O Padeiro da Vila em Época Covid

16 / 04 / 2021

—

Conceção e Análise de Algoritmos

—

Bruno Rosendo up201906334

Domingos Santos

João Mesquita

Índice

Descrição do Problema

O trabalho incide sobre a distribuição de pães, começando na padaria e passando pela casa de todos os clientes que aderiram. Por fim, o padeiro tem ainda de voltar à padaria.

Cada cliente indica a hora preferencial para que o pão lhe seja entregue, tendo uma margem de tolerância definida (antes e depois da hora). Quando o padeiro chega a uma habitação, demora uma certa quantidade de tempo 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, 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, teremos 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, de modo a 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 de cada entrega também será diferente (o condutor pode ser mais ou menos falador).

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 dirigido pesado, 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 dirigido pesado, 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 obras públicas ou demasiado afastadas da padaria, tendo em conta que a padaria do Sr. Sílvio apenas efetua encomendas na vila da Tocha, ou seja, num raio (Ra) à volta da padaria (simplificando).

Após a primeira filtragem, será necessário remover os vértices que não pertencem ao mesmo componente fortemente conexo do grafo onde a padaria 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

Será 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 prova-se ser computacionalmente exigente, visto que calcula a distância entre todos os pares de vértices e não apenas os necessários.

Por fim, a aplicabilidade destes algoritmos requer uma análise da eficiência do programa tendo em conta a dimensão do mapa considerado. Naturalmente, a sua utilização irá depender também do algoritmo escolhido para resolver a 2ª fase do problema, já que nem todos usufruem dos cálculos provenientes do mesmo.

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. 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:

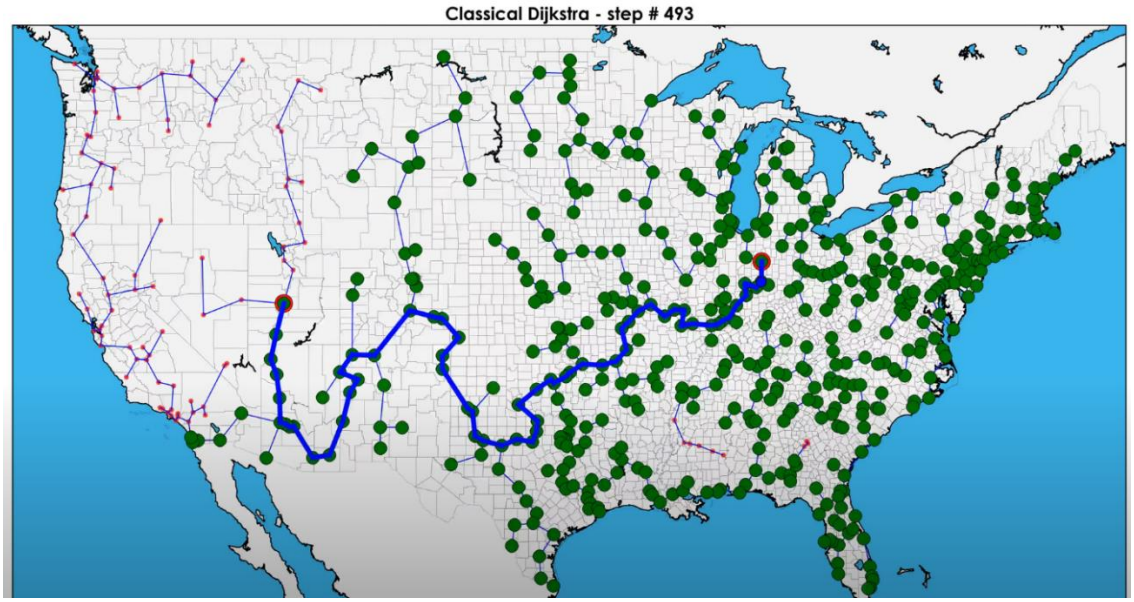

Figura 1 - Pesquisa com Algoritmo de Dijkstra Unidirecional

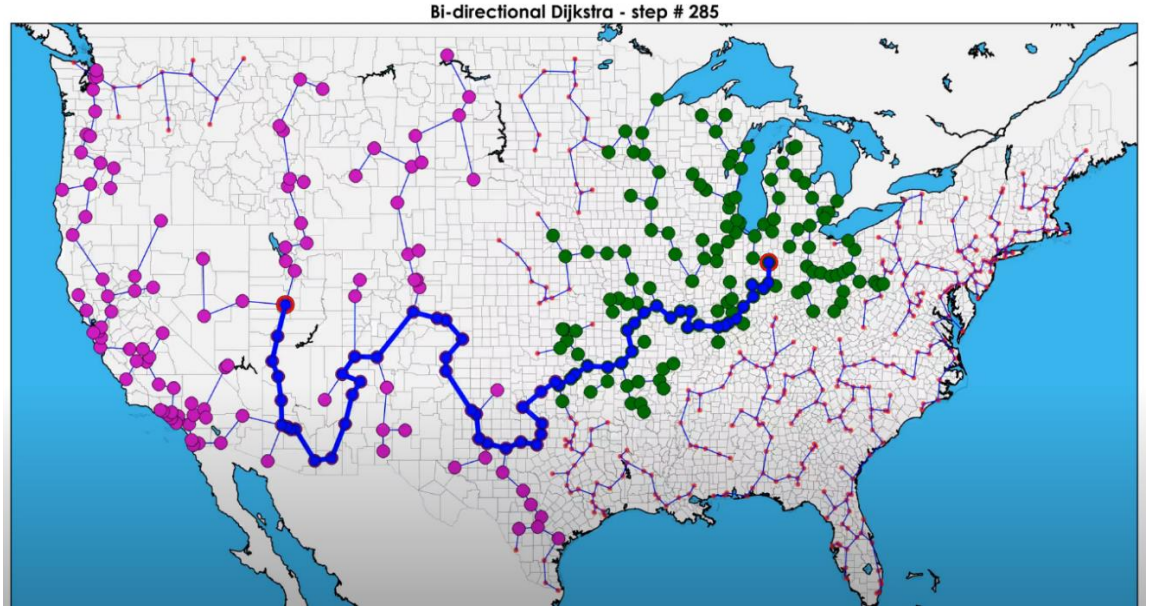

Figura 2 - Pesquisa com algoritmo de Dijkstra Bidirecional

Percurso de duração mínima entre todos os pares de pontos

Como já foi referido na fase de pré-processamento, surge a hipótese de pré-calcular os percursos de duração mínima entre todos os pares de vértices, aumentando a eficiência do resto do programa, caso seja escolhido um método que usufrua destes cálculos.
Com efeito, serão consideradas duas abordagens:

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)
```

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, teremos de encontrar novos algoritmos que tomem isto em consideração ou, em alternativa, alterar as soluções da 1ª fase de modo a integrar as novas mudanças. Devido a caminhos mais curtos e horas de entrega diferentes, é agora possível passar por casa de um cliente mais do que uma vez (embora se faça uma única entrega).

Em relação a uma solução exata, poderá ser usada **força bruta**, tal como se fez para a 1ª fase, calculando os tempos de viagem e atraso para todas as ordens possíveis de clientes. Estes tempos podem 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 “a caminho” 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, terão de ser consideradas otimizações a aplicar ao algoritmo. Uma delas poderá ser 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.

Será 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
```

Outra técnica que será considerada, embora mais complexa e com uma implementação mais trabalhosa, é 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, poderemos 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 3 - 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 4 - 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, integrar-se-á 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.

Uma possível abordagem será resolver o problema de alocação das carrinhas independentemente, usando, após esta primeira fase, 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, caímos 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, podemos usar uma abordagem gananciosa e preencher primeiro as carrinhas com maior capacidade, dando prioridade a clientes que moram perto uns dos outros, até acabarem as encomendas.

De forma a otimizar o algoritmo, e como é provável que a última carrinha fique com algum espaço livre, podemos percorrer 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 consideravelmente mais perto das moradas dos clientes alocados ao último veículo *(edge cases)*.

```
// 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)
```

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

Este método armazena recursivamente o resultado para que cada vez que um problema é dado, é retornado o resultado e não 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.

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. Apresentam-se de seguida dois algoritmos para resolver este problema:

Algoritmo de Kosaraju

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. Apesar de executar em tempo linear, executa apenas uma pesquisa em profundidade, sendo por isso mais eficiente.

Casos de Utilização

A aplicação a desenvolver terá 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 ótima (e possivelmente mais demorada) ou aproximada (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.

Além disso, o programa terá as seguintes funcionalidades:

- Cálculo do caminho ótimo (ou aproximado) 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 a 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 para as três fases previamente enunciadas. Deste modo, um dos principais problemas associados a esta estratégia foi a implementação de um algoritmo que permitisse entregar o pão num caminho mais curto e com menor tempo de atraso, tendo em conta a hora de cada encomenda, os pontos de entrega a visitar e a distância entre eles.

A primeira análise do problema, mais simples, consistia 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, com base na fase anterior, 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 os algoritmos já elaborados, foram efetuadas alterações de modo a resolver o mesmo problema de estudo, tendo em conta as duas fases já concluídas, bem como novas funcionalidades, nomeadamente, a escolha do número de carrinhas que tratarão das entregas e a alocação otimizada das mesmas aos clientes.

Assim, é importante referir quais foram os métodos/algoritmos utlizados 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)**.

Contribuição

Cada membro do projeto realizou as tarefas a si atribuídas, nomeadamente:

- **Bruno Rosendo (1/3)**:
  - Descrição do problema
  - Formalização do problema
  - Algoritmos para as 1ª e 2ª fases
  - Casos de utilização
  - Bibliografia
- **Domingos Santos (1/3)**:
  - Algoritmos para a 3ª fase
  - Conclusão
  - Estruturação do documento
  - Contribuição
- **João Mesquita (1/3)**:
  - 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

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=1oVuQsxkhY0