h;
- $\forall traj \in E,\ traj.t\_dist > 0$ - Todos os trajetos demoram um dado tempo;
- $\forall cond \in C,\ cond.t\_entr > 0$ - Cada condutor demora um dado tempo a entregar;
- $\forall cli \in V,\ cli.t\_pref - t\_ant \leq cli.t\_real \leq cli.t\_pref + t\_atr$ - A encomenda só pode ser realizada entre a hora mínima e máxima que o cliente se encontra disponível.

2.4 Funções objetivo

$$f = min(\sum_{cli \in V} \begin{cases} 0 & cli.t\_real - cli.t\_pref \leq 0 \\ cli.t\_real - cli.t\_pref & cli.t\_real - cli.t\_pref > 0 \end{cases}) \quad (1)$$

$$g = min(pad.entr\_real) \quad (2)$$

$$h = max(número\_de\_entregas) \quad (3)$$

A função f procura minimizar o atraso ao longo das entregas do dia.

A função g procura minimizar o tempo de chegada à padaria, ou seja, o tempo total do itinerário.

A função h pretende que se complete o máximo de entregas possível.

Considerar-se-á h como prioritária, e de seguida uma média pesada de f e g:

$$o = k * g + l * h,\ k + h = 1.0 \quad (4)$$

O algoritmo irá evidentemente explorar esta média pesada de forma a produzir resultados diferentes dependendo do que o utilizador do programa quer priorizar.

3 Descrição da solução

3.1 Solução 1

3.1.1 Considerações iniciais

À partida, encontramos já um problema com algumas propriedades nunca antes exploradas:

- Categoria de problemas do caixeiro viajante (TSP), com fatores de desempate;
- Janelas de tempo - com penalidades (de atraso) e impedimentos totais;
- Custo de percorrimento depende do vértice de chegada.

Para além disso, uma estratégia puramente gulosa (*greedy*) que minimize o tempo consumido no deslocamento e entrega não funciona.

Isto deve-se ao facto de neste problema, apesar de ser de maximização/minimização, uma optimização local não garantir necessariamente um solução ótima em todo o problema devido ao caráter dinâmico do custo de uma dada escolha. Logo, não tem uma subestrutura ótima.

No entanto, **a título ilustrativo**, apresentamos o pseudocódigo e um exemplo de insuficiência de tal algoritmo:

3.1.2 Pseudocódigo

```
1: greedy_base (G=(V, E)): // G - Grafo
2:   // Inicialização base
3:   T = 7h
4:   can_deliver = false
5:   no_more_paths = false

6:   for each v in V do
7:       // Dar reset aos vértices do grafo
8:       v.path= nil
9:       v.t_real = 0h

10:   // A padaria é o vértice inicial
11:   working_v = pad

12:   do:
13:       // Ordenar arestas do working_v por ordem
             decrescente
14:       edg_ord = sort_desc(working_v.Adj)
15:       // Remove as arestas que levam de volta a padaria
16:       remove_to(edg_ord, pad)

17:       while true do:
18:           // Escolher a primeira aresta, ou seja aquele
                 com o menor tempo
19:           shortest_edg = first(edg_ord)
20:           remove_first(edg_ord)
21:           // Esta condição determina se podemos entregar
                 o produto
22:           can_deliver = shortest_edg.dest.t_pref -
                            shortest_edg.dest.t_ant
                            <= shortest_edg.t_dist + T <=
                            shortest_edg.dest.t_pref +
                            shortest_edg.dest.t_atr

23:           if can_deliver:
24:               edg_ord.dest.t_real = shortest_edg.dist + T
25:               T = T + shortest_edg.dist + T_entr
26:               working_v = edg_ord.dest
```

```
27:                 break

28:             if EMPTY(edg_ord):
29:                 no_more_paths = true
30:                 break

31:   while ¬ no_more_paths

32:   // volta-se à padaria
33:   t_real(pad) = T + edg_to_pad_from(working_v).dist
```

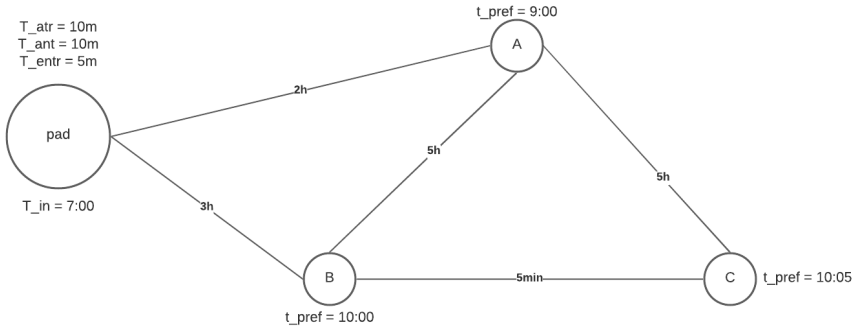

Figura 1: A estratégia gulosa *(greedy)* enunciada escolheria o caminho pad $\rightarrow A$, quando pad $\rightarrow B \rightarrow C$ é possível e melhor.

Este algoritmo no caso médio tem complexidade espacial $O(|E|)$ (espaço necessário para guardar o array/pilha de arestas de um vértice) e complexidade temporal $O(|E|^2 log(|E|))$, onde $|E| * log(|E|)$ corresponde ao algoritmo de ordenação de arestas *quicksort* e o outro fator $|E|$ à escolha de arestas a percorrer.

A partir dos *t_real* de cada vértice, podemos inferir o caminho e descrever o itinerário.

3.1.3 Considerações finais

Há, no entanto, uma constatação útil: conhecendo um percurso possível que atende N clientes, só será necessário considerar futuros caminhos de tamanho A $\geq$ N;
Assim, podemos usar um algoritmo de força bruta, com alguma heurística, de forma a tentar ao máximo encontrar o percurso que faz mais ou o mesmo número de entregas. No caso de um percurso ter o mesmo número de entregas será importante usar as equações de optimização **2** e **3** de forma a encontrar um solução ideal **(4)** e prosseguir para a próxima iteração.

Evidentemente, esta abordagem exaustiva terá complexidades de ordens exponenciais...

3.2 Solução 2

3.2.1 Considerações iniciais

Este problema representa uma mudança de paradigma: passando de 1 para N veículos de entrega, tem-se, daqui em diante, um tipo de vehicle routing problem (VRP), mais concretamente de VRP com janelas de tempo (VRPTW). Do ponto de vista do fornecedor, este modelo traz algumas vantagens relevantes, que alteram também a forma de abordar o problema. Ter mais viaturas a entregar permite realizar entregas que seriam de outra forma incompatíveis, solucionando situações como a da **Figura 1** em que tendo no mínimo 2 viaturas (uma para percorrer cada um dos caminhos mutuamente exclusivos) é possível assegurar que todos os clientes são atendidos.

Tendo isto em conta, uma possível implementação relativamente *naive* seria ordenar os clientes por hora de saída e alocá-los, um a um, a cada uma das viaturas de modo a equilibrar o mais possível a carga imposta.

3.2.2 Variação do algoritmo base

Nesta solução, antes de se ordenar as arestas, divide-se, se possível, as arestas provenientes da padaria igualmente por cada carrinha disponível. De seguida segue-se com a lógica definida anteriormente, iterando uma vez por cada condutor. Outra possível solução seria efetivamente ordenar os condutores por uma dada prioridade e sequencialmente correr o algoritmo para cada um deles.

3.2.3 Observações

Encontrámos uma publicação sobre VRPTW **(2)**, no entanto esta procura minimizar o número de viaturas utilizadas, enquanto o objetivo desta solução passa por fazer o melhor possível com um número fixo de viaturas definidas à entrada.

3.3 Solução 3

3.3.1 Considerações iniciais

Este VRP é mais complexo do que o descrito na solução anterior. Para além de termos N carrinhas, agora os condutores passam a ter um tempo de entrega variável. Desta forma, cada troço do grafo percorrido por cada condutor vai variar no tempo de percorrimento, dependendo do condutor que estiver associado. Isso irá implicar uma análise de eficiência de cada condutor por subdivisão do grafo, de forma a encontrar a associação condutor-troço mais eficiente.

3.3.2 Variação do algoritmo base

Aqui há que considerar uma seleção de condutores para cada sub-grafo. Propõe-se que se associe o condutor com tempo de entrega maior ao sub-grafo com as primeiras arestas mais pesadas, e vice-versa. Assim, utilizam-se ao máximo os condutores mais eficientes para mitigar o tempo total num circuito com mais clientes. Deste modo, se possível, uma análise de regiões densas de clientes ajudaria a identificar quais os locais a serem atendidos pelos condutores mais rápidos.

Para este fim, poder-se-ia recorrer à solução 2 para um desempenho otimista dos condutores (ou seja, assumindo que todos demoram o tempo de entrega do melhor) para obter as possíveis soluções. De seguida, alocar os condutores às viaturas de forma real baseado em quantos clientes atende cada um, como mencionado acima, para filtrar as opções.

3.3.3 Observações

Depois de se encontrar uma publicação **(3)**, apercebe-se que se passa a tratar de um VRP dinâmico com janelas de tempo (DVRPTW), cujo dinamismo se encontra nos condutores.

Contudo, a solução disponibilizada pelo artigo não se adequa muito à nossa questão, visto que essa resposta implica variação na quantidade de veículos, entrando em conflito com o nosso enunciado.

3.4 Solução 4

3.4.1 Considerações iniciais

Esta é a solução ao problema de quando temos capacidade limitada de produtos dentro de um veículo, para além de existir um número limitado de carrinhas e condutores com durações de entrega características. De uma certa perspetiva, esta solução poderá facilitar o processo de associação condutor(e carrinha)-troço, pois vai restringir condutores cuja carrinha não tenha capaciade suficiente para percorrer um dado troço, antes de se realizar a análise de eficiência de cada condutor, tal como foi descrito na secção anterior.

3.4.2 Variação do algoritmo base

Nesta solução há um incremento de complexidade na seleção dos condutores para cada sub-itinerário, pois temos de ter em consideração a possibilidade do veículo não ter capacidade suficiente para conseguir percorrer o troço. Idealmente dever-se-ia dar preferência a condutores troços cuja carga seja semelhante, com mais ou menos desvio, à capacidade do veículo em questão. Esta pré-seleção dar-se-ia antes da seleção em função do tempo/eficiência do condutor, que nesta situação será mais rápida, devido à existência desta pré-preferência.

4 Casos de utilização e funcionalidades a ser implementadas

O *software* a desenvolver não só poderá ser utilizado pelo padeiro de forma a planear o seu trajeto para aquele dia mas também por qualquer pessoa que pretenda resolver um problema VRPTW do nosso tipo.

Iremos implementar as diferentes soluções que apresentamos anteriormente permitindo ao utilizador escolher todos os dados de entrada. Irá ser possível visualizar todo o grafo criado e respetiva solução do problema através da interface gráfica de visualização de grafos fornecida.

5 Implementação

5.1 Considerações iniciais

À data de entrega, foram efetivamente implementadas as quatro soluções que foram consideradas nos pontos acima descritos. Foram usadas *Priority Queues*, *Mutable Min-Priority Queues* e grafos como estruturas de dados para representar a situação e prestar auxílio na formulação das soluções.

Tendo em conta o que fora demonstrado em pseudocódigo na secção **3.1.2** o algoritmo que foi de facto implementado difere nos seguintes pontos:

- Existem vértices onde não há entrega - Vértices intermédios. Deste modo, ao invés de ser usando um algoritmo de ordenação de arestas para encontrar a conexão mais próxima, foi usado o algoritmo de caminho mais curto de *Dijkstra* de forma a encontrar todos os caminho e de seguinda um algoritmo que calcula qual desses caminhos é o mais curto - *Nearest Neighbour*;
- A nossa solução implementa a versão mais complexa do algoritmo (**3.4**). Portanto, existem parâmetros e verificações adicionais, por exemplo, capacidade da carrinha e quantidade de entrega de pães;
- Foram realizadas optimizações comumente denominadas de pre-processamento que aceleram o cálculo do vizinho mais próximo de subsequentes iterações.

5.2 Pseudocódigo

Segue-se o pseudocódigo que representa a solução deste problema:

```
1:minPath(G=(V, E), PQC : priority_queue(C)):/C - Carrinhas e
                                          condutores, G - Grafo
2:  while C != ∅:
3:      c <- DEQUEUE(PQC) // funcao da priority_queue
4:      calcMinPath(G, c) -> use data // usar os dados devolvidos
                                        pela função e apresentar
                                        ao cliente (utilizador)

5:  resetGraph(G)
```

Neste pedaço de código a *priority_queue* de condutores/carrinhas está ordenada por maior capacidade e, no caso se serem iguais, por menor tempo de entrega. (*greedy approach*)

```
1:calcMinPath(G=(V, E), c): // c - Carrinha e
                               condutor, G - Grafo
2:  working_node_id = pad // a padaria e o vértice inicial
3:  current_time = 7h // tempo inicial é igual ao da padaria
4:  initial_time = current_time
5:  total_delay = 0 // delay inicial
6:  itinerário = ∅ // itinerário, inicialmente vazio, vetor
7:  vertCopy = ∅ // Cópia dos vértices do grafo para
                    pre-processamento, inicialmente vazio, vetor

8:  For each v in V: // remove todos os vértices onde a
                        entrega já foi realizada e nós
                        intermédios - pre-processamento
9:      if (v.t_pref != 0h && v not in path && v.id != 0):
10:          vertCopy.insert(v)

11: Do:
12:     dijkstra(G ,working_node) // algoritmo de dijkstra sobre
                                     o vértice atual
13:     short_path = {NIL, INF, 0} // vértice, distância,
                                      índice, pães a entregar

14:     For each cV in vertCopy: // calcula o vértice mais
                                    proximo e possível
15:         in_time = current_time + cV.dist <= cV.t_pref +
                                                cV.t_atr
16:         if (cV.dist < short_path[1] && cV not in path &&
                cV.id != working_node_id &&
                in_time &&
                dm.cap >= cV.pq):
17:                 short_path = {cV, cV.dist, cV.pq}
18:         else if (cV in path || cV.id = working_node.id)
19:                 vertCopy.erase(cV) // optimização

20:     if (short_path[0] == NIL)
21:         break; // se não houver qualquer caminho possivel
```

```
                    o algoritmo tem de parar

22:     dm.cap -= short_path[2]
23:     current_time = max(current_time + short_path[1],
                  short_path[0].t_pref - short_path[0].t_atr)
24:     total_delay += current_time - short_path[0].t_pref
                    // em minutos
25:     current_time += dm.t_entr
26:     working_node = short_path[0]
        short_path[0] -> in path
        itinerário.push(short_path[0])

27: While (true)

28: dijkstra(G, working_node) // caminho para a padaria
29: current_time += pad.dist // adicionar o tempo de viagem

30: return {itinerary_time = current_time - initial_time,
            delay = total_delay, itinerary = itinerário}
```

O algoritmo de dijkstra usado está apresentado em baixo em pseudocódigo. Usa-se uma estrutura auxiliar, neste caso uma *priority_queue*, para realizar o algoritmo de Dijkstra:

```
1:dijkstra(G=(V, E), s): // G grafo,
                         s é vértice em V de G
2:  For each v in V:
3:      v.dist = INF
4:      v.path = nil
5:  s.dist = 0
6:  Q = ∅ // min-priority queue
7:  INSERT(Q, (s, 0)) // insere s com chave 0
8:  While Q != ∅:
9:      v = EXTRACT-MIN(Q) // guloso (greedy)
10:     For each w in v.Adj:
11:         if w.dist > v.dist + weight(v,w):
12:             w.dist = v.dist+ weight(v,w)
                // weight - valor da aresta entre v e w
13:             w.path = v
14:         if w in Q: // distância antiga era INF
```

```
15:             INSERT(Q, (w, dist(w))) // função da MPQ
16:         else
17:             DECREASE-KEY(Q, (w, dist(w))) //função da MPQ
```

Nota: O uso de *min-fibonacci heaps* também seria possível de forma a optimizar o algoritmo.

A função que dá um reset geral do grafo apresenta-se em baixo:

```
1: resetGraph(G=(V, E))
2:  For each v in V:
3:      v.path = NIL
4:      v.dist = INF
5:      v.queueIndex = 0
6:      v no longer in path
```

5.3 Análise da solução

5.3.1 Notas Iniciais

O algoritmo utilizado incluir alguns dados auxiliares que não se encontram presentes na secção **2.1**:

- **Vertex.path** - Utilizado pelo algoritmo de Dijkstra para indicar o vértice que antecede o *Vertex*.
- **Vertex.dist** - Utilizado pelo algoritmo de Dijkstra para indicar a distância mínima entre o vertice em estudo e o *Vertex*.
- **Vertex.queueIndex** - Utilizado na *Mutable Min-Priority Queues* para definir o index do elemento na fila.
- **Vertex in Path** - Utilizado pelo algoritmo principal para determinar se o *Vertex* ja foi visitado e a entrega já fora realizada por outra carrinha/condutor.

5.3.2 Complexidade

5.3.2.1 Análise Teórica

Em termos espaciais, a solução implementada requer espaço adicional da ordem de $O(|V|)$, visto que há um pré-processamento dos vértices de modo a facilitar o cálculo do vizinho mais próximo. A complexidade temporal, como é possível observar acima, é $O((|V| + |E|) * log(|V|) * |V| * |C|)$,

sendo que $(|V| + |E|) * log(|V|)$ advém do algoritmo de Dijkstra, e $|V| * |C|$ de ser calculado para cada combinação vértice - condutor.

5.3.2.2 Análise Empírica

Na prática, o algoritmo demora bastante menos tempo a executar que o limite teórico, visto que com a grande quantidade de restrições (temporais, de capacidade e de quantidade e qualidade de condutores), as possibilidades de percorrimento diminuem muito rapidamente, cortando significativamente o tempo de execução no caso médio.

A visualização da escalabilidade do algoritmo é severamente limitada pela quantidade exacerbada de variáveis em jogo, tendo sido feita uma análise mais focada nos fatores *dataset*, D, (com tamanhos diferentes e sendo sempre extensões do *set* anterior) e condutores, C. Deste modo, uma tabela bidimensional é a escolha mais acertada para representar os dados. Cada entrada (i,j) representa o tempo médio, em ms, do cálculo da melhor rota para D_i e C_j, ou seja, o *dataset* i e número de condutores j.

	D_1	D_2	D_3
C_1	4	5	6
C_2	10	12	15
C_3	11	17	22

Tabela 1: Tempos de execução com diferentes *datasets* e grupos de condutores

D_1: 5 Vértices, 13 Arestas
D_2: 10 Vértices, 24 Arestas
D_3: 14 Vértices, 34 Arestas

C_1: Capacidade 100, Tempo de entrega 10 minutos
C_2: Capacidade 200, Tempo de entrega 5 minutos + C1
C_3: Capacidade 100, Tempo de entrega 12 minutos + C2

É de notar o crescimento aproximadamente linear com o aumento dos condutores, ainda que com retornos diminuídos - este facto é especialmente aparente em D_1C_3, que tem um aumento negligente pois o terceiro condutor não adiciona nada a uma solução já satisfatória e completa.

5.4 Conetividade

Depois da implementação, conseguimos concluir que o grafo usado é efetivamente **Misto**, **Pesado** e em termos de conectividade **Fortemente Conexo**:

- **Misto** - As arestas podem ser uni ou bidirecionais, correspondendo a estradas/caminhos de sentido único ou não;
- **Pesado** - As arestas têm um peso associado que neste caso corresponde ao tempo que aquele trajeto demora em minutos;
- **Fortemente Conexo** - Existe a partir de cada vértice sempre um caminho para outro vértice. Não faria sentido existirem lugares de entrega sem qualquer tipo de acesso a partir de um outro vértice. Podem efetivamente existir certos locais para os quais o acesso é muito complicado sendo necessário percorrer uma distância muito maior do que a *ideal*. No entanto, nunca serão inacessíveis, tal como acontece na vida real, e este é um pressuposto importante para a execução.

6 Conclusão

6.1 Preliminar

Foi de dificuldade notável a transposição do problema em mão para formulações e soluções lecionadas/conhecidas ou semelhantes - principalmente:

- O caráter dinâmico do problema (ter uma dimensão tempo que é restritiva, pela penalidade de atraso, e proibitiva, pela intolerância para lá de um tempo limite);
- A tipologia de problema principal (TSPTW, mTSPTW, mTSPTW com tempo de entrega heterogéneo, CVRPTW com capacidade e tempo de entrega heterogéneos e número de veículos limitado), que já seria problemática não fossem todos os outros acréscimos.

Para além disso, mesmo uma pesquisa profunda e intensa em publicações académicas relacionadas resultou numa aquisição de conhecimento muito fraco e, de um modo geral, impraticável ao problema em questão, o qual é exacerbadamente específico.

Deste modo, o relatório é globalmente inconclusivo e o processo de produção da solução final poderá exigir algumas simplificações do problema de modo a assegurar uma solução computável em tempo útil.

6.2 Pós implementação

Em análise retrospetiva, a primeira solução foi a mais difícil de implementar, devido ao facto de ser necessário fazer pela primeira vez a transformação do escrito para código, com toda a preparação que acarreta.

A partir daí, as restantes foram mais simples, porque as mesmas são derivações da primeira solução, logo basta acrescentar novas restrições ao algoritmo presente na função *minpath*. A sua generalização passou pela implementação de todas as restrições possíveis que depois poderiam ser progressivamente relaxadas (com capacidade da carrinha infinita ou usando apenas uma carrinha, por exemplo) de modo a gerar as soluções intermédias propostas. De facto, estas manipulações podem criar outros tipos de soluções não descritas, combinando restrições de uma forma nova.

De um modo geral, é uma solução bastante satisfatória, cumprindo os objetivos autopropostos de modo apreciável. Foi possível transpor o algoritmo na sua totalidade para código, estando funcional como planeado. Apresenta uma interface simples mas eficaz, com a ilustração dos percursos de cada condutor. Não foi realizada a integração com os *datasets* de OSM por causa da complexidade dos dados de entrada necessários. Introduzir as *time windows* sem planeamento prévio dá, quase sempre, um resultado nada interessante/relevante porque se torna impossível ou muito limitado. Também não foram criados *datasets* aleatórios pelo mesmo motivo. Apesar disso, são fornecidos alguns *sets* meramente ilustrativos.

7 Esforço de cada elemento do grupo

Afonso Duarte de Carvalho Monteiro → Formalização do problema; Desenvolvimento teórico das soluções 2, 3 e 4; Análise de soluções; Conclusão ($\frac{1}{3}$).

João Ferreira Baltazar → Correção linguística e técnica; Formalização do problema; Análise teórica de soluções; Conclusão ($\frac{1}{3}$).

Joel Alexandre Vieira Fernandes → Pseudocódigo; Formalização do problema; Formatação do documento e imagens; Casos de utilização de funcionalidades a ser implementadas; Implementação em C++ ($\frac{1}{3}$).

Nota: Houve um esforço conjunto e colaborativo em todo o trabalho, especialmente nos aspetos não mencionados acima.

Bibliografia

[1] Moradi, B. The new optimization algorithm for the vehicle routing problem with time windows using multi-objective discrete learnable evolution model. Soft Computing 24, 6741–6769 (2020). https://doi.org/10.1007/s00500-019-04312-9

[2] Nasser A. El-Sherbeny, Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods, Journal of King Saud University - Science. https://doi.org/10.1016/j.jksus.2010.03.002

[3] Yang, Z., van Osta, JP., van Veen, B. et al. Dynamic vehicle routing with time windows in theory and practice. Nat Comput 16, 119–134 (2017). https://doi.org/10.1007/s11047-016-9550-9