O Padeiro da Vila em Época Covid

24 / 05 / 2021

—

Conceção e Análise de Algoritmos

—

Bruno Rosendo

Domingos Santos

João Mesquita

Índice

Descrição do Problema

O trabalho incide numa distribuição de pães, começando na padaria e passando pela casa de todos os clientes que aderiram. Por fim, teremos ainda de voltar à padaria.

Cada cliente indica a hora preferencial para a sua entrega, tendo uma margem de tolerância definida (antes e depois da hora). Quando o padeiro chega a uma habitação, demora uma quantidade de tempo definida a efetuar a entrega.

A solução ótima é uma rota a seguir, minimizando o tempo total do itinerário e equilibrando o tempo de atraso nas entregas. Para simplificar o problema, dividimo-lo em diversas fases:

1ª Fase: Minimizar o tempo do itinerário sem considerar hora de entrega e apenas uma carrinha de capacidade ilimitada

Numa primeira fase, despreza-se a hora preferível dos clientes e as entregas são todas feitas por uma única carrinha de capacidade ilimitada. Assim, o problema resume-se a encontrar o trajeto mais curto (neste caso, o menos demorado) que começa na padaria (vértice inicial), passa por todas as moradas dos clientes e volta à padaria.

É importante notar que as entregas só conseguem ser efetuadas se existir pelo menos um caminho que liga as moradas de todos os clientes e a padaria (tanto de ida como de volta), ou seja, todos os vértices de entrega devem fazer parte do mesmo componente fortemente conexo. Assim, é necessário avaliar a conetividade do grafo subjacente à zona considerada, com o fim de identificar moradas com pouca acessibilidade, cujas estradas podem ter sido obstruídas por imprevistos como obras públicas (se alguma morada não fizer parte deste componente, a entrega a esse cliente será cancelada).

2ª Fase: Entrega considerando a hora preferencial, equilibrando o tempo de atraso, com uma carrinha de capacidade ilimitada

Numa segunda fase, ter-se-á em consideração a ordem de entrega aos clientes, de modo a satisfazer a hora preferencial deles.

Como consequência desta nova restrição, teremos que minimizar dois parâmetros: o tempo total do itinerário e o tempo de atraso das entregas, sendo que se dará prioridade ao último critério, para que os clientes não tenham de esperar mais do que precisam.

3ª Fase: Entrega com múltiplas carrinhas de capacidade limitada

Numa última fase, passa-se a considerar múltiplas carrinhas com capacidade limitada, ou seja, cada cliente pede X pães e cada carrinha entrega Y pães, sendo que o objetivo é minimizar o número de carrinhas utilizadas. Assim sendo, no início do algoritmo, cada carrinha será alocada aos seus respetivos clientes.

Com esta adição, temos três critérios de minimização, sendo eles, por ordem decrescente de prioridade: número de carrinhas alocadas, tempo de atraso das entregas e tempo total do itinerário.

Além disso, como cada carrinha tem um condutor diferente, o tempo entrega de cada um também poderá ser diferente.

Formalização do Problema

Dados de Entrada

Ci- Sequência de carrinhas na padaria, sendo Ci(i) o seu i-ésimo elemento. Cada um é caracterizado por:

- Q- quantidade total de pães que a carrinha consegue transportar (infinito nas 1ª e 2ª fases)
- T- tempo que o condutor demora a fazer a entrega

Gi = (Vi, Ei) – Grafo pesado (dirigido ou não), composto por:

- V- vértices, representando pontos da rede rodoviária, com:
  - ID- identificador do vértice
  - Adj ⊆ E, arestas que partem do vértice
  - Lat- latitude real do ponto no mapa
  - Long- longitude real do ponto no mapa
- E- arestas, representando estradas da rede rodoviária, com:
  - ID- identificador da aresta
  - W- Peso da aresta, que no contexto do projeto representa o tempo aproximado que se demora a percorrer a aresta
  - Dest ∈ V- destino da aresta

Ui- Sequência de clientes da padaria, sendo Ui(i) o seu i-ésimo elemento. Cada um é caracterizado por:

- Name- nome do cliente
- ID- identificador do cliente
- Addr- morada do cliente (deve pertencer aos vértices do grafo)
- H- hora da entrega
- Q- quantidade de pães na encomenda

S ∈ Vi - vértice da padaria (inicial)
Ra – Raio de ação dos veículos (centro na padaria)
Upper- Tolerância de atraso nas entregas
Lower- Tolerância de adiantamento nas entregas

Dados de Saída

Gf = (Vf, Ef) – grafo pesado (dirigido ou não), tendo Vf e Ef os mesmos atributos que Vi e Ef (à excessão de atributos utilizados pelos algoritmos).

Cf- Sequência de carrinhas com a informação das entregas realizadas, sendo Cf(i) o seu i-ésimo elemento. Cada carrinha é composta por:

- Qf- quantidade final de pães (sobras)
- Qe- quantidade entregue de pães
- T- Tempo total do itinerário
- Ta- Tempo total de atraso nas encomendas
- Uf (u ∈ Ui | 1 <= j <= |Uf|) - sequência de clientes a quem a carrinha realizou entregas, sendo Uf(i) o seu i-ésimo elemento. Cada cliente é caracterizado por:
  - Name- nome do cliente
  - Addr- morada do cliente
  - Ht- hora marcada para a encomenda
  - Hp- hora real da encomenda
- I (i ∈ Ei | 1 <= j <= |I|) – sequência de arestas a percorrer no itinerário, sendo P(j) o seu j-ésimo elemento.

Restrições

Aos dados de entrada:

- ∀ e ∈ Ei, w(e) > 0, visto que w representa o tempo necessário para percorrer uma aresta.
- Ra > 0, visto este valor representar a zona de entregas.
- S ∈ Vi, visto que a padaria tem que pertencer ao grafo.
- Upper > 0 e Lower > 0, visto que é virtualmente impossível chegar a todos os locais no instante exato que foi marcado.
- ∀ c ∈ Ci, Q(c) > 0 e T(c) > 0, visto que não faz sentido usar uma carrinha que não tem pães para entregar e essas entregas não podem ser instantâneas.
- ∀ u ∈ Ui, Q(u) > 0, visto que não faz sentido haver encomendas em que não se entregam pães.

Nota: Não é obrigatório a morada do cliente fazer parte do grafo e da zona de entrega, mas, se tal acontecer, essas encomendas serão canceladas. O mesmo acontece com moradas fora da mesma componente fortemente conexa do grafo.

Aos dados de saída:

- |Cf|<=|Ci|, visto que pode não ser preciso usar todas as carrinhas.
- ∀ vf ∈ Vf, ∃ vi ∈ Vi tal que vi e vf têm os mesmos atributos (à exceção de atributos utilizados pelos algoritmos).
- ∀ ef ∈ Ef, ∃ ei ∈ Ei tal que ei e ef têm os mesmos atributos (à exceção de atributos utilizados pelos algoritmos).
- ∀ cf ∈ Cf,
    - Qe(cf) > 0
    - T(cf) > 0
    - Ta(cf) >= 0
    - ∃ ci ∈ Ci tal que Q(ci) = Qf(cf) + Qe(cf)
    - ¬ ∃ c2 ∈ Cf tal que cf = c2
    - ∀ i ∈ [1, |Uf|-1], Hp(Uf[i]) < Hp(Uf[i+1])
- ∀ uf ∈ Uf,
    - ∃ ui ∈ Ui tal que name(uf)=name(ui), addr(uf)=addr(ui) e Ht(uf)=H(ui)
- Seja e1 o primeiro elemento de I, é necessário que e1 ∈ adj(S), pois cada carrinha parte da padaria.
- Seja ef o último elemento de I, é necessário que dest(ef)=S, pois cada carrinha regressa à padaria.

Funções Objetivo

A solução ótima do problema passa por minimizar o número de carrinhas usadas, o tempo de atraso das entregas e o tempo total do itinerário. Logo, é necessário minimizar as seguintes funções:

$f = |Cf|$
$g = \sum_{c \in Cf} Ta(c)$
$h = \sum_{c \in Cf} T(c)$

Tal como foi dito na descrição do problema, dar-se-á prioridade às funções na ordem f->g->h.

Perspetiva de Solução

Demonstram-se de seguida os principais obstáculos encontrados ao longo das diferentes etapas e as suas respetivas soluções. Serão identificadas as técnicas de conceção e os algoritmos a serem desenvolvidos.

Pré-Processamento dos Dados de Entrada

Tratamento do grafo

De modo a melhorar a eficiência temporal dos algoritmos que serão aplicados posteriormente, o grafo G deve ser pré-processado, reduzindo o seu número de vértices e arestas.

Inicialmente, devem ser eliminadas as arestas do grafo que se encontrem inacessíveis, devido a fatores externos, tal como vértices inacessíveis ou demasiado afastados do inicial, ou seja, num raio (Ra) à volta deste (simplificando).

Após a primeira filtragem, será necessário remover os vértices que não pertencem ao mesmo componente fortemente conexo do grafo onde a padaria (vértice inicial) se encontra. Este segundo fator de eliminação resulta da natureza circular do trajeto, dado que as carrinhas regressam à sua origem, após a finalização das entregas.

Assim, é possível aumentar significativamente a eficiência temporal dos algoritmos que serão aplicados de seguida, garantindo que o grafo não terá vértices nem arestas que não podem ser percorridos pelas carrinhas.

Tratamento dos Clientes

Inicialmente, a sequência de clientes C deve ser percorrida à procura de clientes cuja morada não pertença ao grafo G. Nos casos em que isto acontece, o respetivo cliente deve ser removido, dada a impossibilidade de efetuar a entrega. Durante esta iteração, será calculado o intervalo de tempo segundo o qual a empresa deverá efetuar a entrega para cada cliente (hora mínima e máxima).

Tratamento dos Percursos

Foi analisada a opção de pré-calcular o caminho mais curto entre todos os pares de vértices do grafo processado, utilizando o algoritmo de **Dijkstra** para cada vértice ou o algoritmo de **Floyd-Warshall**.

A utilização de um destes métodos permitiria aumentar a eficiência do restante programa, já que não seria necessário calcular o peso mínimo entre os vários pontos em cada novo percurso. No entanto, esta alternativa provou-se demasiado exigente a nível computacional, visto que calcula a distância entre todos os pares de vértices e não apenas os necessários.

Por esta razão, foi decidida a não utilização desta técnica.

Identificação dos problemas encontrados

Na primeira fase, com uma única carrinha e sem restringir a ordem de entrega pela hora das encomendas, o problema inerente é o de passar por todos os pontos de interesse e regressar à partida, gastando o menor tempo possível, assemelhando-se vivamente ao problema *NP-hard* chamado ***Travelling Salesman Problem (TSP).***

Numa segunda fase, adiciona-se a restrição da hora de entrega preferível pelos clientes, pelo que terão de ser concebidos novos algoritmos que consigam ter este detalhe em atenção.

Passando para uma frota de carrinhas, na última fase do projeto, o algoritmo deve conseguir decidir o número de carrinhas a usar e a alocação ótima delas aos clientes, de modo a minimizar a função objetivo. Assim sendo, o problema tem parecenças com o ***Vehicle Routing Problem***, uma generalização do TSP, que tínhamos nos pontos anteriores.

Percurso de duração mínima entre dois pontos

Primeiramente, é preponderante apresentar soluções para o problema base deste projeto, isto é, obter o caminho mais curto (de peso total mínimo) entre dois pontos do grafo.

Para tal, sugere-se a utilização do algoritmo de **Dijkstra**, devido à sua facilidade de implementação e à sua eficiência temporal, com uma complexidade de $\Theta((|V| + |E|)\log(|V|))$. Para além disso, é possível otimizá-lo, fazendo-o parar quando alcança o vértice de destino.

Contudo, este algoritmo ganancioso efetua a sua procura num círculo que se vai expandindo em torno da origem e, consequentemente, se os dois vértices não estiverem próximos, o método pode tornar-se altamente ineficiente devido à quantidade crescente de vértices inúteis a serem explorados.

De modo a melhorar a sua eficiência temporal, será tida em conta a possibilidade de utilizar o algoritmo **Bidirectional Dijkstra**, uma vez que executa o algoritmo previamente descrito nos dois sentidos, ou seja, de s (origem) para t (destino) e de t para s, alternando entre um e outro e

terminando a iteração atual quando se vai processar um vértice que já foi processado na outra direção. Por fim, basta manter a distância do caminho mais curto encontrado até ao momento e verificar se ainda é possível ser melhorado através de outra iteração.

Este último método enunciado processa uma área de $2\pi r^2$, em vez de $4\pi r^2$ na pesquisa unidirecional (ganho ~ 2x), tal como podemos observar nas imagens apresentadas de seguida:

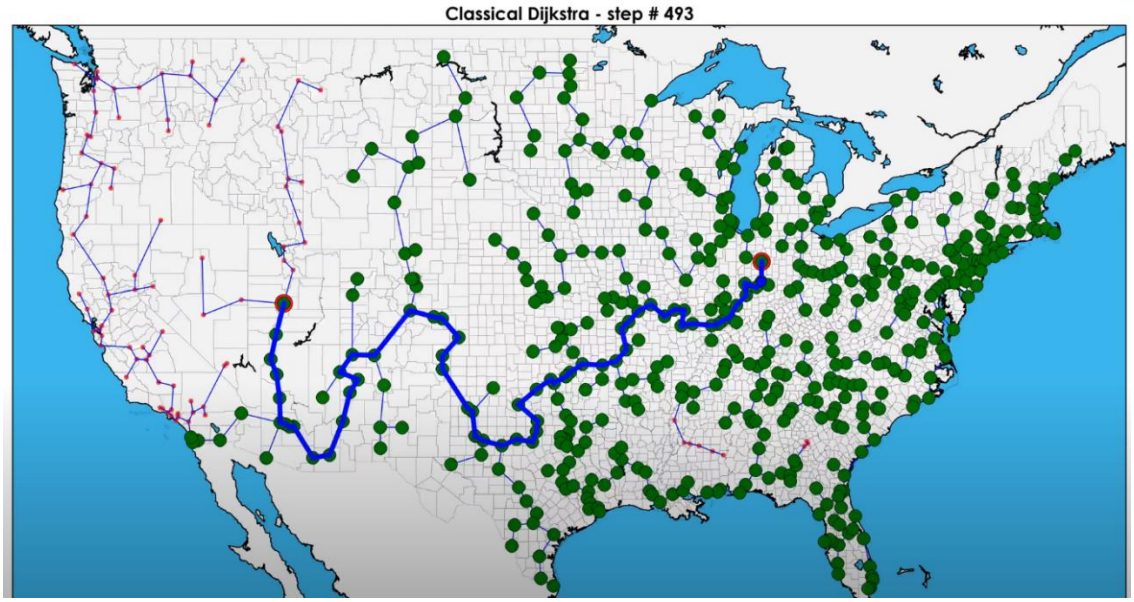

Figura 1 - Pesquisa com Algoritmo de Dijkstra Unidirecional

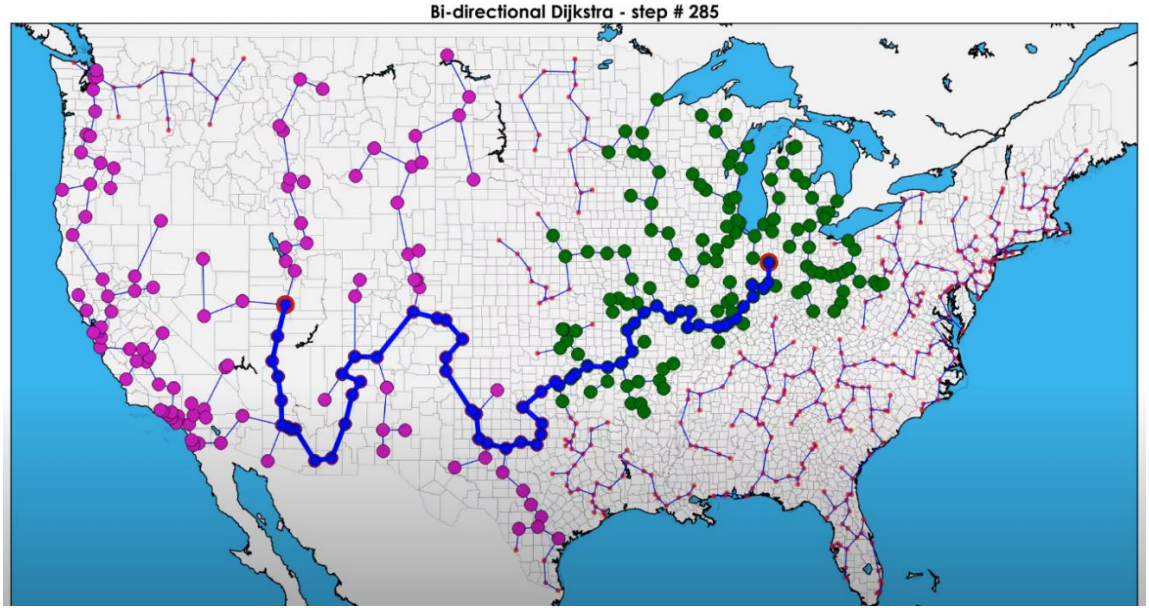

Figura 2 - Pesquisa com algoritmo de Dijkstra Bidirecional

Percurso de duração mínima entre todos os pares de pontos

Como já foi referido na fase de pré-processamento, surgiu a hipótese de pré-calcular os percursos de duração mínima entre todos os pares de vértices.
Embora não tenham sido utilizadas, as duas abordagens foram analisadas:

Execução repetida do Algoritmo de Dijkstra

O algoritmo de Dijkstra, como já foi previamente referido, executa com uma complexidade temporal de $\Theta((|V|+|E|)\log|V|)$. Consequentemente, visto ser necessário executar este algoritmo para todos os vértices, esta abordagem executa com uma complexidade temporal de $\Theta(|V|(|V|+|E|)\log|V|)$ e é recomendada a sua utilização em grafos esparsos ($|V| \sim |E|$), como é normalmente o caso das redes viárias.

Finalmente, a complexidade desta alternativa será $\Theta(|V|^2 \log(|V|))$, sendo mais eficiente do que o algoritmo que será mencionado de seguida.

Algoritmo de Floyd-Warshall

Este algoritmo baseado em programação dinâmica, prova ser superior ao anterior se o grafo for denso ($|E| \sim |V|^2$). Adicionalmente, outro fator a seu favor é simplicidade do seu código e utiliza uma matriz de adjacências com pesos. Por fim, o algoritmo retorna uma matriz de distâncias mínimas e uma matriz de predecessores, que permite determinar os vértices que pertencem ao caminho mais curto entre dois pontos.

A sua complexidade temporal é ($\Theta(|V|^3)$), pelo que, relativamente aos outros métodos possíveis, é preferível em situações de grafos densos.

Algoritmos considerados para a 1ª Fase

Como este problema se assemelha ao TSP, existem vários algoritmos que procuram alcançar soluções para este problema. Assim, existem soluções exatas de elevada complexidade temporal e, por outro lado, soluções mais eficientes, que alcançam resultados aproximados.

Soluções Aproximadas

O **nearest neighbour (NN)** é um algoritmo heurístico, que fornece uma solução razoavelmente eficiente, mas que não é a mais próxima da ótima (em média, possui um desvio de 25%). Este algoritmo, de natureza gananciosa, começa escolhendo um vértice aleatório e,

repetidamente, visitando o vértice mais próximo que ainda não foi visitado, até ter percorrido todos os nós, tendo uma complexidade temporal de $O(n^2)$, no pior caso.

Observe-se que, neste caso de estudo, o vértice inicial será sempre a padaria, o que remove a aleatoriedade associada a este passo.

```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E} (∀ v ∈ V)
S ∈ V

Function NEAREST_NEIGHBOUR(G, S)
    for v ∈ V
        visited(v) = false
    v = S
    numVisited = 0

    while true
        visited(v) = true
        OUTPUT info(v)
        numVisited++
        if numVisited == size(V)
            Return
        v = GET_NEAREST_UNVISITED_CITY(v)
```

Figura 3 - Pseudocódigo para o algoritmo Nearest Neighbour

O **Bitonic tour (BT)** é um algoritmo baseado em programação dinâmica que calcula a cadeia poligonal fechada, de perímetro mínimo, que inclui todos os vértices de um grafo. Este poderá ser aplicado nos vértices de interesse, resolvendo o problema desta fase. As implementações deste algoritmo também têm, usualmente, complexidade temporal de $O(n^2)$, embora sejam conhecidas algumas com $O(n \log^2 n)$.

Soluções Exatas

O algoritmo **naïve** (*brute force*) consistiria em calcular todas as permutações entre vértices do grafo, escolhendo a solução que possui o caminho mais curto. No entanto, possui uma complexidade de O (n!).

O algoritmo **Held-Karp** é outra solução possível, baseada em programação dinâmica, com complexidade O ($n^2 2^n$).

Desta forma, se for pretendida uma solução ótima para este problema, será tido em conta o número de vértices (N) a analisar no caso em questão, dado que para $N < 8$ a técnica de *brute force* apresentará um tempo menor de execução, relativamente ao segundo método apresentado. Consequentemente, para $N \geq 8$, o Held-Karp revela-se superior.

Integração da hora de entrega do pão (2ª Fase)

Como se pretende minimizar o tempo de atraso das entregas para além do tempo de viagem total, foi necessário encontrar novos algoritmos que tomam isto em consideração. Devido a caminhos mais curtos e horas de entrega diferentes, é agora possível passar pelo vértice de um cliente mais do que uma vez (embora se faça uma única entrega).

Em relação a uma solução exata, poderia ser usada **força bruta**, tal como para a 1ª fase, calculando os tempos de viagem e atraso para todas as ordens possíveis de clientes. Estes tempos poderiam ser calculados previamente (para tentar melhorar ligeiramente o tempo de execução), usando, para tal, um algoritmo de cálculo do percurso de duração mínima entre todos os pontos, que foram expostos previamente. Mesmo assim, este algoritmo teria uma complexidade temporal fatorial, o que não é viável.

A primeira possibilidade de solução heurística, que se torna apeladora pela sua simplicidade de implementação, é a de **ordenar** todos as encomendas pelas suas horas de preferência e, por essa ordem, descobrir, dois a dois, o tempo mínimo de viagem entre as casas dos clientes, usando um algoritmo de cálculo do percurso mínimo entre dois pontos, como o de **Dijsktra**, o que nos daria uma solução aproximada. No entanto, devido à sua natureza gananciosa, que só tenta minimizar o tempo de atraso da próxima entrega, o erro (comparando à solução ótima) deste método pode crescer significativamente de acordo com o caso em questão (por exemplo, imaginemos um caso onde a casa de um cliente que pediu uma encomenda às 9:10 fica no trajeto da casa de um cliente que pediu para as 9:00. Se forem já 9:05, é provavelmente melhor parar na casa do primeiro cliente antes do segundo em vez de repetir o caminho de volta, embora a hora dele seja mais tarde).

Para combater este problema, foram consideradas otimizações a aplicar ao algoritmo. Uma delas foi a seguinte condição:

- Se, a caminho da casa de um cliente, passarmos pela casa de um outro cliente cuja hora de preferência é mais tardia, mas a hora atual está dentro da margem de entrega, então realiza-se essa entrega.

Foi também estudada a possibilidade de escalar esta condição a vértices próximos dos que estamos a percorrer, tal como outras possíveis otimizações.

Este algoritmo tem a mesma complexidade temporal que o algoritmo usado para calcular a distância entre os vértices, visto ser a ação mais dispendiosa.

```
G = (V, E) // Graph
Adj(v) = {w | (v, w) ∈ E} (∀ v ∈ V)
Deliveries = (H, v ∈ V) // List

Function GET_PATH_WITH_DIJSKTRA(G, C, S)
    rootTotalTime = 0
    delayTotalTime = 0
    // The order might be changed inside our Dijsktra based algorithm
    SORT_BY_HOUR(Deliveries)
    D1 = S
    while Deliveries ≠ ∅
        D2 = Deliveries.first
        // Calculates the shortest path between two vertexes
        // Updates the total times and
        // Deletes the deliveries completed from the list
        MODIFIED_DIJSKTRA(D1, D2, rootTotalTime, delayTotalTime, Deliveries)
        D1 = D2
    OUTPUT rootTotalTime, delayTotalTime
```

Figura 4 - Pseudocódigo do algoritmo para visita ordenada a clientes, usando Dijsktra

Outra técnica que foi considerada, embora mais complexa e com uma implementação mais trabalhosa, foi uma aplicação de *General Variable Neighborhood Search (GVNS)*, que se divide em duas fases: uma fase construtiva, que resulta numa solução possível, mas não necessariamente perto da ótima e uma fase de otimização, que pega na solução anterior e tenta melhorá-la o máximo que conseguir.

Para a fase construtiva, poderíamos reutilizar uma solução da 1ª fase ou, para tentar diminuir a complexidade temporal, gerar uma solução aleatória e melhorá-la até termos uma solução prática (de forma semelhante à segunda fase). Em relação à fase de otimização, a ideia será fazer mudanças

aleatórias ligeiras, calcular o resultado da nova solução e comparando-a à antiga e atualizando-a, caso tenhamos obtido um resultado melhor.

De seguida, é possível analisar pseudocódigo que tenta explicitar este algoritmo, que terá de ser adaptado para o projeto, retirado do artigo *"A General VNS heuristic for the traveling salesman problem with time windows"* (ver bibliografia):

Algorithm 2: VNS - Constructive phase

Output: X

```
repeat
    level ← 1 ;
    X ← RandomSolution() ;
    X ← Local1Shift(X) ;
    while X is infeasible and level < levelMax do
        X′ ← Perturbation(X, level) ;
        X′ ← Local1Shift(X′) ;
        X ← better(X, X′) ;
        if X is equal to X′ then
            level ← 1 ;
        else
            level + + ;
        end
    end
until X is feasible;
```

Figura 5 - Fase construtiva

Algorithm 3: GVNS

Input: X, levelMax

Output: X

```
level ← 1 ;
X ← VND(X) ;
while level <= levelMax do
    X′ ← Perturbation(X, level) ;
    X′ ← VND(X′) ;
    X ← better(X, X′) ;
    if X is equal to X′ then
        level ← 1 ;
    else
        level + + ;
    end
end
```

Figura 6 - Fase de otimização, onde VND e Perturbation são funções para obter novas soluções

Também foi estudada uma possibilidade de modificar os algoritmos implementados na 1ª fase, alterando a forma como são tratados os pesos das arestas, de forma a calcular, dinamicamente, novos pesos tendo em conta a hora atual (dando mais importância ao tempo de atraso do que ao tempo de viagem). No entanto, este método pareceu rebuscado e iria ser difícil ter em conta a possibilidade de

repetir visitas a vértices passados, visto que os métodos anteriores buscam passar apenas uma vez por cada cliente.

Utilização de múltiplas carrinhas com capacidade limitada (3ª Fase)

Numa última fase, integrou-se múltiplas carrinhas ao problema resolvido na 2ª fase, sendo que os clientes também pedem um certo número de pães. Assim sendo, será necessário alocar as várias carrinhas aos seus clientes, de forma a minimizar o número de veículos usados, além dos atributos que já se tentavam minimizar anteriormente.

A abordagem preferida foi resolver o problema de alocação das carrinhas independentemente, usando, após este primeiro passo, um dos algoritmos implementados na 2ª fase para cada um dos veículos utilizados (como já sabemos que a carrinha tem capacidade para todos os clientes alocados, caímos num problema idêntico ao anterior).

Assim sendo, encontramo-nos num problema semelhante ao ***Knapsack problem***, com a diferença de ter várias "mochilas". Como o nosso objetivo primário é minimizar o número de carrinhas usadas, usamos uma abordagem gananciosa e preenchemos primeiro as carrinhas com maior capacidade, até acabarem as encomendas.

De forma a otimizar o algoritmo, e como é provável que a última carrinha fique com algum espaço livre, percorremos as alocações das outras carrinhas à procura de clientes cuja entrega seja um transtorno menor na rota do último veículo, isto é, se a carrinha tiver um número de entregas muito superior à última ou se há casas que ficam mais perto das moradas dos clientes alocados ao último veículo *(edge cases)*. Além disso, se surgirem clientes cuja encomenda era demasiado grande para ser realizada por uma carrinha, é possível a encomenda deste ser separada por várias.

```
// V- List of vans
// C- List of clients
Function ALLOCATE_VANS(V, C)
    SORT_BY_CAPACITY(V)
    for van ∈ V
        // Allocates the better clients to a given van
        KNAPSACK(van, C)
        if C.empty
            Break
    // Considers the last van with allocated clients
    // And takes some clients from other vans,
    // in the beneficial cases
    OPTIMIZE_LAST_USED_VAN(V)
```

Figura 7 - Pseudocódigo do algoritmo de alocação de clientes a carrinhas

O problema de ***Knapsack*** pode ser resolvido através de vários métodos, como a programação dinâmica, o *backtracking*, métodos gananciosos, entre outros. No entanto, optamos por utilizar o primeiro, pois é uma forma muito eficiente de ultrapassar este obstáculo.

Este método armazena recursivamente o resultado para que cada vez que um problema é dado, é retornado o resultado ao invés de outro problema (de forma iterativa).

Como o problema do *knapsack* pode ser resolvido em O(nW), onde W é a capacidade máxima da carrinha e n o número de clientes, este algoritmo teria complexidade O(nWC), onde C é o número de carrinhas.

Um segundo algoritmo utilizado na alocação dos clientes às carrinhas trata-se de uma abordagem gananciosa, que, para cada carrinha e a cada iteração, escolhe o cliente com mais valor cuja encomenda ainda é suportada por essa carrinha. O valor destes é calculado em cada iteração e tem em conta:

- Uma constante (qualquer cliente tem valor por si próprio)
- O número de pães na encomenda (maior número de pães implica valor maior)
- Soma das distâncias à padaria e aos outros clientes (maior distância implica valor substancialmente menor). É usada a distância euclidiana pela sua simplicidade e eficiência de cálculo
- Soma do valor absoluto das diferenças (em minutos) da hora de entrega em relação às horas dos outros clientes já alocados (um maior afastamento das horas dos outros clientes implica valor consideravelmente maior). Isto é feito com o objetivo de distribuir, ao máximo, as encomendas ao longo do dia, reduzindo atrasos.

No fim deste processo, também é aplicado um algoritmo se otimização idêntico ao usado no *Knapsack*.

Outro algoritmo que merece uma menção é o *"Vehicle Routing Problem with Time Windows",* que usa MCPSO (*multi-swarm cooperative particle swarm optimizer)* para tentar minimizar custos de deslocação de vários veículos que cobrem um mesmo conjunto de pontos no mapa. No entanto, esta técnica tem uma complexidade elevada e não tem foco no nosso principal objetivo, o de usar o mínimo de carrinhas possíveis, pelo que foi decidido não o implementar.

Análise da conectividade

Tal como já foi referido anteriormente, durante a fase de pré-processamento do grafo, os vértices que não se encontram no componente fortemente conexo da padaria devem ser removidos do grafo.

Consequentemente, torna-se necessário analisar a conectividade do grafo, de forma a excluir os nós com pouca acessibilidade.

Como os grafos analisados podem tanto ser dirigidos como não dirigidos, consideram-se análises para esta duas situações distintas. Apresentam-se de seguida alguns algoritmos para resolver este problema:

Algoritmo de Kosaraju

Aplicado em grafos dirigidos, este algoritmo encontra todos os componentes fortemente conexos de um grafo e consiste em fazer uma pesquisa em profundidade, numerando os vértices em pós-ordem, e fazendo, após a inversão de todas as arestas, uma segunda pesquisa em profundidade, começando pelos vértices de numeração superior. No final desta sequência de passos, cada árvore obtida é um componente fortemente conexo. Contudo, apenas interessa o componente que inclui a origem, ou seja, a padaria.

Este método tem uma complexidade O (|V| + |E|), ou seja, corre em tempo linear.

Algoritmo de Tarjan

O método proposto por Robert Tarjan surge como uma alternativa ao algoritmo de Kosaraju, sendo, portanto, também aplicado a grafos dirigidos. Apesar de também executar em tempo linear, executa apenas uma pesquisa em profundidade, sendo por isso mais eficiente.

Por este motivo, a implementação deste algoritmo é mais vantajosa.

Depth-First Search

Para analisar a conetividade de um grafo não dirigido, basta aplicar DFS a partir do vértice considerado (padaria), marcando os vértices percorridos como visitados. Pela natureza do grafo, todos os vértices a que a pesquisa chegar terão também caminhos de volta para o vértice inicial, criando um componente fortemente conexo.

A DFS visita todos os vértices para os marcar como não visitados e, na pesquisa, percorre todas as arestas de cada vértice encontrado, tendo, portanto, uma complexidade temporal O (|V| + |E|).

Estruturas de dados e Classes utilizadas

Estruturação do Grafo

Grafo (classe *Graph*)

A representação do grafo, baseada na implementação fornecida para as aulas práticas de CAL, é constituída por um vetor de vértices (vector<Vertex*>), tal como um H*ashMap* (unordered_map<int, Vertex*>) , associando o ID do vértice com o próprio. Isto foi feito tendo em conta que as duas operações mais frequentes para com os vértices do grafo, iteração e acesso único, são facilitadas e otimizadas nestas duas estruturas de dados, respetivamente.

Além disso, o grafo contém vários algoritmos implementados que são usados na resolução dos problemas em questão, entre eles, *Dijkstra* (várias versões, sendo mais versátil em diversas situações), algoritmos de análise de conetividade e pesquisa em profundidade.

O grafo também apresenta algumas funções utilitárias, tais como adicionar/remover arestas/vértices, imprimir o grafo e obter o caminho de arestas calculado num outro algoritmo.

É importante notar que este grafo tem a possibilidade de ser tanto dirigido como não dirigido, escolhido dinamicamente pelo utilizador (por *input* ou por ficheiro de texto).

Vértice (classe *Vertex)*

Tendo em conta a implementação do grafo, estes vértices têm um ID único, igual ao usado no mapa anterior, uma posição, um vetor de arestas adjacentes a este (*outgoing*), tal como outros atributos criados especificamente para os algoritmos aplicados nesta estrutura, como *Dijkstra* e DFS.

Aresta (class *Edge*)

As arestas possuem informação os vértices de origem e de destino (Vertex*), um ID único e um peso, que, neste caso, representa o tempo, em minutos, que a carrinha demora a passar nessa aresta. Contém ainda alguns métodos que facilitam o seu uso e a sua mostragem ao utilizador.

Modelação da Padaria

Padaria (classe *Bakery*)

De modo a agrupar e manipular toda a informação sobre as entregas do serviço, foi criada uma classe que contém:

- Um vetor de clientes (vector<Client*>), cada um com a respetiva informação sobre o seu pedido.
- Um vetor de carrinhas (vector<Van>), cada uma com a respetiva informação (explicitada em baixo).
- Um vértice de início, representando a padaria de onde todas as carrinhas terão de partir e voltar.
- Parâmetros sobre o raio máximo de entrega, os tempos máximos de atraso e adiantamento, e o próprio grafo onde se incide o serviço.
- Métodos para a resolução das três diferentes fases do problema, parametrizadas com as escolhas dos algoritmos a utilizar, de acordo com a preferência do utilizador em relação à melhor otimização ou à rapidez dos algoritmos.
- Vários algoritmos essenciais para resolver o problema, que usam, como base, o grafo previamente carregado dum ficheiro de texto.

Cliente (classe *Client*)

Tal como explicitado na formalização do problema, os clientes são constituídos por um ID próprio, um nome, um vértice contido no grafo em questão, representando a sua morada, a quantidade de pães que encomendou e a hora marcada.

Estes também contêm atributos e métodos auxiliares aos algoritmos que usam esta classe (e.g. *flag* a informar se já foi alocado) e um tempo real de entrega, a ser definido pelos mesmos.

Carrinha (classe *Van*)

As carrinhas têm informação sobre a sua capacidade máxima, o tempo de entrega do seu condutor, um vetor de clientes a quem vai realizar as encomendas (vector<Client*>) e um vetor de arestas (vector<Edge>) a ser preenchido pelos algoritmos que incidem nesta classe.

Além disso, também contêm vários atributos a serem definidos, tais como a quantidade de pão reservada e entregue e os tempos totais de viagem e atraso. Conta também com alguns métodos úteis, como o de ordenação dos seus clientes pela hora de entrega e o de remover e retornar o cliente que mais se distancia de todos os outros (tendo em conta uma quantidade máxima de pães aceite), usado em algoritmos de otimização.

Interface (class *Interface*)

Foi criada uma classe cujo objetivo é interagir com o utilizador, permitindo-o escolher os dados do problema (por ficheiro texto ou *input* direto) e quais os algoritmos a usar. Depois da escolha, a interface invoca a classe *Bakery* para obter o resultado.

Após obtida a solução para o problema, a interface permite mostrar a solução na saída padrão,

gravá-la num ficheiro de texto e até apresentá-la numa interface gráfica (*GraphViewer*), mostrando ao utilizador o caminho percorrido num mapa real do grafo escolhido.

Classes Utilitárias

Posição (classe *Position*)

Visto que temos várias situações onde são usadas coordenadas (vértices, clientes, *GraphViewer*, etc.), foi criada uma classe que facilita o uso destas, agrupando-as e implementando métodos inerentemente relacionados, tais como a distância euclidiana e a comparação de posições.

Tempo (classe *Time*)

Tendo em conta a natureza do problema, foi considerada pertinente a criação de uma classe que facilitasse o armazenamento e manipulação de horas (e.g. de entrega) e durações (e.g. da viagem das carrinhas ou do atraso das entregas).

Assim sendo, esta estrutura tem a possibilidade de ser criada com uma hora e minutos definidos ou simplesmente com o número de minutos. Também apresenta métodos utilizados por vários algoritmos, tais como a conversão em minutos (para facilitar cálculos) e a comparação entre "tempos".

Construtor de Grafos (classe *GraphBuilder*)

Com vista a facilitar a criação de grafos e a separar esta responsabilidade da *Bakery*, foi criada uma classe cujo encargo é criar um grafo padrão (*hard coded*) ou carregá-lo de um ficheiro de texto.

MutablePriorityQueue

Retirada das aulas práticas de CAL, esta classe foi utilizada num dos algoritmos principais da solução para o problema, permitindo-o atualizar dinamicamente o item no início da fila prioritária, sem ter que repetir ou iterar elementos já lá contidos.

Algoritmos Implementados

Destacam-se nesta secção os principais algoritmos concessionados para o projeto. Estes serão separados pela fase do projeto em que são usados, começando pelos comuns a todas elas. Alguns dos algoritmos não serão analisados a fundo, visto serem já vivamente conhecidos e implementados, nomeadamente, DFS, Tarjan e Dijkstra. Para estes, será explicitada a pertinência no projeto e a sua implementação.

Nota: Em todas as análises empíricas, apenas uma variável é mudada em cada ponto calculado (o que está explicitado no eixo das abcissas do gráfico). Esta foi feita no seguinte computador:

- Modelo: Lenovo Ideapad 5 15ARE05
- CPU: AMD Ryzen 5 4500U
- Memória RAM: 8GB
- Placa Gráfica: AMD Radeon Graphics

Análise da Conetividade - Depth-First Search

Pseudocódigo

```
// v: Start Vertex
function DFS(v)
    v.visited = True
    For edge ∈ v.adj
        if (!edge.dest.visited)
            DFS(edge.dest)
```

Figura 8 - Pseudocódigo do Algoritmo de DFS

Análise da Complexidade

Este algoritmo começa por visitar todos os vértices do grafo, marcando-os como não visitados. Depois, a partir do vértice inicial parametrizado, visita, recursivamente, todas as arestas que encontra nos vértices por onde passa, percorrendo e marcando o componente fortemente conexo relevante. Assim sendo, visto que o número de arestas visitadas é no máximo |E|, o algoritmo terá uma complexidade temporal de O (|V| + |E|), tal como foi dito anteriormente, e uma espacial de O(|V|), tendo em conta o atributo adicional do vértice, necessário para saber se já foi ou não visitado.

Análise da Conetividade – Algoritmo de Tarjan's

Pseudocódigo

```
// vertexU: Starting Vertex
// st: Stack with vertices on current DFS
function tarjan(vertexU, st)
    for v ∈ vertexSet
        v.disc = -1
        v.low = -1
        v.stackMember = False
        v.visited = False

    tarjanUtil(vertexU, st)

function tarjanUtil(vertexU, st)
    static discoveryOrder ← 0

    vertexU.disc = vertexU.low ← ++discoveryOrder
    st.push(vertexU)
    vertexU.stackMember ← True

    for edge ∈ vertexU.adj
        adjVertex ← edge.dest
        if adjVertex.disc == -1
            tarjan(adjVertex.id, st)
            // Check if the subtree rooted with 'adjVertex'
            // has a connection to an ancestor of 'vertexU'
            vertexU.low ← min(vertexU.low, adjVertex.low)

        // Update low value of 'vertexU' only if 'adjVertex'
        // is still in stack (back edge)
        else if adjVertex.stackMember
            vertexU.low ← min(vertexU.low, adjVertex.disc)

    // Head node found
    if vertexU.low == vertexU.disc
        while st.top() != vertexU
            currVertex ← st.top()
            currVertex.stackMember ← False
            if vertexU.disc == 1    // Traversing the starting vertex tree
                currVertex.visited ← True
            st.pop()
            if currVertex == vertexU
                break
```

Figura 9 - Pseudocódigo Tarjan

Análise da Complexidade

Este algoritmo começa por visitar todos os vértices do grafo, atualizando alguns atributos que irão ser utilizados pelo algoritmo. De seguida, invoca a função utilitária recursiva tarjanUtil, que será responsável por marcar os vértices do componente fortemente conexo como visitados. Os passos envolvidos são:

1. Pesquisa em profundidade sobre os nós
2. Dois valores disc[u] e low[u] são guardados para cada vértice.
    - Disc[u] – Ordem de descoberta do vértice u
    - Low[u] – Guarda o valor de 'disc' mais baixo que é atingível a partir do vértice u, que não faz parte de outro componente fortemente conexo
3. À medida que os nós são explorados, são inseridos numa stack
4. Os nós que ainda não foram explorados são explorados e atualizam o seu disc[u] e low[u] adequadamente
5. Um nó encontrado onde low[u] = disc[u] é o primeiro nó explorado no seu componente fortemente conexo e, consequentemente, todos os nós acima dele na stack serão removidos e identificados como pertencentes ao seu *SCC* (strongly connected component).
    - Especificamente, no âmbito deste projeto, para obter o componente fortemente conexo da padaria, bastou marcar os vértices pertencentes à stack, quando o nó da padaria foi explorado, como visitados.

Como tal, sabendo que o algoritmo é baseado em pesquisas em profundidade, conclui-se que cada nó é visitado no máximo uma vez e, para cada nó é visitada a sua lista de arestas adjacentes. Portanto, a complexidade temporal deste algoritmo é O(|V| + |E|).

Relativamente ao espaço utilizado pelo algoritmo de Tarjan, verifa-se que a complexidade espacial do mesmo é linear, ou seja, O(|V|), já que é necessário armazenar para cada vértice 4 atributos (low[u], disc[u], stackMember[u] e visited[u]) e, para além disso, recorre-se a uma stack com tamanho máximo |V|.

Algoritmo de Dijkstra e variações

Pseudocódigo

```
// start: Start Vertex
function DijkstraShortestPath(start)
    for v ∈ vertexSet
        v.dist ← INF
        v.path ← null

    start.dist ← 0
    Q ← ∅
    ENQUEUE(Q, start)

    while Q != ∅
        v ← DEQUEUE(Q)
        for e ∈ v.adj
            oldDist ← e.dest.dist
            if (Relax(v, e))
                if (oldDist = INF) DEQUEUE(Q, e.dest)
                else DECREASE_KEY(Q, e.dest)
```

Figura 10 - Pseudocódigo Dikstra

```
// v: vertex
// e: edge
function Relax(v, e)
    w ← e.dest
    if (v.dist + e.weight < w.dist)
        w.dist ← v.dist + e.weight
        w.path ← v
        w.pathEdge ← e
        Return True
    Return False
```

Figura 11 - Pseudocódigo Função Relax

Análise da Complexidade

O algoritmo começa por marcar a distância de todos os vértices como infinita, à exceção do inicial, que é marcado com 0. De seguida, usando uma fila de prioridade mutável, visita,

recursivamente, todas as arestas do vértice atual e coloca os vértices de destinos na fila, visitando, de seguida, sempre o vértice com menor distância, até os visitar a todos.

Assim sendo, a complexidade do algoritmo será O ((|V| + |E|) log(|V|)). O log é derivado do tempo de inserção/atualização na fila de prioridade, enquanto que |V| e |E| derivam da iteração nos vértices e arestas do grafo.

Por sua vez, a complexidade espacial será O (|V|), visto serem necessários atributos adicionais no vértice (distância e caminho), tal como uma fila de prioridade que irá guardar os vértices a serem visitados (todos dependem de |V|).

Variações do algoritmo

Dijkstra com vértice de destino: Parametrizando um vértice de destino, o algoritmo é capaz de terminar a sua execução mais rapidamente, visto não ter que visitar todos os restantes vértices depois de encontrar o de destino.

Esta variação é útil nos casos em que apenas importa chegar a uma posição específica do grafo, como por exemplo, a casa do cliente seguinte. Como é usada uma fila de prioridade mutável, também é obtido o caminho mais curto entre os dois pontos.

Dijkstra com um conjunto de possíveis destinos: Em certos algoritmos, não nos é importante encontrar o caminho para um cliente específico mas para um dos clientes restantes (***nearest neighbour***), pelo que esta variação de Dijkstra se torna muito relevante.

Parando a execução no encontro de um dos destinos possíveis, satisfazemos perfeitamente o problema num tempo consideravelmente menor do que com o algoritmo clássico.

Dijkstra bidirecional: Com a mesma intenção do Dijkstra com vértice de destino, esta variação revela-se ainda mais eficaz do que o anterior (tal como explicado anteriormente).

Ao procurar, paralelamente, o caminho de ambos os vértices (início e destino), é possível convergir para uma solução mais rapidamente.

Pré-processamento:

Pseudocódigo

```
// start: start vertex
function FilterByRadius(start, radius)
    for v ∈ vertexSet
        if EuclideanDistance(start, v) > radius
            vertexSet.remove(v)

function FilterBySCC(start)
    AnalyzeConnectivity(start) // Either Tarjan or DFS
    for v ∈ vertexSet
        if !v.visited
            vertexSet.remove(v)

// G: Graph
function FilterClients()
    for client ∈ clientSet
        if !G.hasVertex(client.vertex)
            clientSet.remove(client)
```

Figura 12 - Pseudocódigo Pré-processamento

Análise da Complexidade

O pré-processamento dos dados pode ser resumido pelos seguintes passos:

- Remover os vértices do grafo, e respetivas arestas, cuja distância seja superior da permitida pelos dados de entrada (raio de entrega). Para tal, precisamos de percorrer todos os vértices, verificando se devem ser removidos, e, se tal acontecer, removê-lo. Por fim, teremos que remover as arestas que começavam em algum dos vértices removidos.
  Visto que a operação de remoção de um vértice é O(|V|) e a de remover as arestas é O(|E|), a complexidade temporal geral deste passo é O ($|V|^2 + |E|$). A complexidade espacial é O (1)

  Nota: As arestas no conjunto de arestas adjacentes em cada vértice nunca são repetidas, pelo que a iteração e remoção destas podem ser consideradas uma operação em tempo linear, em

função do número de arestas, visto que nunca vão ser feitas mais que |E| iterações.

- Análise da conetividade e posterior remoção de vértices fora do SCC. Seja feito com DFS (grafo não dirigido) ou com Tarjan (grafo dirigido), este passo tem uma complexidade de O (|V| + |E|).
  A remoção dos vértices não visitados pelo algoritmo de conetividade é uma situação idêntica à do passo anterior, pelo que a complexidade deste passo vai ser dita por esta, sendo O ($|V|^2$ + |E|). A complexidade espacial é O (|V|) em ambos os casos.

- Filtragem dos clientes cuja morada não corresponde a um vértice do grafo. Visto usarmos um *HashMap* para procurar vértices (*lookup* em O (1)), a complexidade temporal depende apenas da iteração e remoção de clientes. Assim sendo, como é usado um vetor, a complexidade será O($|C|^2$), visto que a operação de remoção ocorre em tempo linear. Por sua vez, a espacial é simplesmente O (1).

Concluindo, tendo em conta os fatores dominantes e a sequência dos passos, o algoritmo de pré-processamento possui, na sua totalidade, complexidade temporal O ($|V|^2$ + |E| + $|C|^2$) e espacial O(|V|).

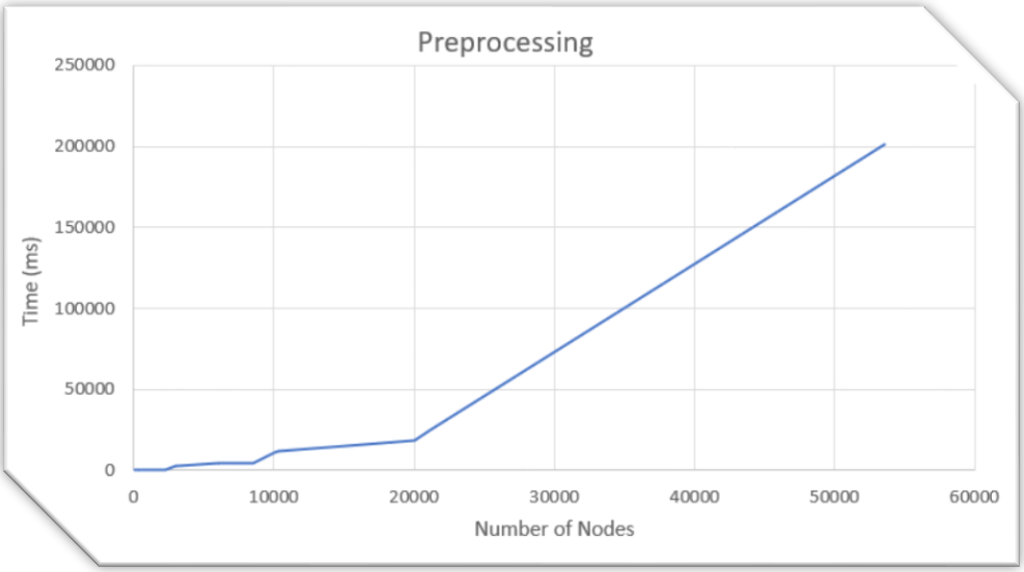

Figura 13 - Análise Empírica Pré-processamento

1ª fase- Carrinha única com capacidade ilimitada e em consideração de horas de entrega:

O algoritmo utilizado para a 1ª fase do projeto é uma implementação do *Nearest Neighbour*, com o auxílio de duas variações do algoritmo de Dijkstra:

Pseudocódigo

```
// G: Graph
function NearestNeighbour(van)
    for v ∈ vertexSet
        v.visited ← False

    v ← G.startingVertex
    numVisited ← 0

    while numVisited < van.numClients
        closestClient ← DijkstraClosestClient(v, clientSet)
        AddPathToVan(van, v, closestClient.vertex)
        van.makeDelivery(closestClient)

        v ← closestClient.vertex
        v.visited ← True
        numVisited++

    returningTime ← BidirectionalDijkstra(v, G.startingVertex)
    AddPathToVan(van, v, closestClient.vertex)
    van.addTime(returningTime)
```

Figura 14 - Pseudocódigo Nearest Neighbour

Análise da Complexidade

O algoritmo inicia-se marcando todos os vértices do grafo como não visitados, o que é feito em O(|V|).

De seguida, começa-se um conjunto de iterações, sendo que cada uma conta com os seguintes passos:

- Cópia para um vetor auxiliar de todos clientes que ainda não foram visitados, tendo complexidade temporal e espacial O (|C|).
- Obtenção do cliente mais próximo do vértice atual (começando na padaria), usando o algoritmo de Dijkstra com conjunto de possíveis destinos, tendo complexidade temporal O ((|V| + |E|) log(|V|)) e espacial O (|V|).
- Adiciona-se à carrinha as arestas do caminho encontrado no passo anterior. Isto é feito percorrendo o caminho do vértice de destino até ao inicial, colocando a aresta utilizada num vetor, e, posteriormente, inserir pela ordem correta no conjunto de arestas da carrinha. Assim sendo, a complexidade temporal é O (|V| + |E|) e a espacial é O (|E|).

Visto que este processo acontece |C| vezes, esta parte do algoritmo tem complexidade temporal O ($|C|^2$ + |C| (|V| + |E|) log(|V|)), tendo em conta os fatores dominantes.

Por fim, é necessário encontrar o caminho mais curto de volta à padaria, a partir do último cliente visitado. Para este cálculo, separamos em 2 opções:

1. Grafo não direcionado – No caso do grafo ser não direcionado, recorremos ao algoritmo de Dijkstra bidirecional para encontrar este caminho de retorno, uma vez que fornece um *speed up* de, aproximadamente, 2 vezes.
2. Grafo direcionado – Caso o grafo seja direcionado, calculamos o caminho de retorno através do algoritmo de Dijkstra entre 2 pontos.

De facto, é necessário fazer esta separação, já que, apesar do Dijkstra bidirecional ser mais eficiente, não é garantido que funcione em grafos dirigidos. Isto porque o algoritmo faz uma pesquisa nas 2 direções, ou seja, da origem para o destino e do destino para a origem. Por sua vez, a pesquisa do destino para a origem é efetuada por arestas no sentido contrário ao pretendido e, naturalmente, não é garantido que exista uma aresta semelhante no sentido contrário no caso dos grafos dirigidos. Evidentemente, a execução temporal está na ordem de O ((|V| + |E|) log(|V|)).

Assim sendo, o algoritmo usado para a 1ª fase tem uma complexidade temporal O ($|C|^2$ + |C| (|V| + |E|) log(|V|)), sendo o ciclo o fator dominante. Por sua vez, a complexidade espacial é O (|C| + |V| + |E|).

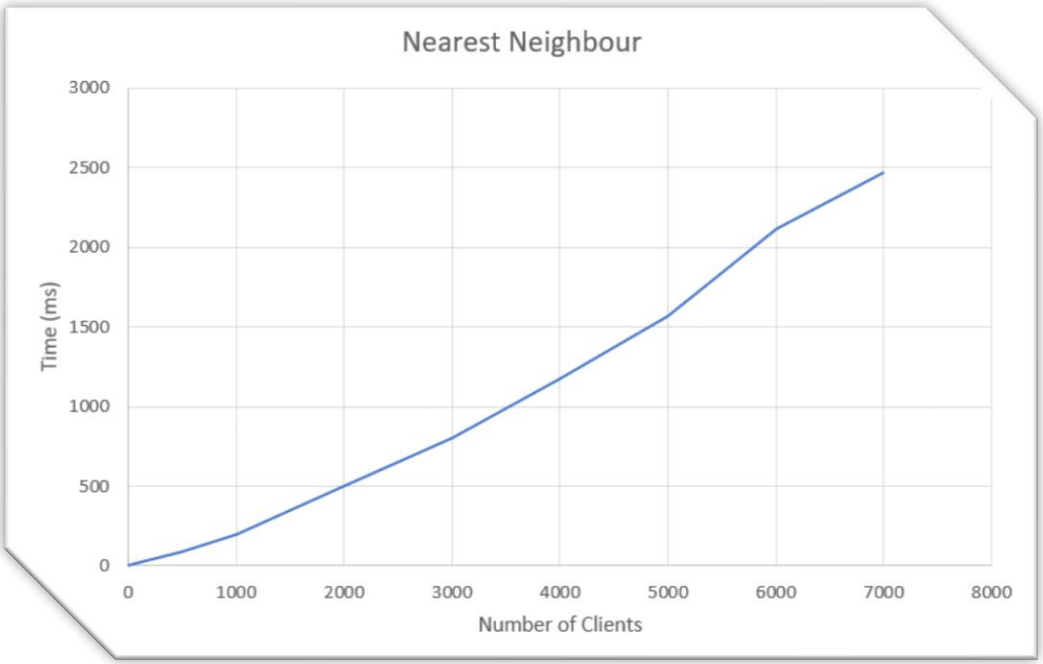

Figura 15 - Análise Empírica Nearest Neighbour

2ª fase- Carrinha única com capacidade ilimitada e com consideração de horas de entrega:

O algoritmo usado para esta fase do projeto tem uma natureza gananciosa, ordenando os clientes pela sua hora de entrega e, posteriormente, calculando o caminho mais curto entre eles usando uma variante do algoritmo do Dijkstra, tal como foi explicado na secção da perspetiva da solução. A otimização falada anteriormente, onde a ordem de entrega dos clientes pode ser trocada, foi implementada independentemente do algoritmo de Dijkstra.

Novamente, é necessário realçar que o algoritmo de Dijkstra utilizado é dependente do tipo de grafo com que se está a trabalhar. Tal como já foi explicado na secção da primeira fase, utiliza-se Dijkstra Bidirecional em grafos não dirigidos e Dijkstra em grafos dirigidos.

Pseudocódigo

```
function GreedyWithDijkstra(van)
    van.SortClientsByTime()
    v1 ← startVertex
    v2 ← ∅
    for client ∈ van.clients
        if nextClient.deliveryTime < currentTime and
        Euclidean(client, v1) > 10 * Euclidean(nextClient, v1)
            swap(client, nextClient)

        v2 ← client.Vertex
        Dijkstra(v1, v2)
        van.makeDelivery(client) // Sets all attributes needed for the result
        v1 ← v2

    Dijkstra(v1, startVertex)
    van.updateTime()
```

Figura 16 - Pseudocódigo Algoritmo Ganancioso

Análise da Complexidade

O algoritmo começa com uma simples ordenação do vetor de clientes, o que pode ser feito em O (C log|C|).

Depois, é executado o seguinte processo para cada cliente:

- Chamada do Dijkstra bidirecional para calcular o caminho entre o vértice atual (começando na padaria) e a morada do próximo cliente, o que é executado em O ((|V| + |E|) log(|V|)).
- De forma idêntica ao *Nearest Neighbour*, adiciona-se à carrinha as arestas do caminho encontrado no passo anterior. Com já foi analisado antes, a complexidade temporal deste passo é O (|V| + |E|) e a espacial é O (|E|).

No seu total, este processo tem uma complexidade temporal de O (|C| (|V| + |E|) log(|V|)).

Por fim, é mais uma vez usado o Dijkstra bidirecional para calcular o caminho de volta à padaria, com O ((|V| + |E|) log(|V|)).

Concluindo, a complexidade temporal deste algoritmo é O (|C| (|V| + |E|) log(|V|)) e a espacial é O (|V| + |E|).

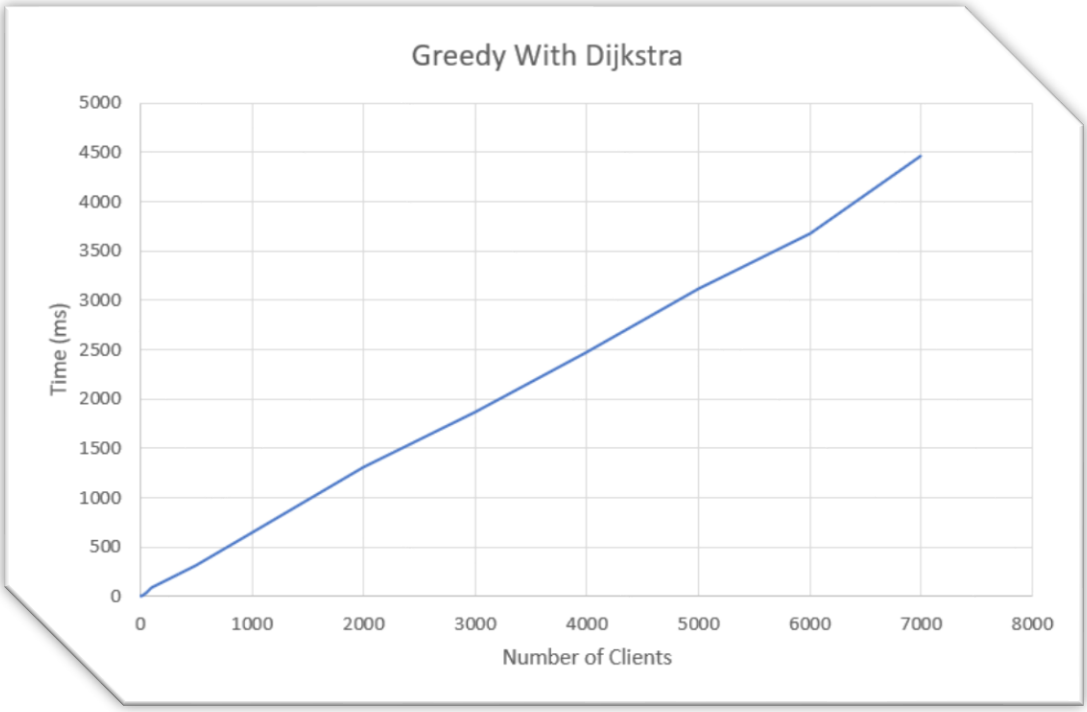

Figura 17 - Análise Empírica Algoritmo Ganancioso

3ª fase- Múltiplas carrinhas com capacidade limitada e com consideração de horas de entrega:

A 3ª fase do projeto utiliza dois algoritmos distintos na sua resolução: um para a alocação dos clientes às carrinhas e outro para efetivamente calcular o trajeto de cada carrinha. Para o segundo, é usado o mesmo algoritmo que na 2ª fase, mas o primeiro tem várias opções, que vão ser explicitadas em baixo.

Pseudocódigo - *Knapsack Allocation*

```
// values: Array
function Knapsack(van, values)
    clients ← Ø // Array
    for client ∈ clientSet
        if !client.isAllocated
            PUSH(cliens, client)

    table ← Ø // 2D Array

    for i ← 0; i < clients.size; ++i
        for w ← 0; w < van.capacity; ++w
            if i=0 OR w=0
                table[i][w] ← 0
            else if clients[i-1].breadQuant <= w
                table[i][w] ← MAX(values[i-1] +
                                  table[i-1][w-clients[i-1].breadQuant],
                                  table[i-1][w])
            else
                table[i][w] ← table[i - 1][w]

    w ← van.capacity
    i ← clients.size
    removed ← 0
    while i>0
        if (table[i][w]-table[i - 1][w - clients[i - 1].breadQuant]=values[i-1])
            van.addClient(clients[i-1])
            removed++
            w ← w - clients[i-1].breadQuant
        i--
    Return removed
```

Figura 18 - Pseudocódigo Knapsack

Pseudocódigo – Alocação Gananciosa

```
function GreedyAllocation(van)
    count ← 0
    capacity ← van.capacity
    while True
        highest ← -INF
        chosen ← NULL
        for client ∈ clientSet
            if client.isAllocated
                Continue
            value ← 5000 + client.breadQuant - Euclidean(start, client.vertex)
            for client2 ∈ van.allocatedClients
                value ← value + ABS(client.time - client2.time)
                    - Euclidean(client.vertex, client2.vertex)
            if (value > highest AND client.breadQuant <= capacity)
                highest ← value
                chosen ← client
        if chosen=NULL
            Break
        v.addClient(chosen)
        count++
        capacity ← capacity - chosen.breadQuant
    Return count
```

Figura 19 - Pseudocódigo Alocação Gananciosa

```
function OPTIMIZE_VANS()
    for C ∈ clientSet
        if (!C.isAllocated) → SplitDelivery(C)

    lastVan ← Ø
    for V ∈ reversedVanSet // Get last used van
        if V.hasClients
            lastVan ← V
            Break

    for van ∈ vanSet
        if van=lastVan
            Break
        client, oldCost ← van.getWorstClientInRange(lastVan)

        if client=NULL
            Continue

        newCost = lastVan.calculateCost(client)
        if newCost < oldCost
            lastVan.addClient(client)
            van.removeClient(client)
```

Figura 20 - Pseudocódigo Otimização de Carrinhas

```
function SplitDelivery(client)
    possible ← False
    totalCapacity ← 0

    for V ∈ vanSet
        totalCapacity += V.availableCapacity
        if totalCapacity >= client.breadQuant
            possible ← True
            Break

    if possible
        for V ∈ vanSet
            if V.availableCapacity >= client.breadQuant
                V.addClient(client)
                Break
            else
                newClient ← van.availableCapacity
                clientSet.add(newClient)
                client.removeBread(newClient.breadQuant)
```

Figura 21 - Pseudocódigo Dividir Entrega

Pseudocódigo - Algoritmo Completo

```
function SolveMultipleVans(useKnapsack, optimize)
    SORT_BY_CAPACITY(vans)
    clientsAllocated ← 0
    if useKnapsack
        for van ∈ vans
            values ← Ø // Array
            if (clientsAllocated=clientSet.size)
                Break
            for client ∈ clientSet
                PUSH(values, client.breadQuant + 10)
            clientsAllocated ← clientsAllocated + KnapSack(van, values)
    else
        for van ∈ vans
            if (clientsAllocated=clientSet.size)
                Break
            clientsAllocated ← clientsAllocated + GreedyAllocation(van)

    if optimize
        OPTIMIZE_VANS()
```

Figura 22 - Pseudocódigo 3ª fase

Análise da Complexidade - *Knapsack Allocation*

O primeiro algoritmo possível de alocação de clientes a uma dada carrinha é baseado no famoso *Knapsack Problem* resolvido com programação dinâmica, sendo o valor dado por uma constante e pela quantidade de pães na encomenda. Assim sendo, o algoritmo dá-se pelos seguintes passos:

- Criação de um vetor de valores, sendo que estes são calculados pela fórmula:

  `Value = numBreads + 10`

- Criação de um vetor de clientes, contido pelos que não foram previamente alocados. Inerentemente, este passo possui uma complexidade temporal e espacial O(|C|).
- A seguir segue-se o cálculo da tabela de soluções (visto tratar-se de programação dinâmica). A complexidade (temporal e espacial) deste depende do número de clientes não alocados e do número da capacidade da carrinha (Representada por W) em O (W |C|)
- Por fim, são retirados da tabela os clientes com maior valor combinado, o que pode ser feito em O(|C|).

Assim sendo, a complexidade temporal e espacial deste algoritmo é O (W |C|), o que é extremamente rápido. No entanto, este algoritmo não obtém resultados tão perto dos ótimos como se pretendia, pelo que se resolveu criar um mais dispendioso, mas que conseguisse obter melhores resultados.

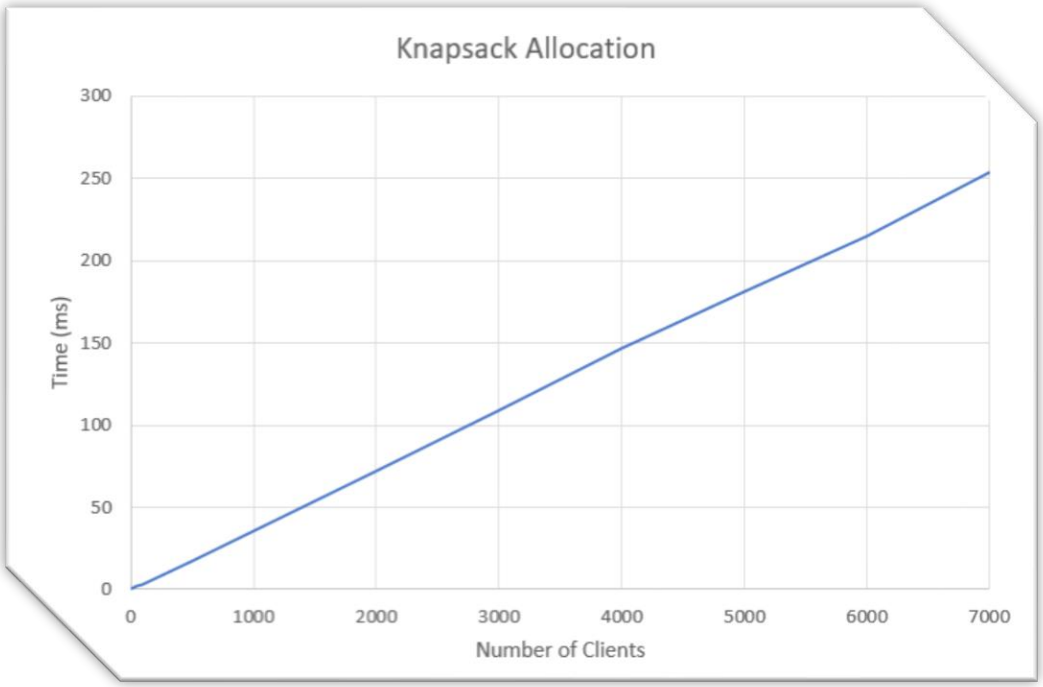

Figura 23 - Análise Empírica Knapsack

Nota: Análise feita com uma carrinha com 2000 de capacidade.

Análise da Complexidade - Alocação Gananciosa

Com o objetivo de obter uma solução heurística mais próxima da ótima, foi criado um algoritmo ganancioso que tem em consideração, a cada iteração, da distância dos clientes aos que já foram alocados (e ao vértice de início) e a hora de entrega destes. Assim sendo, foi usada a seguinte fórmula para calcular, dinamicamente em cada iteração, o valor dos clientes:

$$Value = 5000 + numBreads + \sum_{v(i)} \in V\ (|v(i).time - time| - Euclidean(v(i).pos, pos))$$

É usada a constante 5000 para equilibrar os valores, estando mais próximos de 0, sendo que podem ser negativos.

Este algoritmo repete, até a carrinha já não tiver capacidade suficiente para mais clientes, um processo que consiste em escolher, em cada iteração, o cliente com mais valor, que ainda não foi alocado e cuja quantidade de pães não excede a capacidade da carrinha.

Como o cálculo valor implica percorrer a lista de clientes já alocada, o algoritmo tem a complexidade O (W $|C|^2$), sendo que W representa a capacidade total da carrinha, e a espacial O (|C|), devido a uso de um atributo auxiliar em *Client*.

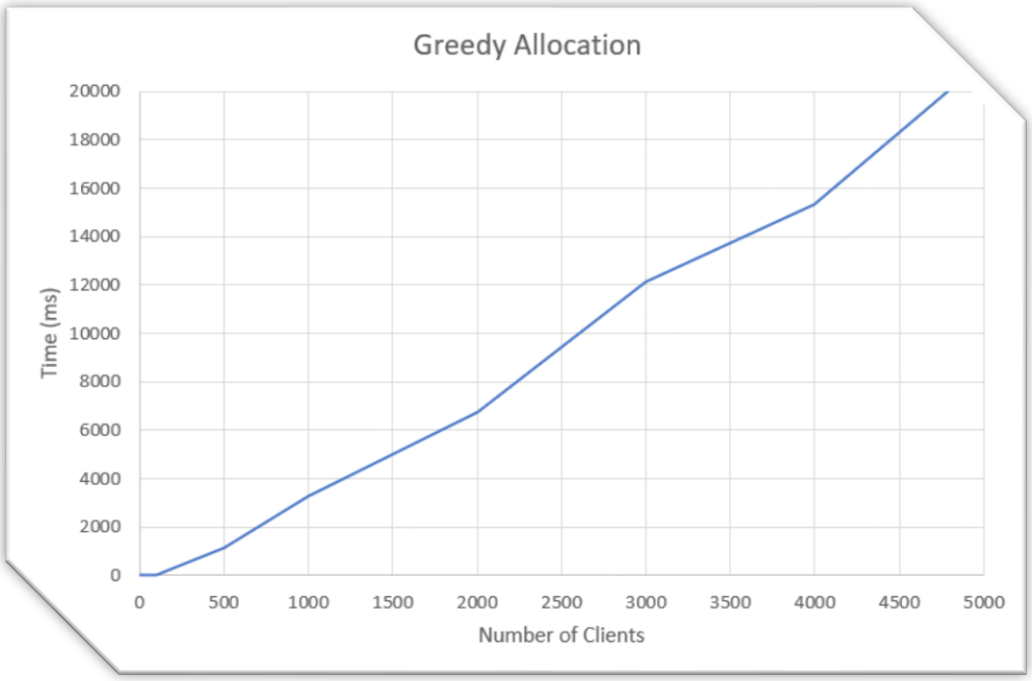

Figura 24 - Análise Empírica Alocação Gananciosa

Nota: Análise feita com uma carrinha com 2000 de capacidade.

Análise da complexidade - Otimização da Alocação

De modo a colmatar possíveis aproximações grotescas (tais como a última carrinha alocada ficar com muito espaço livre ou a existência de clientes cuja encomenda não conseguir ser alocada, por falta de espaço nas carrinhas), o utilizador tem a possibilidade de usar um algoritmo de otimização, que consegue obter melhores resultados.

Este algoritmo passa pelos seguintes passos:

- Percorre a lista de clientes e, se encontrar alguns que não foram alocados, tenta separar as suas encomendas e atribuí-las a várias carrinhas. Este processo tem uma complexidade temporal de O (|V||C|) e espacial de O (1), desprezando a memória extra utilizada na representação de novas encomendas.
- Percorre a lista de carrinhas à procura da última que foi utilizada. Inerentemente, em O(|V|), em relação ao tempo, e O (1) em relação ao espaço.
- Percorre de novo o conjunto de carrinhas, retirando, de cada uma destas, o seu cliente mais custoso que poderia ser alocado à carrinha encontrada no passo anterior. Depois, calcula o custo desse cliente na nova carrinha e, se for menor, troca-o de veículo. Isto é executado em O (|V| $|C|^2$) e tem uma complexidade espacial O (1).

Assim sendo, no seu todo, o algoritmo de otimização tem uma complexidade temporal O (|V| $|C|^2$), tendo em conta os fatores dominantes, e espacial O (1).

Nota: A análise empírica é feita juntamente com o algoritmo completo.

Análise da complexidade - Algoritmo completo

Terminada a análise de todos algoritmos usados, o algoritmo da 3ª fase, de acordo com as escolhas do utilizador, pode ter as seguintes complexidades, considerando que usa o algoritmo da 2ª fase para todas as carrinhas alocadas e que, antes da alocação, ordena todas as carrinhas por ordem decrescente de capacidade:

Knapsack sem otimização: O (|V| (log|V| + W |C)|)) temporal e O (W |C| + |V| + |E|) espacial

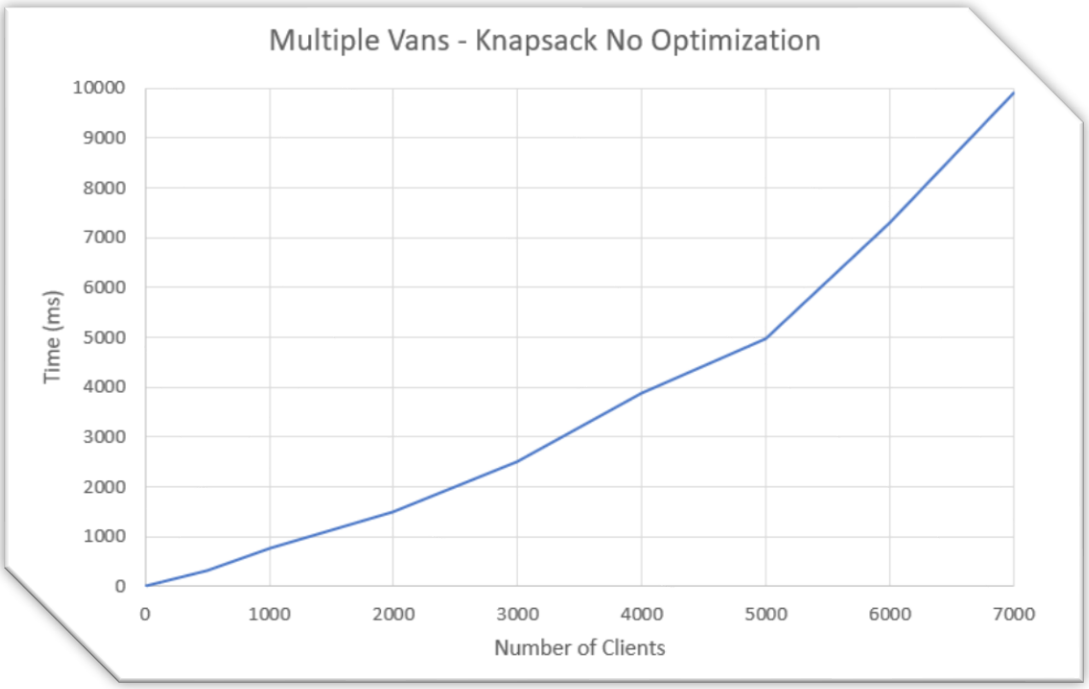

Figura 25 - Análise Empírica Knapsack Sem Otimização

Knapsack com otimização: O (|V| (|V| + W |C|)) temporal e O (W |C| + |V| + |E|) espacial

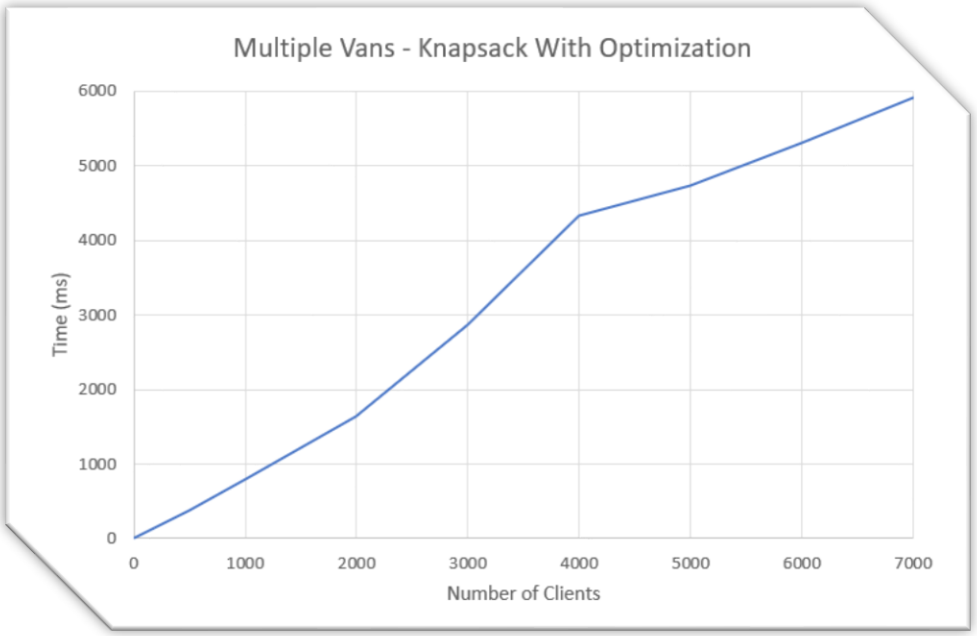

Figura 26 - Análise Empírica - Knapsack Com Otimização

Nota: Para ser notório o crescimento temporal em função do número de clientes na utilização do algoritmo de otimização, foi usado um caso ligeiramente diferente ao de cima, pelo que os resultados não devem ser comparados.

Gananciosо sem otimização: O (|V| (log|V| + W $|C|^2$)) temporal e O (|C| + |V| + |E|) espacial

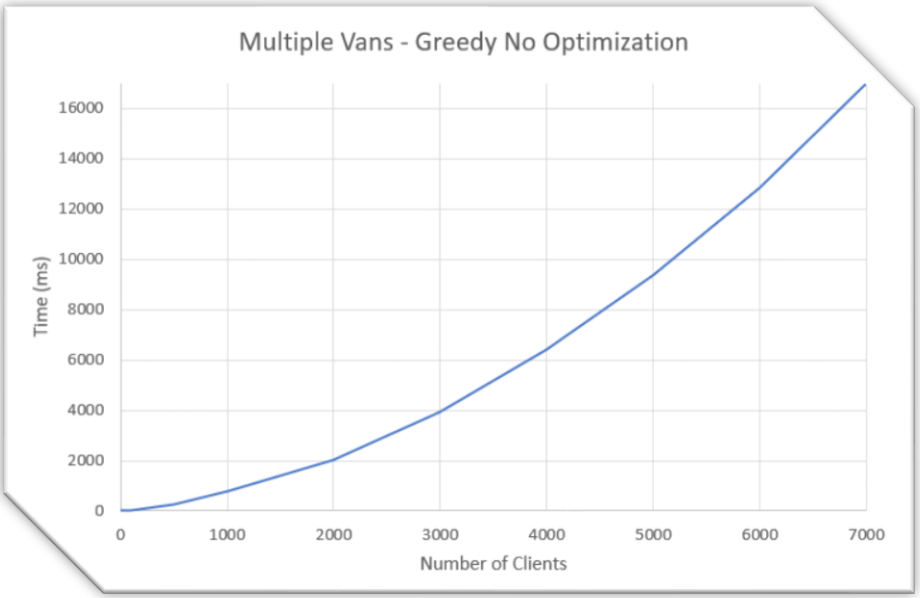

Figura 27 - Análise Empírica Gananciosо sem Otimização

Ganancioso com otimização O (|V| (|V| + W |C|²)) temporal e O (|C| + |V| + |E|) espacial

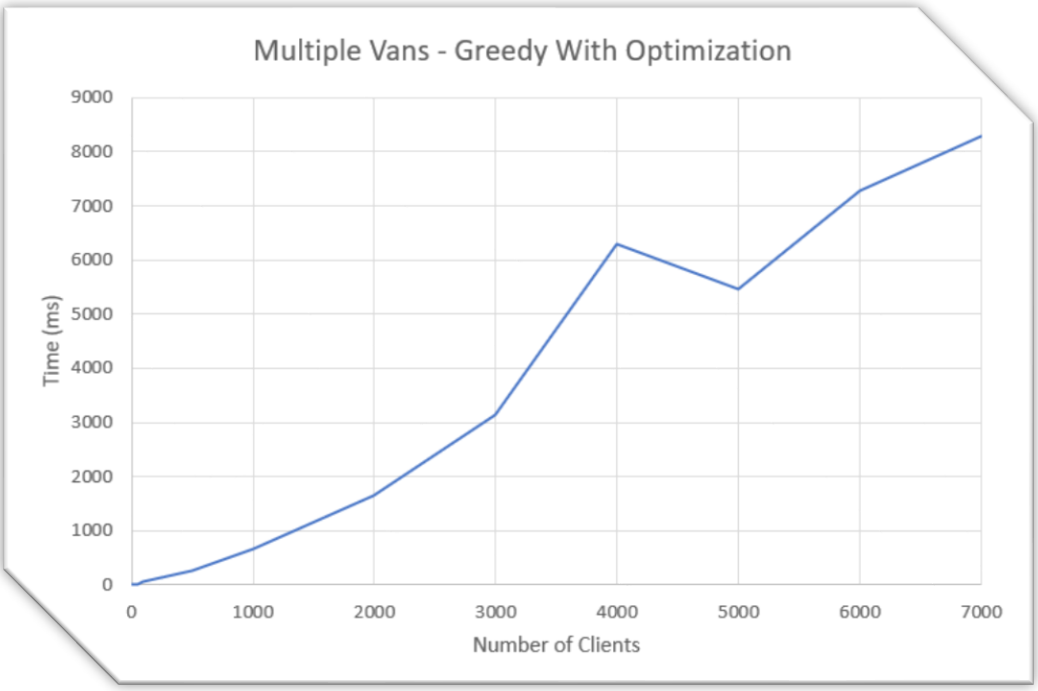

Figura 28 - Análise Empírica Ganancioso com Otimização

O pico que se pode ver na figura deriva-se da natureza do algoritmo de otimização, que depende não só do tamanho dos dados de entrada como da situação em questão.

Nota: Aplicou-se a mesma estratégia explicada no algoritmo *knapsack* com otimização.

Casos de Utilização

A aplicação desenvolvida tem uma interface simples onde o utilizador poderá inserir os dados de entrada e escolher as opções que pretende, tais como:

- Se pretende carregar um ficheiro de texto com toda a informação necessária para o algoritmo ou se quer inserir a informação manualmente.
- Se quer guardar o resultado do programa num ficheiro de texto.
- Se quer utilizar uma única carrinha com capacidade infinita (e dentro desta opção, se quer inserir horas de entrega ou não) ou várias com capacidades limitadas.
- Se prefere uma solução mais otimizada (e possivelmente mais demorada) ou menos (e mais rápida).
- Nome do ficheiro contendo os dados de entrada ou apenas do grafo (coordenadas reais retiradas do OpenStreetMaps).
- Caso escolha fazê-lo manualmente, pode inserir informação sobre a(s) carrinha(s), os clientes, o raio de entrega da padaria e as margens de falha para a hora preferencial dos clientes. O vértice de partida será o primeiro no ficheiro do grafo.
- É possível, ainda, visualizar o grafo representante dum mapa e caminho efetuado pelas carrinhas, através do *GraphViewer*.
- Também existe uma opção de gerar um ficheiro de *input* aleatório, carregando na tecla 'L' no início do programa.

Além disso, o programa terá as seguintes funcionalidades:

- Cálculo do caminho para realizar as encomendas.
- Verificação da acessibilidade dos destinos pretendidos (moradas dos clientes).
- Organização e alocação de encomendas a uma lista de carrinhas.
- Toda a informação obtida (trajeto, entregas, tempo total, etc.) pelos pontos anteriores será mostrada ao utilizador (e guardada, se assim for pretendido).

Conclusão

Em suma, o caso de estudo em questão, a criação de um programa que trata da distribuição de pão numa vila em tempo de COVID-19, foi analisado e desenvolvido de modo a criar um produto final eficaz e funcional. Para tal, o seu desenvolvimento foi dividido em três partes, progressivamente mais complexas.

Para a execução do projeto, foi feito um estudo acerca dos problemas intrínsecos ao tema em questão, tendo como objetivo o planeamento e discussão sobre as técnicas e algoritmos mais adequados e eficientes.

Com efeito, através da utilização de grafos, procedemos à construção do pseudocódigo, e posterior implementação, para as três fases previamente enunciadas. Foram implementados vários algoritmos para as diferentes fases, pedindo a preferência do utilizador, tendo em conta os seus dados.

A primeira análise do problema, mais simples, consistiu em elaborar uma solução para um problema semelhante ao NP-hard, mais conhecido como **TSP,** que tratava a deslocação de uma única carrinha, sem restrição de ordem de entrega pela hora das encomendas, nem limite de capacidade da carrinha e retorno ao ponto de partida, ou seja, a padaria. Tudo isto, no menor tempo possível.

De seguida, na segunda fase, criou-se um algoritmo capaz de executar o problema, tendo em atenção a restrição da hora de entrega, equilibrando o tempo de atraso nas entregas através da alteração de ordem de passagem pelos pontos de interesse.

Por fim, tendo em conta o que já tinha sido elaborado, foram implementados algoritmos de alocação de modo a permitir múltiplas carrinhas com diversas capacidades, utilizando, depois, um algoritmo desenvolvido para a segunda fase do projeto.

Assim, é importante referir quais foram os métodos/algoritmos utlizados/analisados ao longo do projeto, sendo eles, o **Dijkstra**, **Floyd-Warshall**, **Nearest Neighbor**, **Held-Karp**, **Trajan, Bitonic Tour** e **Kosaraju**. Além disso, recorreu- se a vários conceitos, entre eles, **recursividade**, ***bruteforce***, **algoritmos gananciosos e programação dinâmica (Knapsack Problem)**.

Relativamente à segunda parte do projeto (implementação da solução proposta), devido à natureza variada dos grafos fornecidos (grelhas ou mapas reais), alterou-se a estrutura de grafo sugerida anteriormente, de modo a transformá-lo, dinamicamente e de acordo com o mapa escolhido, em dirigido ou não dirigido.

Inicialmente, era pretendida a implementação de mais algoritmos (e.g. força bruta) para as diferentes fases, de modo a conceber uma comparação mais aprofundada da performance destes. No entanto, devido a dificuldades na integração do *GraphViewer*, na utilização e manipulação dos mapas fornecidos e no cálculo de dados para a análise empírica (*script* para geração automática de dados de entrada com tamanho considerável, computador com condições estáveis e sem oscilações no tempo de execução, e consideração de fatores dificilmente controláveis como otimizações do compilador em execuções sequenciais do mesmo algoritmo), surgiram atrasos, pelo que nos mantivemos pela implementação e análise dos algoritmos mais importantes, que foram devidamente testados e otimizados.

Por outro lado, para facilitar o uso da aplicação e a escolha dos métodos e dados e serem usados, foi criada uma interface textual que comunica com o utilizador, permitindo-lhe ver e comparar resultados de forma simples e rápida.

Enquanto analisávamos, empiricamente, os algoritmos da terceira fase, surgiu uma situação peculiar que nos chamou a atenção. Com certos dados de entrada, usar algoritmos de alocação com complexidade temporal superior resultava em tempos de execução similares e, por vezes, inferiores aos obtidos com algoritmos de complexidade inferior. Primeiramente, pensamos ser algum tipo de lacuna, mas depois apercebemo-nos que, em certos casos, uma melhor alocação (mesmo que mais dispendiosa) resultava em clientes mais próximos entre si, pelo que o algoritmo de *Dijkstra*, que termina quando encontra o seu destino, executava mais rapidamente. Com isto, concluiu-se que, dependendo da situação, podemos até obter uma solução melhor sem custos temporais acrescidos! No entanto, há também casos onde o tempo de execução é realmente mais elevado, pelo que se deve escolher o algoritmo tendo em conta a situação em questão.

Finalizando, concluímos que existem diversos algoritmos capazes de dar boas soluções ao problema sugerido, sendo que não existe um vencedor óbvio. O melhor algoritmo será sempre o que é medido face às necessidades do utilizador, sejam elas a rápida execução, a precisão da solução, a importância da alocação e da rota calculada, ou apenas uma solução simples e breve.

Contribuição

Bruno Rosendo:

- **1ª Parte:**
  - Descrição do problema
  - Formalização do problema
  - Algoritmos para as 1ª e 2ª fases
  - Casos de utilização
  - Bibliografia
- **2ª Parte:**
  - Estruturas de dados e Classes utilizadas (relatório)
  - Análise teórica e algoritmos implementados (relatório): DFS, pré-processamento, *nearest neighbour*, 2ª fase, 3ª fase.
  - Análise empírica (resultados e relatório)
  - Parte final da conclusão (relatório)
  - Modelação inicial do problema (posteriormente adaptada por todos)
  - Algoritmos implementados: 2ª fase (ganancioso com *Dijkstra*), alocação de clientes às carrinhas (ambas as opções) e respetivas otimizações, DFS, filtragem de vértices, *Dijkstra* clássico
  - Interface- comunicação com o utilizador, integração da classe *Bakery*, leitura de dados por *input*, mostragem dos resultados na saída padrão

Domingos Santos:

- **1ª Parte:**
  - Algoritmos para a 3ª fase
  - Conclusão
  - Estruturação do documento
  - Contribuição
- **2ª Parte:**
  - Conclusão (relatório)
  - Estruturação do relatório
  - Estrutura do ficheiro de dados de entrada e consequente leitura
  - Contribuições para a apresentação do *GraphViewer*
  - *Script* de geração de dados de entrada

João Mesquita:

- **1ª Parte:**
  - Descrição do problema
  - Pré-processamento
  - Análise da conetividade
  - Algoritmos de percurso de duração mínima entre dois pontos
  - Algoritmos de percurso de duração mínima entre todos os pares de pontos

- **2ª Parte:**
  - Análise teórica e algoritmos implementados (relatório): *Tarjan*, *Dijkstra*
  - Análise da conetividade (relatório)
  - Estrutura de dados utilizada para o grafo
  - Integração e manipulação do *GraphViewer*
  - Algoritmos implementados: *nearest neighbour*, filtragem de clientes, *Tarjan*, variações de *Dijkstra* (bidirecional e paragem condicional)
  - Interface- mostragem dos resultados no *GraphViewer*, opção de verificação de componentes fortemente conexos
  - Contribuições para a *script* de geração de dados de entrada

Bibliografia

- Slides utilizados nas aulas teóricas de CAL, pelos professores: R. Rossetti, A. Rocha, L. Ferreira, G. Leão, F. Ramos e J. Fernandes
- "Introduction to Algorithms", Thomas H. Cormen
- Travelling Salesman Problem- https://en.wikipedia.org/wiki/Travelling_salesman_problem https://en.ppt-online.org/31503
- Vehicle routing problem- https://en.wikipedia.org/wiki/Vehicle_routing_problem
- Strongly Connected Component- https://en.wikipedia.org/wiki/Strongly_connected_component
- Heuristics and Local Search- https://paginas.fe.up.pt/~mac/ensino/docs/OR/CombinatorialOptimizationHeuristicsLocalSearch.pdf
- Nearest neighbour algorithm- https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- Tarjan's strongly connected components algorithm- https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
- Kosaraju's algorithm- https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm
- Held–Karp algorithm- https://en.wikipedia.org/wiki/Held–Karp_algorithm
- Bitonic tour- https://en.wikipedia.org/wiki/Bitonic_tour
- Problema da Mochila- https://pt.wikipedia.org/wiki/Problema_da_mochila
- A General VNS heuristic for the traveling salesman problem with time windows- https://core.ac.uk/download/pdf/82432413.pdf
- An Optimal Algorithm for the Traveling Salesman Problem with Time Windows- https://pubsonline.informs.org/doi/pdf/10.1287/opre.43.2.367
- Vehicle Routing Problem with Time Windows and Simultaneous Delivery and Pick-Up Service Based on MCPSO- https://www.hindawi.com/journals/mpe/2012/104279/
- Dijkstra vs Bi-directional Dijkstra Algorithm on US Road Network- https://www.youtube.com/watch?v=10VuQsxkhY0

- Tarjan's Algorithm to find Strongly Connected Components- https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/