

Mestrado Integrado em Engenharia Informática e Computação
Ano Lectivo 2020/2021

Concepção e Análise de Algoritmos: O Padeiro da Vila em Época de Covid (Tema 9)

Análise do problema e discussão de possíveis soluções

Turma 7 - Grupo 4

Ângela Filipa Ribeiro Coelho	up201907549
Marta Cristina dos Santos Mariz	up201907020
Patrícia do Carmo Nunes Oliveira	up201905427

Índice

1. Descrição do Problema	3
1.1. 1 ^a iteração: Entregas sem hora preferencial	3
1.2. 2 ^a iteração: Entregas com hora preferencial	3
1.3. 3 ^a iteração: Entregas com horas preferencial e tolerância de antecipação	4
2. Formalização do Problema	5
2.1. Dados de entrada	5
2.2. Dados de saída	5
2.3. Restrições	6
Dados de Entrada	6
Dados de Saída	6
2.4. Funções objetivo	7
3. Perspetiva de Solução	8
3.1. Pré-processamento dos dados de entrada	8
Pré-processamento do Grafo	8
Pré-processamento das encomendas	8
Pré-cálculo dos percursos de duração mínima	8
3.2. Problemas encontrados	9
3.3. Conectividade do Grafo	9
Algoritmo de Pesquisa em Profundidade (depth-first search)	10
Algoritmo de Pesquisa em Largura (breadth-first search)	10
Baseado em Conjuntos Disjuntos	11
Algoritmo Paralelo de Shiloach-Vishkin	12
3.4. Caminho mais curto entre dois pontos	13
Algoritmo de Dijkstra	13
Algoritmo de Dijkstra bidirecional	15
Algoritmo A*	15
3.5. Caminho mais curto entre todos os pares de vértices	16
Algoritmo de Dijkstra	16
Algoritmo de Floyd-Warshall	17
3.6. Caminho mais curto, partindo da padaria, passando por todos os clientes	17
Algoritmos Exatos	17
Algoritmos Aproximados	19
4. Casos de Uso	20
5. Melhorias na Aplicação	21
5.1. Existência de uma frota de Carrinhas	21
5.2. Condutores afetam o tempo de entrega	22
5.3. Carrinhas com capacidade limitada	23
5.4. Utilização de mapas reais	23
6. Objetivos do Trabalho	24

7. Principais funcionalidades e cenários efetivamente implementados	26
8. Estruturas de dados utilizadas	27
9. Algoritmos Implementados (pseudocódigo)	28
9.1. Análise da conectividade	28
9.2. Pré-processamento dos dados	28
9.3. Entrega de todas as encomendas	29
9.4. Entrega de encomendas num intervalo de tempo	29
10. Análise da complexidade dos algoritmos	31
10.1. Análise Teórica (temporal e espacial)	31
10.2. Análise Temporal Empírica	31
11. Análise da Conectividade do Grafo	32
12. Conclusões	33
12.1. Aspetos positivos	33
12.2. Aspetos negativos	33
13. Contribuições	34
14. Bibliografia	35

1. Descrição do Problema

Na Vila da Tocha há um padeiro, o Sr. Sílvio, que vende o pão diretamente aos clientes, distribuindo-o com a sua carrinha. Assim, o Sr. Sílvio parte às 7 horas da manhã com o objetivo de realizar as entregas no menor tempo possível, só regressando de novo à padaria quando já todas as encomendas tiverem sido entregues.

Para simplificação do problema nestas três iterações iniciais, é considerado que apenas existe uma carrinha cuja capacidade é tomada como sendo ilimitada e também que cada entrega em si tem uma duração de 5 minutos, não dependendo em nada do condutor.

1.1. 1^a iteração: Entregas sem hora preferencial

Numa primeira fase considera-se que as entregas não têm uma hora preferencial para serem entregues, sendo apenas necessário que estas sejam entregues a todos os clientes (uma aproximação simplista do *Travelling Salesman Problem*). Para que tal seja possível é necessário que cada cliente tenha uma morada de entrega associada.

Portanto, o Sr. Sílvio irá partir da sua padaria às 7 horas e apenas se terá de preocupar em fazer todas as entregas relativas a esse dia no trajeto de menor duração, só regressando à sua padaria após ter concluído as entregas.

É importante notar que o cálculo da duração do trajeto deverá ter em conta que cada entrega em si tem uma duração constante de **5 minutos**, não dependendo do condutor. Para além disso, é fundamental ter em conta que podem haver determinados fatores, como obras nas vias públicas, que impeçam o Sr. Sílvio de chegar a determinada morada, exigindo uma análise cuidadosa.

1.2. 2^a iteração: Entregas com hora preferencial

Nesta fase já é tido em conta a hora preferencial de entrega, sendo que para além de cada cliente ter uma morada de entrega associada também passará a ter uma hora preferencial para que esta seja feita. Assim, o Sr. Sílvio terá não só de passar por todos os clientes no menor tempo possível, como também as horas preferenciais relativas aos clientes.

Uma vez que nem sempre será possível encontrar um trajeto que satisfaça exatamente todas as horas preferenciais dos clientes, ter-se-á em conta um **período de**

tolerância de 30 minutos. Posto isto, no caso de um cliente preferir que a entrega seja feita às 8:00, o Sr. Sílvio terá entre as 8:00 e as 8:30 para a realizar.

Aqui passa então a haver o novo objetivo de equilibrar o tempo de atraso nas entregas para além do de minimizar o tempo total do itinerário.

1.3. 3^a iteração: Entregas com horas preferencial e tolerância de antecipação

Por fim, na última fase considera-se que uma entrega só pode ser feita quando apenas faltar um determinado período de tempo constante (**10 minutos**) e igual para todos os clientes, até à hora preferencial. Desta forma, se faltar mais tempo do que o mínimo definido, o Sr. Sílvio terá de esperar até que o tempo se encontre mais próximo da entrega, perfazendo esse período mínimo.

Assim sendo, de forma a clarificar melhor o funcionamento das entregas, tome-se como exemplo um cliente que tem como preferência que a entrega seja feita às 9:00. Neste caso, o Sr. Sílvio não poderá fazer uma entrega antes das 8:50 e terá de ser feita no máximo até às 9:30. Caso a encomenda seja entregue, por exemplo, à hora esperada, isto é, às 9:00, então o Sr. Sílvio só passará para a próxima encomenda ou voltará à padaria às 9:05.

2. Formalização do Problema

2.1. Dados de entrada

- O_i - Sequência de encomendas previstas. Cada encomenda é caracterizada por:
 - address - morada do cliente;
 - preferredTime - hora preferencial de entrega;
 - quantity - quantidade de pães encomendados.
- $G_i = (V_i, E_i)$ - Grafo dirigido pesado, composto por:
 - V - vértices que representam pontos da cidade, como endereços de um mapa, com:
 - id - identificador do vértice;
 - latitude - coordenada real obtida através do mapa;
 - longitude - coordenada real obtida através do mapa;
 - $Adj \subseteq E$ - conjunto de arestas que partem do vértice;
 - reachable - indicador de acessibilidade.
 - E - Arestas que representam as vias de comunicação, com:
 - id - identificador da aresta;
 - w - peso da aresta que neste contexto corresponde ao tempo que demora a percorrer essa mesma aresta;
 - dest $\in V$ - vértice de destino da aresta.
- B_i - Vértice que representa a padaria do Sr Sílvio.

2.2. Dados de saída

- $G_f = (V_f, E_f)$ - Grafo dirigido pesado com V_f e E_f com os mesmos valores para todos os atributos que V_i e E_i , respetivamente.
- O_f - Sequência ordenada de encomendas que representa o caminho mais curto, tendo em conta as horas preferenciais, sendo cada uma caracterizada por:
 - address - morada pretendida para a encomenda;
 - preferredTime - hora preferencial de entrega;
 - deliveryTime - hora a que de facto foi entregue a encomenda.

2.3. Restrições

Dados de Entrada

- $\forall e \in E, w(e) > 0$, dado que o peso de uma aresta representa o tempo que uma carrinha demora a percorrer essa mesma aresta e, uma vez que não há tempos negativos, o peso neste caso terá de ser sempre superior a zero.
- $\forall o \in O_i, preferredTime(o) > 0$, mais uma vez porque a hora preferencial de entrega se trata de um tempo.
- $\forall o \in O_i, quantity(o) > 0$, já que uma quantidade não poderá apresentar nem valores negativos nem nulos.
- $\forall o \in O_i, dest(o)$ deve pertencer à mesma componente fortemente conexa do grafo inicial (G_i) que o vértice relativo à padaria (B_i), visto ser necessário calcular o caminho de retorno à padaria.
- $B_i \in V_i$, a padaria trata-se de um vértice do grafo.

Dados de Saída

- $|D_i| = |D_f|$, pois o número de encomendas a entregar inicialmente terá de ser igual ao número de encomendas entregues.
- $\forall v_f \in V_f, \exists v_i \in V_i$ tal que v_i e v_f têm os mesmos valores para todos os atributos.
- $\forall e_f \in E_f, \exists e_i \in E_i$ tal que e_i e e_f têm os mesmos valores para todos os atributos.
- $\forall o \in O_f, deliveryTime(o) > 0$, uma vez que se trata de uma medida de tempo.
- $\forall o \in O_f, preferredTime(o) > 0$, uma vez que se trata de uma medida de tempo.
- $\forall o_f \in O_f, \forall o_i \in O_i, address(o_f) = address(o_i) \Rightarrow$
$$preferredTime(o_f) = preferredTime(o_i)$$

Isto porque a hora preferencial das entregas inicialmente terá de se manter ao longo do programa, porque se trata de um valor fixo, quer se entregue a essa hora ou não.

- $\forall o \in O_f, deliveryTime(o) \leq preferredTime(o) + 30$, o Sr. Sílvio pode chegar até 30 minutos depois da hora preferencial indicada.
- $\forall o \in O_f, deliveryTime(o) \geq preferredTime(o) - 10$, o Sr. Sílvio pode chegar até 10 minutos antes da hora preferencial indicada.

2.4. Funções objetivo

O algoritmo a ser implementado deve minimizar o tempo total do itinerário, e equilibrar o atraso em cada entrega:

- $f = \sum_{e \in E_f} w(e)$
- $g = \sum_{o \in O_f} |preferredTime(o) - deliveryTime(o)|$

3. Perspetiva de Solução

3.1. Pré-processamento dos dados de entrada

Pré-processamento do Grafo

De forma a aumentar a eficiência temporal, antes das iterações, o grafo deve ser pré-processado de forma a reduzir o número de vértices e arestas. Primeiro são ignoradas as arestas que representam locais inacessíveis (devido a obras nas vias públicas, por exemplo). Numa segunda etapa ignoram-se os vértices que não permitem ao Sr. Sílvio regressar à sua padaria após realizar todas as entregas. Melhor dizendo, identificam-se com um booleano (através do atributo *reachable*) todos os vértices que não pertencem ao mesmo componente fortemente conexo do grafo onde se encontra a padaria.

Desta forma garante-se que quando se passar à aplicação dos algoritmos, estes já não consideram vértices e/ou arestas inacessíveis à carrinha.

Pré-processamento das encomendas

Após o pré-processamento do grafo com apenas vértices e arestas relevantes à distribuição, não faz sentido manter na lista de encomendas aquelas que não pertencem ao grafo. Assim, percorre-se a lista de encomendas e aquelas cuja morada de entrega do cliente não se encontrar no grafo pré-processado, são eliminadas da lista e as restantes associadas aos vértices.

Por fim, de forma a realizar as entregas pela ordem temporal, ordena-se a lista de encomendas de acordo com a hora preferencial dos clientes. Com isto pretende-se não só respeitar a prioridade como também minimizar o tempo de espera dos clientes.

Pré-cálculo dos percursos de duração mínima

Ao realizar este pré-cálculo da duração mínima entre os vários pontos são obtidos resultados mais eficientes, já que isto não terá de ser feito a cada novo percurso. Contudo, como terá de ser feito o cálculo da distância mínima entre todos os pares de vértices, quer sejam relevantes para um dado percurso ou não, surge aqui uma exigência computacional. Portanto, terá de ser feita uma análise cuidadosa da eficiência destes algoritmos tendo sempre em conta o tamanho do mapa em questão.

3.2. Problemas encontrados

Na primeira iteração há apenas uma única carrinha de capacidade ilimitada não havendo hora preferencial para as entregas, pelo que o importante é apenas encontrar o **caminho mais curto** independentemente da ordem das encomendas.

Já numa segunda iteração é adicionado ao problema a hora preferencial de cada cliente, fazendo com que o Sr. Sílvio tenha de passar em determinados pontos até determinada hora, não ultrapassando a tolerância de atraso. Ou seja, passa haver uma ordem específica pela qual o trajeto deve ser traçado passando a ser importante **equilibrar o tempo de atraso nas entregas** para além de minimizar o tempo total do itinerário.

Por fim, na última iteração é adicionado o tempo de tolerância de antecedência vindo reforçar mais uma vez a importância de haver um equilíbrio de tempo nas entregas, respeitando sempre a prioridade destas.

Portanto, o problema proposto assemelha-se ao **Problema do Caixeiro Viajante** (ou, em inglês, *Travelling Salesman Problem*). No entanto, neste contexto, os pontos inicial e final são determinados no pré-processamento dos dados, correspondendo ambos à padaria do Sr. Sílvio (parte às 7:00h da padaria e retorna a esta após todas as entregas terem sido realizadas).

3.3. Conectividade do Grafo

Só é possível realizar todas as entregas a todos os clientes se a partir do vértice inicial, a padaria do Sr. Sílvio, for possível chegar a todos os outros vértices (uma única árvore de expansão - dois vértices estão conectados se pertencerem à mesma árvore de expansão). É necessário, assim, avaliar a conectividade do grafo dirigido.

Tal também garante que existe pelo menos 1 caminho que ligue qualquer par de vértices do grafo, sendo que no pior caso este corresponde ao percurso que liga todos os vértices entre si. Nesta fase é necessário ter especial atenção a zonas ou clientes inacessíveis devido a arestas indisponíveis, que correspondem a ruas desimpedidas, por isso inutilizáveis para a navegação

Esta análise pode ser realizada recorrendo a algoritmos que verificam a existência de componentes conexos, como por exemplo algoritmo de **Pesquisa em Profundidade**, **Pesquisa em Largura**, **Baseado em Conjuntos Disjuntos** ou **Algoritmo Paralelo de Shiloach-Vishkin**.

Algoritmo de Pesquisa em Profundidade (*depth-first search*)

Neste algoritmo, as arestas são exploradas a partir do próximo vértice adjacente ao atual, desde que este ainda não tenha sido visitado, explorando até chegar a um vértice folha (vértice sem adjacentes ou apenas com adjacentes já visitados), momento no qual ocorrerá o *backtracking*. A sua implementação em pseudocódigo é a seguinte:

DFS(G): 1. for each $v \in \text{VertexSet}$ do: 2. $\text{visited}(v) \leftarrow \text{false}$ 3. for each $v \in \text{VertexSet}$ do: 4. if $\text{not visited}(v)$ then: 5. VisitDFS(G, v)	VisitDFS(G, v): 1. $\text{visited}(v) \leftarrow \text{true}$ 2. for each $w \in \text{Adj}(v)$ do: 3. if $\text{edge}(v, w)$ reacheable and $\text{not visited}(w)$ then: 4. VisitDFS(G, w)
---	--

Complexidade temporal de $O(|V| + |E|)$ e complexidade espacial de $O(|V|)$.

Algoritmo de Pesquisa em Largura (*breadth-first search*)

Inicia a pesquisa no vértice da padaria do Sr. Sílvio, explora todos os vértices a que se pode chegar a partir deste, só depois passando para o seguinte. A ordem dos próximos vértices a serem analisados é controlada pelo uso de uma fila de prioridade. A sua implementação em pseudocódigo encontra-se a seguir, em baixo.

BFS(G, s): 1. for each $v \in \text{VertexSet}$ do: 2. $\text{visited}(v) \leftarrow \text{false}$ 3. $Q \leftarrow \emptyset$ 4. push (Q , s) 5. $\text{visited}(s) \leftarrow \text{true}$ 6. while $Q \neq \emptyset$ do: 7. $v \leftarrow \text{pop}(Q)$ 8. for each $w \in \text{Adj}(v)$ do: 9. if $\text{edge}(v, w)$ reachable and $\text{not visited}(w)$ then: 10. push (Q , w) 11. $\text{visited}(w) \leftarrow \text{true}$

Complexidade temporal de $O(|V| + |E|)$ e complexidade espacial de $O(|V|)$.

Baseado em Conjuntos Disjuntos

Neste algoritmo, tira-se partido de conjuntos disjuntos para identificar os vértices conexos entre si, sendo que tal é verdade se pertencerem ao mesmo conjunto, ou seja caso tenham em comum a mesma “raiz” desse conjunto.

Os conjuntos disjuntos são conjuntos cuja interseção resulta num conjunto vazio (cada elemento encontra-se apenas num único conjunto) e todos os conjuntos são distinguidos por um membro que os representa de forma unívoca.

Para a implementação deste algoritmo são necessárias as seguintes operações por parte da estrutura de dados que representa os conjuntos disjuntos:

- **Make Set**: cria um novo conjunto com um elemento ‘x’, garantindo que este já não se encontre em nenhum outro dos conjuntos existentes;
- **Union**: une dois conjuntos contendo ‘x’ e ‘y’, alterando os conjuntos originais;
- **Find Set**: Retorna o representante do conjunto (“raiz”) ao qual o elemento pertence.

Generalizando, sem definir ainda a estrutura de dados que representa os conjuntos disjuntos, o pseudocódigo deste algoritmo seria o seguinte:

```
DisjointSet(G, s):
1.   for each v ∈ VertexSet do:
2.       makeSet(v)
3.   for each edge(u, v) ∈ G.adj() do:
4.       if findSet(u) != findSet(v) then:
5.           union(u, v)

sameComponent(u, v):
1.   if findSet(u) == findSet(v) then:
2.       return true
3.   else
4.       return false
```

Um *array* poderia ser usado como um dos métodos possíveis de representar um conjunto disjunto. Neste caso, as operações *makeSet* e *union* teriam uma complexidade temporal de $O(|V|)$ e a operação de *findSet* de $O(1)$. A complexidade espacial seria de $O(|V|)$. Assim, a complexidade temporal global do algoritmo corresponderia a $O(|E| * |V|)$.

Algoritmo Paralelo de Shiloach-Vishkin

Trata-se de um algoritmo paralelo que usa $O(|E| + |V|)$ processadores, para encontrar componentes conexos de um grafo não dirigido. A complexidade temporal deste é $O(\log(|V|))$, uma vez que tira partido do processamento em paralelismo para realizar esta análise. Por outro lado, ao aceder simultaneamente ao mesmo local de memória permite a leitura e gravação de “instruções”.

Torna-se assim num algoritmo muito potente e rápido no processamento de grandes mapas, comparativamente com os algoritmos anteriormente apresentados. No entanto, necessita também de um elevado poder computacional e depende muito do número de cores disponíveis no sistema.

Um possível esqueleto para exemplificar este algoritmo é o seguinte:

```
procedure Shiloach-Vishkin( $V, E$ ):
1: for all  $v \in V$  :  $\pi(v) \leftarrow v$ 
2:  $hooking \leftarrow \text{true}$ 
3: while  $hooking$  do
4:    $hooking \leftarrow \text{false}$ 
5:   for all  $u \in V$  do in parallel
6:     for all  $v \in \mathcal{N}(u)$  do in parallel
7:       if  $\pi(u) < \pi(v)$  and
8:          $\pi(v) = \pi(\pi(v))$  then            $\triangleright hook phase$ 
9:            $\pi(\pi(v)) \leftarrow \pi(u)$ 
10:           $hooking \leftarrow \text{true}$ 
11:        end if
12:      end for
13:    end for
14:    for all  $v \in V$  do in parallel
15:      while  $\pi(\pi(v)) \neq \pi(v)$  do            $\triangleright shortcut phase$ 
16:         $\pi(v) \leftarrow \pi(\pi(v))$ 
17:      end while
18:    end for
19:  end while
20: return  $\pi$ 
```

Concluindo, é ótimo algoritmo em termos exploratórios e temporais, no entanto, no âmbito deste projeto estamos a tratar grafos dirigidos e este algoritmo aplica-se a grafos não dirigidos.

3.4. Caminho mais curto entre dois pontos

Após o pré-processamento do grafo, apenas lhe pertencem pontos acessíveis nos quais se incluem as moradas de entrega. O objetivo reduz-se então a encontrar o caminho mais curto (com menor duração) entre apenas dois pontos.

Existem vários algoritmos para resolver este problema, entre os quais se encontra o algoritmo de **Dijkstra** que permite não só encontrar o caminho mais curto entre dois pontos, mas também encontrar o caminho mais curto entre todos os pontos.

No entanto, apesar de se tratar de um algoritmo simples de implementar e possível de otimizar, no caso de se pretender encontrar o caminho mais curto em trajetos de longa distância pode demorar muitos segundos ou até minutos. Isto porque o algoritmo tem como base fazer uma procura circular em torno do vértice de partida, pelo que no caso de tratar de um grafo com grandes dimensões (como é esperado neste contexto), se os vértices de partida e não estiverem muito próximos, vai ser percorrida uma data de vértices sem importância (podendo até haver um crescimento exponencial).

Com vista a minimizar estas desvantagens trazidas pelo algoritmo de **Dijkstra**, vão ser ainda considerados outros algoritmos: **Dijkstra bidirecional** e **A***.

Algoritmo de Dijkstra

O algoritmo de Dijkstra criado por Edsger W. Dijkstra em 1956, apresenta várias variantes, sendo que o objetivo do algoritmo original é encontrar o caminho mais curto entre apenas dois pontos. No entanto, a variante mais comum é aquela cujo objetivo é encontrar o caminho mais curto desde o vértice de origem (padaria) até todos outros vértices no grafo, produzindo árvores de caminhos mais curtos. É também importante referir que este algoritmo é aplicado a árvores dirigidas, como acontece neste projeto, ou a árvores não-dirigidas onde todas as arestas apresentam valores positivos para o peso.

```
Dijkstra(G, s): // G = (V, E), s ∈ V
1. for each v ∈ V do:
2.     dist(v) ← ∞
3.     path(v) ← nil
4.
5.     dist(s) ← 0
6.     Q ← ∅ // min-priority queue
7.     INSERT(Q, (s, 0)) // inserts s with key 0
```

```

8.   while Q ≠ Ø do:
9.     v ← EXTRACT-MIN(Q) // greedy
10.    for each w ∈ Adj(v) do:
11.      if dist(w) > dist(v) + weight(v, w) then:
12.        dist(w) ← dist(v) + weight(v, w)
13.        path(w) ← v
14.        if w ∉ Q then: // old dist(w) was ∞
15.          INSERT(Q, (w, dist(w)))
16.        else w ∈ Q then: // old dist(w) was ∞
17.          DECREASE-KEY(Q, (w, dist(w)))

```

Como podemos observar no pseudocódigo, de forma a manter um registo de quais vértices processar a seguir, é utilizada uma fila de prioridade que, seguindo uma abordagem gananciosa, deve ser mutável para que seja possível maximizar o ganho imediato (neste contexto: minimizar o tempo gasto na viagem).

Além disso, cada vértice devem ter os seguintes dois campos:

- **path** - vértice que permite chegar ao vértice atual através do caminho mais curto;
- **dist** - duração mínima até à origem.

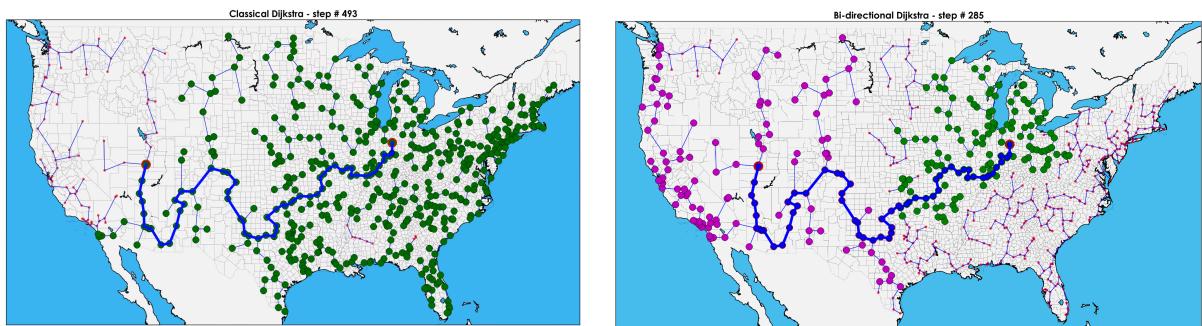
O algoritmo divide-se então em duas fases: **preparação** (de 1 a 6 no pseudocódigo) e **pesquisa** (restante parte do pseudocódigo).

Numa primeira etapa realiza-se a preparação que consiste em inicializar todos valores *path* e *dist* dos vértices do grafo. Uma vez que isto é feito percorrendo todos os vértices do grafo, este passo é exequível em **tempo linear**, $O(|V|)$, onde V corresponde ao número de vértices pertencentes ao grafo.

De seguida, é realizado o próximo passo: pesquisa, onde mais uma vez são percorridos todos os vértices, mas desta vez são também percorridos todas as arestas, o que para já resulta numa complexidade temporal de $O(|V| + |E|)$, onde V representa novamente o número de vértices e E o número de arestas. Ora, na fila de prioridade podem ser ainda realizadas operações de inserção, extração ou *decrease-key* por cada passo, o que resulta em $O(\log|V|)$, já que o tamanho máximo da fila corresponde ao número total de vértices. Assim, a complexidade temporal total relativa à pesquisa é de $O((|V| + |E|) * \log(|V|))$.

Algoritmo de Dijkstra bidirecional

Anteriormente foi dito que existem várias variantes do algoritmo de Dijkstra e esta é o exemplo de uma. No caso do algoritmo de Dijkstra previamente visto, a pesquisa era feita apenas numa direção, o que no caso de mapas de grandes dimensões e pontos de partida e chegada resultaria em fracos resultados.



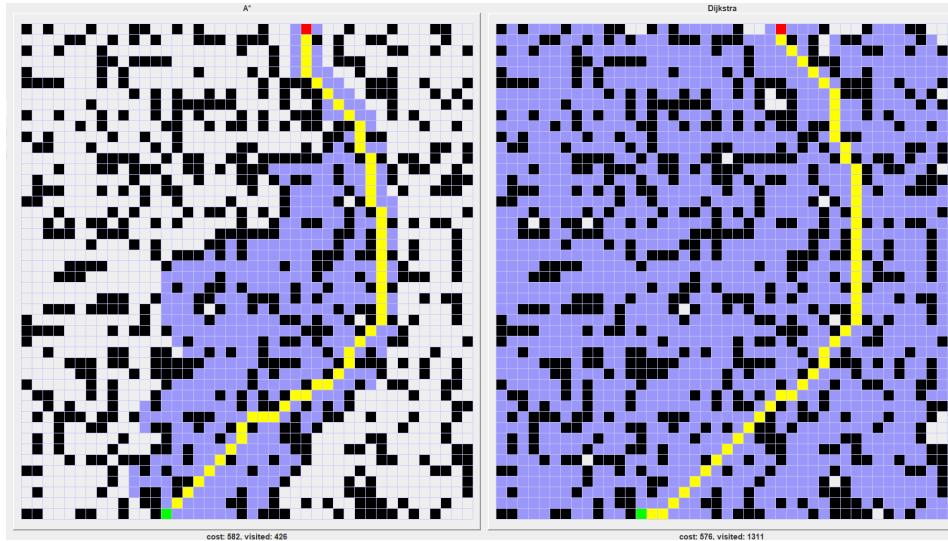
Então, esta variante aparece de forma a obter melhores resultados apresentando uma complexidade temporal um pouco melhor: $O(E + V * \log(V))$. Esta melhoria resulta de uma pesquisa simultânea em dois sentidos, ou seja, é realizada uma pesquisa em frente a partir do vértice de origem e outra para trás partindo do vértice de destino. O algoritmo termina quando as duas pesquisas se encontram a si mesmas num ponto intermédio, sendo possível assumir então que existe um caminho entre os dois vértices.

No exemplo das imagens acima observa-se que para o algoritmo de Dijkstra *clássico* foram necessários 493 passos, enquanto que o algoritmo bidirecional de Dijkstra só necessitou de 285 passos que é aproximadamente metade dos utilizados no primeiro algoritmo, sendo de facto mais eficiente.

Algoritmo A*

Peter Hart, Nils Nilsson e Bertram foram os primeiros a publicar o algoritmo A* em 1968. Este algoritmo pode ser visto como uma extensão do algoritmo de Dijkstra mas com uma maior eficiência (principalmente em grafos mais densos, como é o caso deste projeto), utilizando heurística.

Estes melhores resultados por parte deste algoritmo são então resultado da análise de menor número de vértices, já que estes são previamente selecionados como sendo “promissores” ou “não promissores”. Isto é feito através do cálculo da distância euclidiana entre dois vértices a partir de uma função heurística, que é um valor fácil de obter num mapa.



É possível notar que o número de vértices visitados no caso do algoritmo A* foi de 426, ao passo que no caso do algoritmo de Dijkstra foram visitados mais do dobro dos vértices: 1131, provando-se que efetivamente o algoritmo A* é mais eficaz.

No entanto, é muito importante notar que o custo do algoritmo A* é de 582, sendo superior ao de Dijkstra que é de 576. Ora, isto vem aqui realçar o facto de que **não é garantido que o algoritmo A* obtenha a solução mais ótima**.

3.5. Caminho mais curto entre todos os pares de vértices

Nesta fase surgem os algoritmos de **Dijkstra** para cada vértice e o de **Floyd-Warshall**, com o intuito de aumentar significativamente a eficiência do projeto ao realizar o pré-cálculo dos caminhos de duração mínima entre os pares de vértices (tal como foi referido anteriormente na etapa do pré-processamento).

Algoritmo de Dijkstra

No ponto anterior este algoritmo já foi analisado concluindo-se que apresenta uma complexidade temporal de $O((|V| + |E|) * \log(|V|))$. No entanto, agora surge a necessidade de calcular o caminho mais curto para todos os pontos, ou seja, iterar mais uma vez todos os vértices ($|V|$). Assim, a complexidade temporal passa a ser de $O((|V| + |E|) * \log(|V|) * |V|)$.

No caso de grafos onde se tem $|E| \approx |V|$, isto é, no caso de grafos esparsos, que geralmente estão associados a redes viárias, a complexidade passa a ser $O(|V^2| * \log(|V|))$. Logo, perante estas situações torna-se mais eficaz.

Algoritmo de Floyd-Warshall

O algoritmo de **Floyd-Warshall**, publicado em 1959 por Bernard Roy e em 1962 por Robert Floyd e Stephen Warshall, baseia-se no conceito de programação dinâmica utilizando duas matrizes de adjacências: uma para registar os valores de distância mínima e outra para guardar o predecessor do caminho mais curto.

Apresenta uma complexidade temporal de $\Theta(|V|^3)$, tratando-se de um algoritmo mais favorável para grafos onde se verifique $|E| \approx |V|^2$, ou seja, grafos densos. Porém, é importante notar que mesmo em grafos pouco densos pode ser vantajoso por corresponder a um código simples.

3.6. Caminho mais curto, partindo da padaria, passando por todos os clientes

Voltando de novo ao cerne do problema, este tem como objetivo entregar todas as encomendas de pão. Isto deve ser feito da seguinte forma: o Sr. Sílvio parte da sua padaria e regressa de novo a esta quando já não houverem mais entregas para fazer, notando-se que há circularidade no caminho feito pelo padeiro.

Portanto, tal como já foi referido anteriormente na fase da exposição dos problemas encontrados, estamos perante um problema semelhante ao **Problema do Caixeiro Viajante (Travelling Salesman Problem)** que se trata de um problema NP-difícil.

Uma vez neste projeto é usado um grafo dirigido, o Sr. Sílvio poderá partir da sua padaria seguindo um caminho e regressar seguindo outro, pelo que as distâncias podem ser diferentes, pelo que se trata de uma variante assimétrica do problema.

Para resolver este problema existem dois tipos de algoritmos: **algoritmos exatos** e **algoritmos aproximados**.

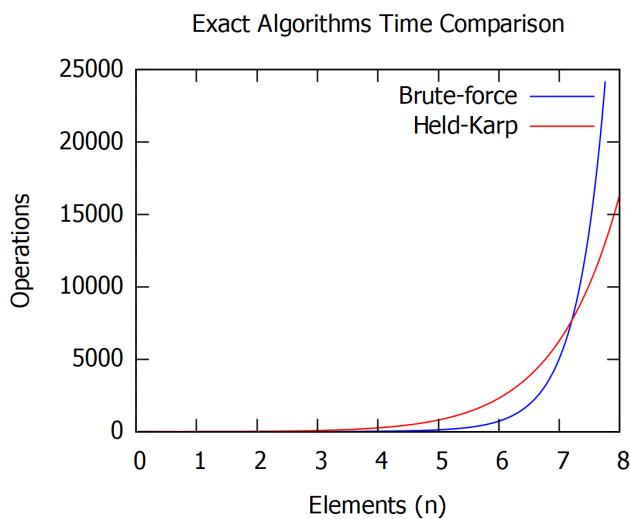
Algoritmos Exatos

A solução mais direta para este problema seria usar **Brute-Force** de forma a explorar todas as combinações possíveis de forma ordenada, o que resultaria numa

complexidade temporal de $O(n!)$. No entanto, mesmo para pequenos números de vértices este algoritmo torna-se absurdamente dispendioso.

Surge então o algoritmo de **Held-Karp** fundamentado na programação dinâmica e que resolve o problema numa complexidade de $O(n^2 2^n)$. Tal acontece porque ao invés de ser usada uma abordagem do tipo *top-down* como no caso do algoritmo anterior, é usada uma abordagem *bottom-up*, onde toda a informação intermediária necessária para resolver o problema é apenas desenvolvida **uma e uma só vez**.

De forma a comparar a complexidade destes dois algoritmos construiu-se o gráfico se apresenta abaixo, onde é possível verificar que o algoritmo *Brute-Force* é mais eficiente para casos em que $n < 8$, mas a partir desse valor é mais rápido o algoritmo de *Held-Karp*.



Apresenta-se de seguida uma tabela com o número de operações a realizar em função do número de elementos para ambos os algoritmos, onde mais uma vez se confirma que $n = 8$ é ponto de viragem entre as eficiências dos algoritmos.

n	Brute-Force	Held-Karp
1	1	2
2	2	16
3	6	72
4	24	256
5	120	800
6	720	2304

7	5040	6272
8	40320	16384
9	362880	41472
10	3628800	102400

Deste modo, de forma a obter a melhor solução possível a estratégia a seguir seria implementar o algoritmo em função do número de vértices a visitar:

- algoritmo de *Brute-Force* para $n < 8$;
- algoritmo de *Held-Karp* para $n \geq 8$.

Algoritmos Aproximados

Nesta categoria de algoritmos, aquele que se destaca mais é o algoritmo **Nearest Neighbour** cujo princípio é partir de um vértice escolhido aleatoriamente e ir repetidamente visitando o vértice mais próximo até que todos tenham sido visitados.

De facto, trata-se de um algoritmo fácil de implementar e que é rapidamente executado, mas o seu resultado varia muito já que não parte sempre do mesmo vértice e normalmente não obtém o caminho mais curto, apenas obtém um caminho curto graças à sua natureza gananciosa.

Ora, visto que neste algoritmo o vértice de partida é escolhido de forma arbitrária e no caso deste projeto o problema apresenta sempre o mesmo vértice de origem (a padaria do Sr. Sílvio), não é possível, é aqui eliminada a aleatoriedade na escolha do vértice de partida.

As possíveis soluções aqui apresentadas terão de ser posteriormente adaptadas de forma a satisfazer a hora preferencial de cada entrega, respeitando sempre as tolerâncias de antecedência e atraso da forma mais equilibrada possível.

4. Casos de Uso

A interação do utilizador com o programa a realizar neste projeto será feita através de uma interface simples de texto baseada em menus com diferentes opções.

Também será guardado o mapa definido pelo utilizador, a partir do qual serão obtidas os seguintes resultados:

- visualização do grafo completo relativo ao mapa através do *GraphViewer*;
- obtenção do caminho de menor duração entre dois pontos;
- construção da rota que passa por todos os clientes, utilizando apenas uma carrinha de capacidade ilimitada;
- análise da acessibilidade da carrinha nos diferentes pontos.

Além disso, permitirá igualmente realizar as seguintes tarefas:

- carregamento de um mapa real;
- inserção e eliminação de locais com encomenda;
- alteração da morada da padaria;
- alteração da quantidade de uma encomenda de um determinado cliente;
- alteração de zonas obstruídas (ruas desimpedidas).

Considerando as melhorias apresentadas na próxima secção poderá implementar ainda as seguintes funcionalidades:

- construção da rota que passa por todos os clientes, utilizando N carrinhas de capacidade ilimitada ou limitada.

5. Melhorias na Aplicação

5.1. Existência de uma frota de Carrinhas

O Sr. Sílvio expande o seu negócio e passa a ter uma frota de carrinhas para fazer as suas encomendas. Cada carrinha fica responsável por um conjunto de clientes fixo, em função da área de residência onde estes se encontram. Ou seja, de forma a maximizar o tempo necessário para realizar todas as entregas, o grafo inicial com todos os clientes é **dividido por zonas de residência, tantas quantas forem as carrinhas disponíveis**, em função da proximidade entre os mesmos e conectividade das zonas adjacentes a estes.

O objetivo é que na mesma zona se encontrem as pessoas que vivem mais próximo entre si, sendo que também deve ser possível passar por todos os vértices da zona, partindo da padaria e voltando a esta. Por outras palavras, é importante que a zona definida também seja conexa.

$$\forall c \in Carrinhas, c \rightarrow G_c \subseteq G$$

Após a alocação de cada carrinha a uma das zonas encontradas (notando-se que qualquer carrinha pode ficar responsável por qualquer zona), o itinerário a ser realizado por cada uma será determinado através da aplicação solução inicial para cada carrinha da frota:

1. **for each** $c \in Carrinhas$ **do**:
2. **percurso** \leftarrow minPercursoPreferencial(c , $c.zona$, start);

É importante salientar que todas as encomendas terão **apenas um vértice comum: o vértice inicial de partida** (loja do Sr. Sílvio). Além disso, também partilham o facto da capacidade ser ilimitada assim como a hora de saída e tempo de pausa.

$$\forall c \in Carrinhas, G_{ci} \cap G_{cj} = \{start\}, \forall i, j, 0 < i, j < N, i \neq j$$

5.2. Condutores afetam o tempo de entrega

Eventualmente, cada condutor de carrinha, por ser mais ou menos conversador, pode demorar diferentes quantidades de tempo diferentes para efetuar uma entrega depois de chegar à morada de um cliente. Assim, o tempo necessário para a realização de uma encomenda por parte de uma carrinha é afetado também por este.

Para tal, é necessário adicionar a uma carrinha um campo condutor que guarde o nome deste e o tempo que este demora em cada morada (os vértices). Ao contabilizar o tempo de paragem em cada vértice, ao invés de adicionar 5 minutos (tempo considerado constante inicialmente pelo condutor da única carrinha existente), adiciona-se o tempo do condutor respetivo:

Percorso(carrinha, G, start):

1. (...)
2. **if** (condutor para em cliente) **then**:
3. tempoTotal ← tempoTotal + carrinha,tempoParagem();
4. (...)

De forma a aumentar a flexibilidade da solução inicialmente consideramos que uma carrinha tem um condutor por omissão, cujo tempo de pausa é exatamente 5 minutos. Com isto pretende-se notar que a mudança é efetivamente realizada apenas na seção de pré-processamento dos dados e não no algoritmo da solução.

Esta alínea poderá também afetar a alocação das zonas de entrega para cada carrinha, na medida em que a carrinha cujo condutor demora mais tempo fica responsável pela zona com menor número de vértices (não é uma situação determinista para minimizar o tempo de entrega, pois não tem em conta o tempo entre os vértices, mas sim um caso médio do problema - *Greedy Algorithm* - sem exageradamente aumentar a complexidade).

```
// Carrinhas ordenadas ascendentemente pelo tempo de pausa do seu condutor
```

AlocarZonas(zonas, carrinhos):

1. vector<pair<int, *zona>> clientesZona;
2. **for each** (z ∈ zonas) **do**:
3. nClientes ← z.numeroClientes();
4. clientesZona.add(nClients, &z);
5. sort(clientesZona); // ordenar de forma decrescente pelo número de clientes
6. **for each** (cz ∈ clientesZona) **do**:
7. carrinha.next().setZona(get<1>(cz));

5.3. Carrinhas com capacidade limitada

As carrinhas podem ter capacidades máximas para transportar os produtos da padaria, sendo esta medida em número de pães. Neste caso, os clientes passam a ter associadas uma encomenda, que corresponde uma quantidade de pães.

Mais uma vez, esta mudança apenas afetará a alocação das zonas a carrinhas. Uma carrinha era inicialmente descrita como tendo capacidade ilimitada sendo que a alocação descrita acima não tem em conta este campo.

Como revisão, no caso mais simples, após a definição das zonas por proximidade e conectividade dos clientes, qualquer carrinha pode ser atribuída a qualquer zona, uma vez que a capacidade é tomada como ilimitada e todos os condutores (os por defeito) demoram 5 minutos. Ao acrescentar variação de tempo entre condutores, a carrinha associada ao condutor com maior tempo de pausa estará associada à zona com menor número de clientes (minimizar o tempo em pausas). Ao acrescentar uma distinção entre carrinhas pela sua capacidade é necessário primeiro verificar se existe uma carrinha que apenas poderá circular numa zona específica e, em caso de empate, escolher segundo o tempo de pausa, como descrito anteriormente.

$$(\forall c \in Carrinhas, \forall z \in Zonas, \exists!z : c.capacidade \geq z.totalEncomendas())$$

Por fim, é relevante referir ainda que não se considera a possibilidade de ser necessária a alteração da definição das diferentes zonas, já que se parte do pressuposto que existem carrinhas com capacidade suficiente para satisfazer as necessidades de todas as zonas.

5.4. Utilização de mapas reais

A definição do grafo inicial poderá ser realizada tendo por base mapas reais, extraídos do *OpenStreetMaps*, localizando nestas as moradas de entrega dos clientes registados no Sr. Sílvio. Tal, permitirá-nos-á perceber a dimensão e complexidade da solução em termos práticos e a influência do tamanho do grafo na execução da mesma.

6. Objetivos do Trabalho

Em suma, o objetivo é criar um itinerário que permita ao Sr. Sílvio realizar todas as entregas no menor tempo possível, respeitando as horas preferenciais dos clientes.

Começamos por dividir o problema em três iterações, tendo em conta o contexto e aumentando o grau de complexidade em cada uma. Para além destas, são também apresentadas as melhorias adicionais que tencionamos implementar: incluir uma frota de carrinhas, considerar um tempo variante por condutor na realização da entrega, considerar a possibilidade das carrinhas terem capacidade limitada e por fim a utilização de mapas reais.

Os problemas de resolução mais complexa surgem na segunda iteração, quando o objetivo deixa de ser apenas encontrar o caminho mais curto, e passa a incluir o equilíbrio dos atrasos nas encomendas. Torna-se, assim, de extrema importância respeitar a prioridade destas. O problema apresentado é semelhante ao **Problema do Caixeiro Viajante**, mas adaptado de forma a ter em conta as janelas de tempo preferenciais para cada encomenda.

Com a finalidade de resolver o problema proposto foram avaliados vários algoritmos, para diferentes fases e com diferentes objetivos, de entre os quais se destacam: **depth-first search**, **breadth-first search**, **baseado em conjuntos disjuntos**, **Dijkstra**, **A***, **Floyd-Warshall**, **Nearest Neighbour**, **Held-Karp**.

Foram ainda aplicados vários conceitos lecionados nas aulas relativos à concepção dos algoritmos que foram explorados, nomeadamente **brute force**, **algoritmos gananciosos**, **retrocesso** e **recursividade**.

Em conclusão, o nosso principal objetivo é implementar os algoritmos apresentados de modo a diminuir ao máximo a complexidade da solução, com o intuito de obter o caminho de menor duração que passa por todos os pontos de entrega.

Mestrado Integrado em Engenharia Informática e Computação
Ano Lectivo 2020/2021

Concepção e Análise de Algoritmos: O Padeiro da Vila em Época de Covid (Tema 9)

Implementação efetiva da solução

Turma 7 - Grupo 4

Ângela Filipa Ribeiro Coelho	up201907549
Marta Cristina dos Santos Mariz	up201907020
Patrícia do Carmo Nunes Oliveira	up201905427

7. Principais funcionalidades e cenários efetivamente implementados

Segundo a ordem previamente proposta na perspectiva de solução, as primeiras tarefas realizadas foram a implementação de todos os componentes base necessários para identificar a padaria do senhor Sílvio e tudo o que a constitui, como carrinhas, clientes e encomendas. Em simultâneo, estava a ser desenvolvida a estrutura do grafo que serviria como ferramenta principal para a análise da zona de entregas e escolha do melhor caminho para proceder à mesma.

De forma a facilitar a visualização do mesmo com os resultados obtidos, com recurso ao GraphViewer, fornecido pelo docente, foram desenvolvidos os métodos para a geração e desenho do mesmo. Neste aspeto, foi importante igualmente criar os métodos necessários para ler os dados para tal dos ficheiros. Bem como, desenhar e gerar o graphViewer a partir do respetivo grafo. Realçando que, para este projeto consideramos que a “Vila da Tocha” corresponde ao mapa do Porto.

Tendo em conta as funcionalidades e cenários do problema apresentado, efetivamente foram implementadas as seguintes:

- Visualização do mapa original, sem qualquer processamento;
- Análise da conectividade do grafo tendo em conta um ponto de partida, utilizando o algoritmo de Pesquisa em Profundidade, posterior eliminação dos vértices e arestas não visitadas e desenho desta componente fortemente conexa;
- Pré-processamento das encomendas, pela remoção daquelas cujas moradas não se encontram na localidade acessível pela padaria;
- Obtenção do caminho mais curto entre dois pontos do grafo através do algoritmo de Dijkstra ou A*;
- O caminho mais curto de uma única carrinha com capacidade ilimitada que passe por todas as entregas da componente conexa, começando e acabando na padaria, tal como descrito na 1^a iteração, utilizando a estratégia de *Nearest Neighbour*;
- O caminho mais curto de uma única carrinha com capacidade ilimitada que passe por todas as entregas da componente conexa, começando e acabando na padaria, mas considerando as horas presenciais das entregas e tolerância de antecipação e atraso, utilizando Brute Force, e considerando a rota mínima corresponda aquela que leva ao mínimo tempo de espera médio entre todos os clientes (2^o e 3^a iteração) - procurar minimizar e equilibrar os tempos de atraso.

8. Estruturas de dados utilizadas

Com o objetivo de manter o código organizado e perceptível, as classes e ficheiros foram divididos em diretórios, de acordo com a sua função.

Para facilitar a manipulação do tempo criamos uma classe chamada *Time* (horas e minutos), que trata das tarefas relacionadas com este, como conversões, adições e comparações. Para além desta, as principais classes em que se baseia o nosso modelo são *Van*, *Order* e *Client*. Cada carrinha tem um vetor de encomendas pela qual é responsável. Uma encomenda tem uma hora preferencial de entrega e um cliente associado. Estes dados são obtidos pela leitura dos respetivos ficheiros, através das funções do *FileReader*.

O mapa inicial é construído a partir da leitura de ficheiros que indicam as características do grafo a que este corresponde. O grafo é construído tendo em conta os vértices e as arestas associadas, e tem o auxílio de uma fila de prioridade mutável para ser necessária para os algoritmos implementados. Cada vértice é identificado por um número único, uma posição (coordenadas x e y) e um vetor de arestas adjacentes. Cada aresta terá um peso associado, que corresponde ao tempo que demora a ser percorrida.

O mapa será sempre desenhado com recurso ao *MapDrawer* que implementa os métodos necessários para a visualização do *GraphViewer* que representa o grafo desejado.

A classe *VanDeliveryPlanner* irá receber o grafo previamente processado, o local da padaria e a carrinha, pré processar as encomendas, eliminando as que não são válidas na componente fortemente conexa da padaria e com base na área e encomendas a serem entregues por essa carrinha calcular o melhor itinerário para a mesma, podendo este ter ou não ter em conta as janelas de tempo preferenciais para cada entrega.

O namespace *Operations* contém as chamadas para as funcionalidades principais implementadas enunciadas no capítulo anterior, que são chamadas a partir da função *main* da aplicação tendo em conta os argumentos recebidos.

9. Algoritmos Implementados (pseudocódigo)

De seguida, serão listados os algoritmos principais implementados em pseudocódigo, sendo estes:

- Análise da conectividade;
- Pré-processamento dos dados;
- Viagem de uma carrinha sem considerar o intervalo de tempo para entrega;
- Viagem de uma carrinha considerando o intervalo de tempo para entrega.

9.1. Análise da conectividade

```
1 // Gi - graph
2 // s - start vertex
3 DFS_CONNECTIVITY(Gi, s):
4     for each v in vertex_set(Gi) do
5         visited(v) = false
6         DFS_VISIT(s)
7
8 // Gi - graph
9 // v - current vertex
10 DFS_VISIT(Gi, v):
11     visited(v) = true
12     for each e in adj(v) do
13         if not visited(dest(e)) then
14             DFS_VISIT(dest(e))
```

9.2. Pré-processamento dos dados

```
1 // Gi - graph representing the map
2 // Oi - sequence of orders
3 // b - bakery
4 PREPROCESS_DATA(Gi, Oi):
5     DFS_CONNECTIVITY(Gi, b)
6     REMOVE_UNVISITED_VERTICES(Gi)
7
8     for each o in Oi do
9         if FIND_VERTEX(Gi, dest(r)) == NULL then
10            REMOVE_ORDER(Oi, o)
11
12     DIJKSTRA_SHORTEST_PATH(Gi)
```

9.3. Entrega de todas as encomendas

```
1 // Gi - graph representing the map
2 // Oi - sequence of orders
3 // v - van to be used
4 // b - bakery
5 PLAN_VAN_DELIVERY_WITHOUT_TIME_WINDOW(Gi, Oi):
6     PREPROCESS_DATA(Gi, Oi)
7
8     // Get Path from bakery
9     closest = GET_NEAREST_NEIGHBOR(b)
10    path_edges = GET_PATH_EDGES(Gi, b, closest)
11    REMOVE_ORDER(Oi, closest)
12
13    current = closest
14    while !Oi.empty
15        // Get Path between closest orders
16        closest = GET_NEAREST_NEIGHBOR(current)
17        path_edges += GET_PATH_EDGES(Gi, current, closest)
18        REMOVE_ORDER(Oi, closest)
19
20        current = closest
21
22    // Get Path back to bakery
23    path_edges += GET_PATH_EDGES(Gi, current, b)
```

9.4. Entrega de encomendas num intervalo de tempo

```
1 // Gi - graph representing the map
2 // Oi - sequence of orders
3 // v - van to be used
4 // b - bakery
5 // startTime - time that leaves the bakery
6 PLAN_VAN_DELIVERY_WITH_TIME_WINDOW(Gi, Oi):
7     PREPROCESS_DATA(Gi, Oi)
8
9     minAverageWaitTime = INF
10    for each o in Oi do
11        tmp_orders = Oi
12        REMOVE_ORDER(tmp_orders, o)
13
14        path = GET_PATH_EDGES(Gi, b, o)
15        arrival = startTime + CALCULATE_PATH_TIME(path)
16        waitTime = CALCULATE_MIN_WAIT_PATH_IN_INTERVAL(o, arrival, tmp_orders, path)
17
18        averageWaitTime = waitTime / SIZE(Oi)
19        if (averageWaitTime < minAverageWaitTime)
20            minAverageWait = averageWaitTime
21            finalPath = path
```

(Parte 1)

```

23
24
25 CALCULATE_MIN_WAIT_PATH_IN_INTERVAL(current, arrival, remaining_orders, path)
26     if (arrival > current.preferredTime + MAX_ARRIVAL_TIME)
27         return INF;
28
29     waitTime = |arrival - current.preferredTime|
30     if (remaining_orders.empty())
31         path += GET_PATH_EDGES(Gi, current, b)
32         return waitTime
33
34     if (arrival < current.preferredTime - MIN_ARRIVAL_TIME)
35         arrival = current.preferredTime - MIN_ARRIVAL_TIME
36
37     arrival += v.deliveryTime
38     minWaitTime = INF
39     for each o in remaining_orders do
40         tmp_orders = remaining_orders
41         REMOVE_ORDER(tmp_orders, o)
42
43         nextPath = GET_PATH_EDGES(Gi, current, o)
44         arrivalTime = arrival + CALCULATE_PATH_TIME(path)
45         waitTime = CALCULATE_MIN_WAIT_PATH_IN_INTERVAL(o, arrivalTime, tmp_orders, nextPath)
46
47         if (waitTime < minWaitTime)
48             minWaitTime = waitTime
49             minPath = nextPath
50
51     path += minPath

```

(Parte 2)

10. Análise da complexidade dos algoritmos

10.1. Análise Teórica (temporal e espacial)

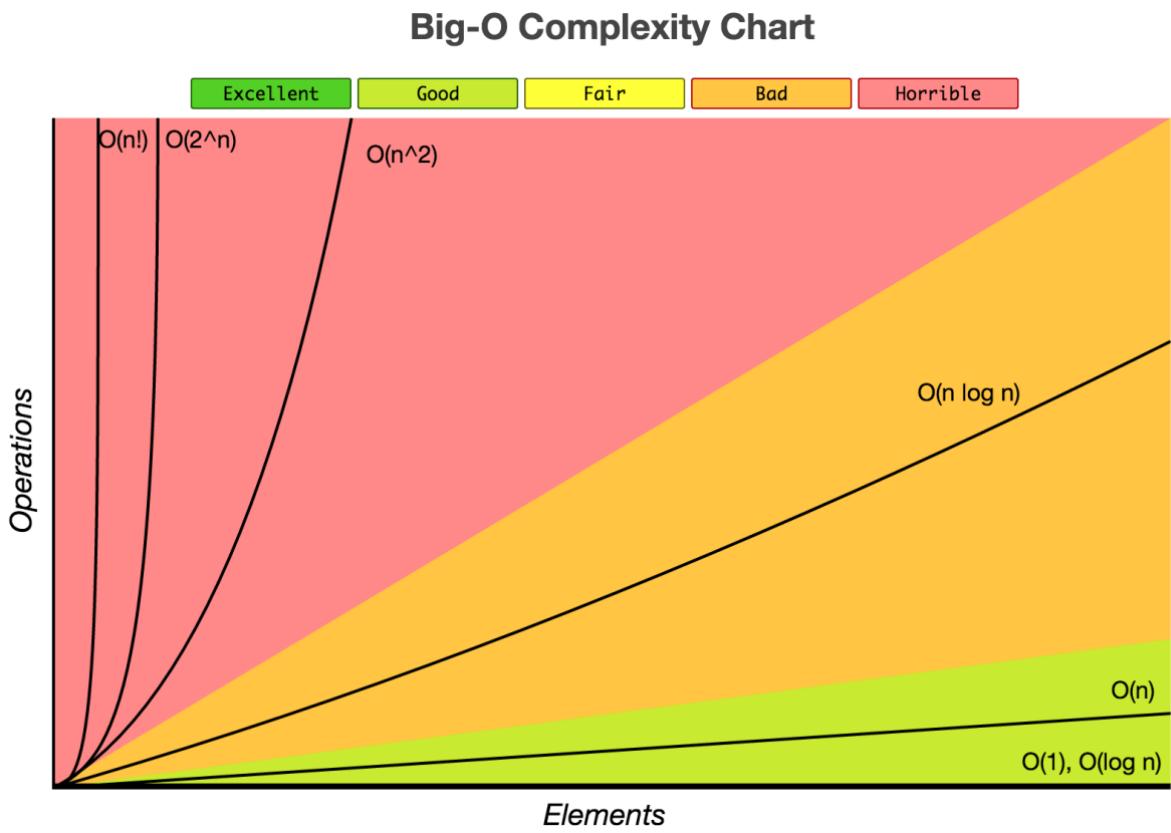
A análise teórica da complexidade dos diferentes algoritmos implementados já tinha sido explorada aquando a discussão sobre os mesmos, incluindo para efeitos de comparação. Globalmente foram escolhidos e implementados os que apresentavam melhores resultados para grandes quantidades de dados a processar, associados a estruturas de dados que tentam otimizar este, como por exemplo ao usar uma função hash na distinção entre vértices de forma a melhorar a pesquisa dos mesmos no grafo.

10.2. Análise Temporal Empírica

Temos a destacar, para além do já enunciado na parte 1 do projeto, que a performance da nossa aplicação é exponencialmente reduzida com o aumento de encomendas a entregar, principalmente ao considerarmos as janelas de tempo em que estas devem ser entregues aos clientes. Tal advém, do facto de, nesta perspetiva ter se optado por uma abordagem *Brute Force*, onde se analisa todas as possíveis ordens para a entrega das encomendas escolhendo aquela que leva a que os clientes esperem em média menos, ou seja, onde a diferença entre a efetiva hora de entrega e a desejada do cliente seja inferior.

Ainda que, quando um caminho se torna inválido (o tempo de chegada ultrapassa a tolerância após o desejado) este deixa de continuar a ser processado, eliminando logo esta opção, as sequências que são efetivamente processadas são numerosas. Para além disso, entre cada um dos pontos de encomenda é igualmente necessário calcular o caminho mais curto entre estes.

As chamadas recursivas levam a uma grande complexidade espacial e o número total de possíveis sequências a um aumento exponencial da temporal com o aumento do número de entregas: $O(|V|!) + |V| * |E|$.



Assim, podemos verificar pelo gráfico que a situação anunciada acima é uma condição horrível para a aplicação e que prejudica muito o desempenho da mesma para grandes volumes de encomendas.

11. Análise da Conectividade do Grafo

Considera-se que a componente fortemente conexa de um grafo partindo de um ponto inicial é o conjunto de arestas e vértices que conseguem ser visitados a partir deste, numa pesquisa em profundidade.

Assim, como apresentado na secção 9.1, a análise da conectividade de um grafo a partir da padaria divide-se em 2 etapas: realização da pesquisa em profundidade a partir do vértice onde se localiza a padaria e posterior remoção dos vértices que não são alcançáveis e quaisquer arestas que tinham como destino algum dos vértices removidos.



(original)



(fortemente conexo)

12. Conclusões

Tendo em conta o trabalho final e tudo o que foi implementado podemos dizer que existe muito espaço para melhoria e para a aplicação crescer, principalmente pela aplicação de todas as possíveis melhorias anunciadas na primeira parte.

Globalmente, consideramos que o nosso trabalho é focado nos requisitos mínimos para o mesmo e que foram variadas as dificuldades aquando o desenvolvimento do mesmo, tornando-se difícil conseguir criar um ritmo de trabalho uniforme. Dizendo isto, em seguida são apresentados os aspetos que consideramos positivos e negativos do resultado final apresentado.

12.1. Aspetos positivos

- Estrutura organizada;
- Formatação do desenho do grafo;
- Análise eficaz da conectividade;
- Boa implementação do caminho mais curto que passa por todos os pontos de encomenda pela abordagem de *Nearest Neighbor*;
- Utilização do algoritmo A* após o primeiro processamento com o algoritmo de Dijkstra, o que resulta num aumento de eficácia;
- Dada a possibilidade de a carrinha definir o seu tempo de entrega em cada ponto.

12.2. Aspetos negativos

- Não é considerada a possibilidade de ruas obstruídas;
- Usado um algoritmo pouco eficaz na 2^a e 3^a iteração (*Brute Force*);
- O cálculo da rota da carrinha tendo em conta janelas de tempo não funciona a 100%;
- Não foram implementadas as melhorias inicialmente propostas como a existência de uma frota de carrinhas, ou carrinhas com capacidade limitada.

13. Contribuições

De seguida estão listadas as tarefas desenvolvidas onde cada um dos elementos teve uma maior percentagem de contribuição.

O elemento Ângela Coelho destacou-se nas seguintes secções:

- Construção da estrutura de dados *Time*;
- Geração de um grafo a partir dos dados de um ficheiro - desenvolvimento das classes *Position* e *GraphLoader*;
- Integração do **graphViewer** no projeto (desenhar a partir dos ficheiros ou a partir de um grafo) - desenvolvimento das classes *MapDrawer* e *MapLabel*;
- Implementação do algoritmo para análise da conectividade de um grafo;
- Criação das exceções *TimeException* e *FileNotFoundException*.

O elemento Marta Mariz contribui nos pontos seguintes:

- Construção da estrutura de dados *Client*;
- Pré-processamento das encomendas;
- Trabalho no relatório.

O elemento Patrícia Oliveira ficou encarregue dos seguintes tópicos:

- Construção das estruturas de dados *Order* e *Van*;
- Construção das classes *Graph*, *Vertex* e *Edge* e implementação dos métodos associados a estas;
- Pré-processamento do grafo;
- Implementação dos algoritmos Dijkstra e A*;
- Desenvolvimento dos algoritmos necessários para as 3 iterações do problema - classe *VanDeliveryPlanner*,
- Desenho e definição das principais operações que a aplicação permitirá realizar - métodos de *Operations*.

Globalmente distribuímos a percentagem de trabalho e esforço pelos elementos da seguinte forma:

- Ângela Coelho - 35%
- Marta Mariz - 20%
- Patrícia Oliveira - 45%

14. Bibliografia

- Slides utilizados nas aulas teóricas no âmbito da disciplina Concepção e Análise de Algoritmos
- Plain As Life: Compare A* with Dijkstra algorithm,
<http://plainaslife.blogspot.com/2015/02/compare-with-dijkstra-algorithm.html>
- Dijkstra's Shortest Path Algorithm, <https://brilliant.org/wiki/dijkstras-short-path-finder/>
- Difference and advantages between dijkstra & A star [duplicate],
<https://stackoverflow.com/questions/13031462/difference-and-advantages-between-dijkstra-a-star>
- Dijkstra vs Bi-directional Dijkstra Algorithm on US Road Network,
<https://www.youtube.com/watch?v=1oVuQsxkhY0>
- Dijkstra's algorithm, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Bidirectional Dijkstra Algorithms,
https://docs.oracle.com/cd/E56133_01/2.7.0/reference/algorithms/dijkstra_bidirectional.html
- A* search algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm
- Floyd–Warshall algorithm,
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- Travelling salesman problem,
https://en.wikipedia.org/wiki/Travelling_salesman_problem
- Held–Karp algorithm,
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#cite_note-3

- Held-Karp vs. Brute-Force complexities,
<https://stackoverflow.com/questions/40708916/for-tsp-how-does-held-karp-algorithm-reduce-the-time-complexity-from-brute-force>
- Nearest neighbour algorithm,
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- Vehicle routing problem, https://en.wikipedia.org/wiki/Vehicle_routing_problem
- Graph connectivity, https://algowiki-project.org/en/Graph_connectivity
- Disjoint set data structure,
<http://www.algorithmsandme.com/disjoint-set-data-structure/>
- An O(log n) parallel connectivity algorithm,
<http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1980/CS/CS0178.pdf>
- An Adaptive Parallel Algorithm for Computing Connected Components,
<https://arxiv.org/pdf/1607.06156.pdf>