FlightNet: Gestão de Voos

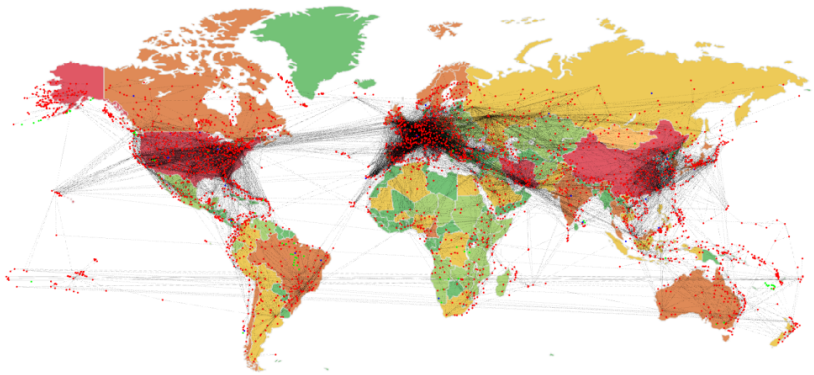
Trabalho Prático 1

CAL 20/21

Turma 5, Grupo 4

André Lino dos Santos - up201907879@edu.fe.up.pt
Edgar Ferreira da Torre - up201906573@edu.fe.up.pt
Maria José Valente da Silva Carneiro - up201907726@edu.fe.up.pt

24 de Maio de 2021



Conteúdo

1 Introdução

O projeto que desenvolvemos, no contexto da Unidade Curricular de Concepção e Análise de Algoritmos(CAL) tem como objetivo implementar um sistema de gestão de voos internacionais entre aeroportos.

1.1 Descrição do Tema

A FlightNet é uma empresa que gere voos comerciais em todo mundo. O objetivo principal é otimizar o transporte de passageiros de modo a que os trajetos escolhidos sejam os mais eficientes. Os aviões andam em rotas circulares, ou seja, após transportar os passageiros e a tripulação a outros aeroportos, voltam ao aeroporto onde começou a rota. Nesses percursos, os aviões fazem escalas em função de vários fatores: rotatividade da tripulação, estado atmosférico em cada aeroporto, gastos de combustivel, destino da viagem e quantidade de passageiros.

1.2 Decomposição do Problema

O problema pode ser decomposto em 2 fases inicias: uma primeira em que apenas consideramos o trajeto de um avião e uma fase posterior em que é introduzida uma frota inteira para vários aeroportos. Por fim serão feitas otimização às anteriores, com o objetivo de alcançar resultados mais positivos e eficientes.

1.2.1 1ª Etapa - Um avião com lotação e escalas

Numa fase inicial, será apenas usado um avião para fazer o transporte de pessoas. As condicionantes como combustível e rotatividade da tripulação serão desde já consideradas ao processar o grafo. O foco será escolher o caminho mais curto desde o aeroporto de partida até aos de chegada, seja ele direto ou com escalas.

Os voos com escala serão prioritizados visto que, caso haja lugares livres e passageiros a querer ir para destinos intermédios(escalas), os lugares são ocupados, poupando futuras viagens. Se nesses destinos intermédios ainda existirem lugares livres e passageiros com destinos comuns à rota do avião em questão, estes poderão também entrar no voo.

Nesta fase vai ser possível descobrir o efeito causado noutros aeroportos por um voo e testar a eficiência do método para os passageiros do local de partida.

1.2.2 2ª Etapa - Vários aviões com lotação e escalas

Numa fase posterior, vão ser introduzidos vários aviões em múltiplos aeroportos. As condicionantes serão as mesmas da primeira iteração, com a diferença de que os voos terão vários aviões para encarregar da viagem. Para além disso, esta iteração já contabilizará vários aeroportos de partida.

Assim, nesta etapa já será possível transportar todos os passageiros, mesmo que haja pouca eficiência nalguns casos. Tal como na primeira iteração, haverão alguns voos ineficientes, quando houver poucos clientes para cada rota.

1.2.3 3ª Etapa - Otimizações das rotas

Após a construção do algoritmo base referenciado nas iterações anteriores, a gestão de voos sofrerá algumas otimizações.

Por exemplo, se houver um número muito reduzido de passageiros com um certo destino, após a definição de todas os outros voos, não se criará uma nova rota que satisfaça esses passageiros. Nesta situação, ficam sem voo, mas por outro lado o transporte de pessoas por distância/tempo será muito mais eficiente e, por consequência, mais lucrativo. Enquanto que nos anteriores o objetivo é minimizar o tempo para todas as pessoas, neste será maximizar o lucro, ignorando viagens pouco requeridas.

Outra otimização possivel, seria utilizar o primeiro método até restarem os passageiros com destinos pouco concorridos e utilizar uma distribuição mais dispersa. Neste caso, as rotas iriam abranger mais locais de escala, fazendo um trajeto circular. Não seria uma solução tão eficiente como a anterior, mas transportaria todos os passageiros.

2 Utilização e Funcionalidades

O programa desenvolvido apresentará uma interface simples e acessível à interação do utilizador com as funcionalidades implementadas. Essas funcionalidades serão, entre outras:

- Análise da conectividade do grafo.
- Visualização dos aeroportos e respetivas rotas aéreas que os ligam, através do GraphViewer.
- Cálculo do percurso mais curto entre 2 pontos.
- Seleção de condicionantes como a capacidade dos aviões, os limites máximos de combustível ou a duração máxima de um turno da tripulação.
- Cálculo das rotas mais eficientes que permitam o transporte de um dado número de passageiros com diferentes destinos, a partir de um ou vários aeroportos, recorrendo a uma quantidade variável de aviões.

De modo a conseguir atingir estes objetivos, foi necessário criar algumas ferramentas, que serão agora enumeradas e descritas.

Para permitir que o utilizador escolha que informações quer obter, foi necessário construir uma rede de menus que são mostrados através de uma interface baseada em texto no terminal. Em cada menu existem várias opções com um número que as identificam, sendo que o utilizador tem de selecionar uma delas. Algumas das opções reencaminham o utilizador para outros menus, onde o processo de cima se repete até que uma das opções seja efetivamente uma ação executável.

```
->-<->-<->-<->-<->-<->-FLIGHTNET-<->-<->-<->-<->-<->-<
1. Check shortest path between two airports
2. Check flights from an Airport
3. Show Map
4. Adjust FlightNet Settings
5. Exit
Select a valid option:
```

Em relação à conectividade, é possivel saber o tempo de execução dos algoritmos de identificação de componentes fortemente conexos. Esta triagem pode ser executada pelos algoritmos de Tarjan e Kosaraju, porém na interface só consta o algoritmo de Tarjan. O utilizador pode ainda escolher o tempo máximo que o um avião consegue voar de seguida, o tempo máximo de trabalho de uma tripulação e a lotação de cada avião.

A maioria destas ações passa por chamar uma função específica de uma classe que implementa um certo algoritmo, sendo que o resultado da função é escrito como texto. Contudo, na ação de mostrar o mapa mundial é preciso enviar uma

imagem. Para conseguir fazer essa ação, foi preciso criar um visualizador de grafos, que abre uma nova janela para mostrar a imagem completa do globo. A classe GraphViewer também foi fornecida, porém foi criada uma nova classe GraphMap, que contém uma instância de GraphViewer, juntamente com outros métodos. É numa instância desta classe GraphMap que ocorre todo o processo necessário para mostrar o globo e visualizar os aeroportos e respetivas rotas.

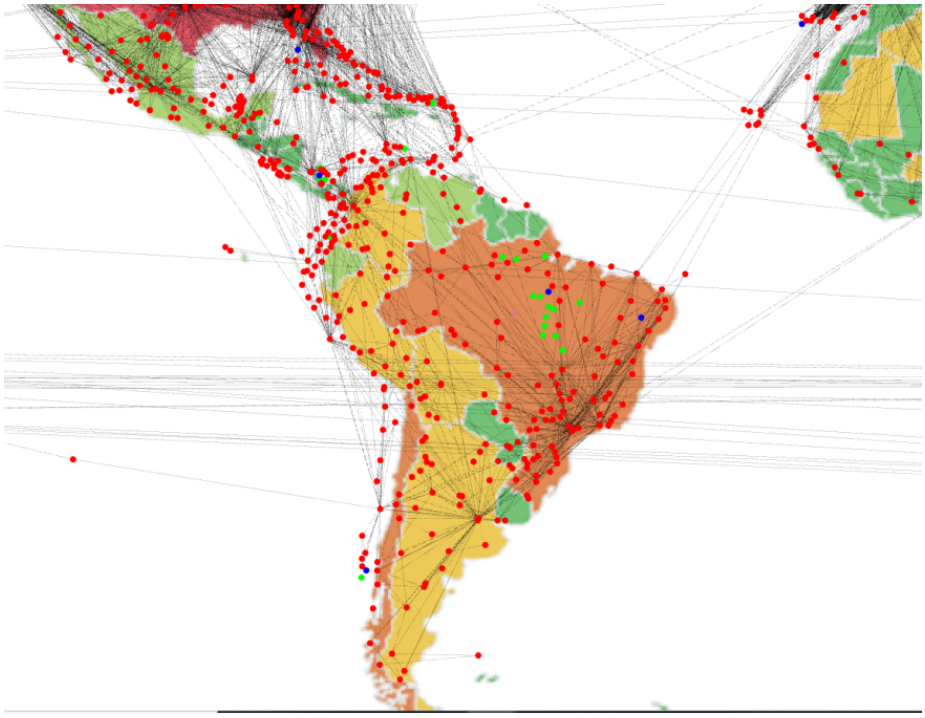


Figura 1: Mapa de aeroportos da américa do sul.
Os vértices vermelhos representam aeroportos funcionais.
Os vértices azuis representam aeroportos desconsiderados devido ao mau tempo.
Os vértices verdes representam aeroportos desconsiderados por não serem fortemente conexos.

É também possivel calcular o caminho mais rápido entre 2 pontos, fazendo recurso a um algoritmo à escolha do utilizador, bem como mostrar essa mesma rota no mapa.

Sobre os cenários implementados, é possivel executar o programa como planeado (implementação com várias viagens de um aeroporto, com todas as condicionantes, processadas sequencialmente). Neste cenário, são aplicadas restrições como combustivel disponivel e rotatividade da tripulação, e são usadas técnicas que permitem optimizar a escolha de rotas, como o aproveitamento de escalas. Após processar as viagens de um aeroporto, é possivel ver as estatísticas do transporte, bem como ver a quantidade de pessoas transportada por rota.

3 Estruturas de dados utilizadas

A maioria das operações feitas por um objeto de uma classe numa estrutura de dados de outro objeto é feita através de apontadores, sendo evitado a ação bastante custosa de copiar a estrutura inteira. As estruturas utilizadas foram diversas:

3.1 Vetor

A estrutura de dados mais usada foi o vetor (std::vector), devido à sua versatilidade e a sua função embutida de pesquisa e acesso por indice. Os vetores são estruturas de dados sequenciais que usam um array alocado dinamicamente para guardar os seus elementos, e que conseguem alocar memória extra em caso de necessidade de expansão. Assim, o vetor guarda a maioria das informações, desde voos a aeroportos.

3.2 Pilha

A pilha (std::stack) foi bastante usada no âmbito deste projeto. Esta estrutura de dados também é sequencial, contudo, difere em que o último objeto a ser inserido é o único que pode ser acedido e removido. Assim, para aceder a elementos que não estão no topo da pilha, é necessário remover o último objeto inserido, verificar se o seguinte é o elemento pretendido, e repetir até efetivamente o localizar, o que é um processo muito custoso e dai deriva a prioridade que atribuimos a outras estruturas de dados como o vetor quando é necessário aceder a elementos no meio da estrtura ou verificar se um elemento existe. Contudo, em casos que apenas é de interesse o último elemento inserido, é bastante útil e eficiente. Em certos algoritmos como o Kosaraju, foi necessário o recurso à mesma.

3.3 Fila

A fila (std::queue) é uma estrutura de dados bastante semelhante à pilha, com a diferença que o primeiro objeto a ser inserido é o único que pode ser acedido e removido, ao contrário da pilha onde é o último objeto. Visto que o seu esquema é idêntico ao da pilha, a fila decorre das mesmas vantagens e desvantagens que a pilha. Para representar a fila de tripulações em descanso/espera de avião, faz todo o sentido que, quando um avião chega a um aeroporto, a tripulação vá para o fim da fila, descansando, entrando no avião a que aguarda há mais tempo.

3.4 Mapa Sem Ordem

O mapa sem ordem (std::unordered_map) é uma estrutura de dados proveniente da Biblioteca Padrão de Templates de C++ (STL em inglês). Ele guarda

pares de chaves e valores atribuidos à chave. É implementado como uma tabela de hash, sendo o hash calculado a partir da chave. Logo, dela decorre uma complexidade temporal constante amortizada nas operações de pesquisa, inserção e remoção. Para guardar o número de passageiros por aeroporto de partida e de destino, considerámos muito custoso o acesso individual a cada. Assim, usando esta estrutura, guardamos todas as ligações entre aeroportos e o respetivo número de passageiros com muito mais velocidade.

3.5 Fila de Prioridade Mutável

A implementação da fila de prioridade mutável foi dada pelos docentes. Consiste num vetor de apontadores para objetos (neste caso foram usados objetos Vertex), herdando todas as operações embutidas de vetores, às quais são adicionadas outros métodos. São necessárias em algoritmos como Dijkstra e A*.

3.6 Grafo

A estrutura de dados que guarda quase toda a informação do programa é um grafo. O seu único atributo é também um vetor de apontadores para vértices, contudo nele são definidos múltiplos métodos que são essenciais para praticamente todos os outros algoritmos. Os vértices são ligados por arestas, com o peso em função da distância. Cada vértice contem informação importante como o respetivo aeroporto.

4 Formalização do Problema

4.1 Dados de Entrada

- Ai - Lista de aviões que irão efetuar o transporte de passageiros, em que Ai_n representa o seu n-ésimo elemento. Cada um é identificado pelo seu id, companhia aérea.
- Ca - Capacidade dos aviões.
- Fl - Limite máximo de tempo que um avião pode viajar até esvaziar o depósito.
- Wt - Duração máxima de um turno da tripulação.
- G(V,E) - Grafo dirigido pesado, composto por:
  - V - Vértices, que representam aeroportos:
    - id - Identificador do vértice.
    - nm - Nome do aeroporto.
    - long - Coordenada longitudinal do aeroporto.
    - lat - Coordenada latitudinal do aeroporto.
    - p - Lista de rotas de cada aeroporto
    - t - Lista de tripulações, identificadas por um id.
    - adj $\subseteq$ E - Arestas que partem do vértice.
  - E - Arestas, que representam as rotas aéreas:
    - dest $\in$ E - Destino da aresta.
    - w - Peso da aresta, que representa a distância hamiltoniana entre os vértices, dado que são interpretados como pontos no globo.

4.2 Dados de Saída

Lista de registos de voos por rota em cada avião.

- Af - Rotas efetuadas por cada aeroporto.

  Cada aeroporto contém uma lista de rotas efetuadas R e cada rota contém uma lista de voos efetuados F.

  Uma rota implica que o avião saia do seu aeroporto, contendo o voo de ida para um destino predefinido. Já o voo de regresso fica guardado nas rotas do aeroporto de destino. Uma rota pode conter mais que 1 voo caso sejam necessárias escalas. Cada voo é então qualquer percurso entre 2 aeroportos, que deve ser registado com:

  - o - Aeroporto de partida.
  - d - Aeroporto de chegada.

- p - Lista de lugares do avião, guardando-se o número total de passageiros para cada destino.
- c - ID da tripulação associada ao voo.

São retornadas estatísticas acerca dos voos e rotas de modo a interpretar os resultados obtidos e a sua eficiência (são implementadas na classe **Stats**)

- Número total de passageiros transportados.
- Número de passageiros transportados que fizeram uma rota completa (do aeroportos de partida até ao de destino).
- Número de passageiros transportados entre aeroportos intermédios (em sub-rotas, entre os aeroportos de partida e de destino).
- Número total de passageiros de cada voo em que o seu aeroporto de partida e de destino corresponde ao aeroporto de partida e de destino do voo.
- Número total de passageiros de cada voo em que o seu aeroporto de partida e de destino não corresponde ao aeroporto de partida e destino do voo.
- Número médio de passageiros por voo.
- Número de passageiros processados na primeira parte do algoritmo, isto é, quando são analisados a primeira metade dos vértices ligados ao aeroporto de origem.
- Número de passageiros processados na segunda parte do algoritmo, isto é, quando são analisados a segunda metade dos vértices ligados ao aeroporto de origem.
- Número total de voos.
- Número total de rotas.
- Número total de rotas que partiram do aeroporto de destino.

4.3 Restrições

- Aos dados de entrada:
  - A capacidade dos aviões está entre 20 a 850 passageiros, para se aproximar ao contexto real de aviões comerciais:

$$\text{Ca} >= 20 \land \text{Ca} <= 850$$

  - A duração máxima de um turno da tripulação é entre 4 e 10 horas:

$$\text{Wt} > 4 \land \text{Wt} <= 10$$

  - O limite máximo de tempo que um avião pode viajar até esvaziar o depósito está entre 4 a 10 horas:

$$\text{Fl} > 4 \land \text{Fl} <= 10$$

  - O peso de cada aresta deve ser superior a zero, dado que mede distâncias:

$$\forall \text{ edge} \in \text{E, w(edge)} > 0$$

  - Arestas não percorríveis não são incluídas no grafo, bem como vértices inacessíveis e que não pertençam ao maior componente fortemente conexo.

- Aos dados de saída:
  Seja a lista de rotas de cada avião R definida como $(r_1, r_2, ..., r_{|r|})$ e a lista de voos em cada rota F como $(f_1, f_2, ..., f_{|f|})$.

  - Cada rota de um aeroporto A para um aeroporto B, implica que haja uma rota do aeroporto B para o aeroporto A.
  - O primeiro voo de cada rota tem de levar sempre pelo menos um passageiro:

$$\forall\, n \in R,\ |r_n.f_1.p| >= 1$$

  - O número de passageiros em cada voo não deve ser maior que a sua capacidade:

$$\forall\, n \in R, \forall\, m \in F,\ |r_n.f_m.p| <= Ca$$

4.4 Função Objetivo

O objetivo do problema é minimizar o tempo total de voo pela frota, o que, consequentemente, significa que a distância total percorrida e o combustível gasto serão também minimizados.

$$\sum_{a\,\in\,Ai}\ \sum_{r\,\in\,R}\ \sum_{f\,\in\,F} \text{t(f)}$$

Numa fase posterior de otimização (Etapa 3), será experimentada uma função objetivo diferente. Ao contrário da anterior, que procura o mínimo de tempo para transportar todos os passageiros, esta terá como base maximizar a quantidade de pessoas transportadas por unidade de tempo, mesmo que implique deixar algumas de fora, num dos casos (opção mais realista que proporcionaria maior lucro).

$$\frac{\sum_{a\,\in\,Ai}\ \sum_{r\,\in\,R}\ \sum_{f\,\in\,F} |p|}{\sum_{a\,\in\,Ai}\ \sum_{r\,\in\,R}\ \sum_{f\,\in\,F} \text{t(f)}}$$

5 Perspetiva de Solução

5.1 Pré-Processamento dos Dados

5.1.1 Grafo

Antes de qualquer iteração o grafo deverá ser pré-processado, isto é, ajustado às condições reais dos aeroportos. A remoção de arestas cuja influência na rota é insignificante, por exemplo, pode diminuir consideravelmente o tempo de execução dos algoritmos posteriormente.

Como mencionado anteriormente, poderão haver vários aeroportos inacessíveis devido às condições atmosféricas ou por motivos geográficos. Se o algoritmo tiver de processar as ligações para o aeroporto em questão, está a aumentar o número de iterações desnecessariamente. Logo, para maximizar a eficiência, devem ser removidos tais aeroportos e todas as rotas de e para ele.

Outra condicionante que pode influenciar a existência de arestas é a distância máxima que um avião pode voar devido aos limites de combustível. O método mais exequível de implementar esta restrição é restringindo rotas aéreas entre aeroportos cujo tempo de viagem exceda os limites da frota, introduzido pelo utilizador.

Da mesma forma que o combustível limita a distância máxima que um avião pode percorrer, também é necessário impedir que a tripulação realize demasiadas horas de voo seguidas por motivos de segurança e de ética. Visto que tanto o combustível como os turnos impedem voos demasiado longos, é possível reunir as duas condições numa restrição única, que delimita o máximo de horas seguidas que um avião pode viajar sem reabastecer e trocar a tripulação.

Implementação do Pré-Processamento do Grafo

Antes do uso dos algoritmos, foram bloqueados aeroportos e rotas devido às condições atmosféricas ou outras condicionantes como descrito previamente.

Após testar as implementações tanto do algoritmo de Tarjan como o de Kosaraju, verificamos que mais de 95% dos aeroportos pertenciam a apenas um componente fortemente conexo, mesmo tendo em conta fatores pseudo-aleatórios como o estado atmosférico. Assim, achamos que faria sentido desconsiderar estes aeroportos, visto que, ao não serem acessiveis pela grande maioria dos aeroportos, perdem utilidade e atrasam os restantes algoritmos. Deste modo, após serem identificados os componentes fortemente conexos, também é identificado o maior deles, e todos os vértices que não lhe pertençam são removidos do grafo.

Depois deste processamento, o grafo resultante é um grafo dirigido fortemente conexo, ou seja, todos os seus vértices são fortemente conexos e é possível chegar a qualquer vértice tendo qualquer outro vértice como partida.

5.1.2 Passageiros

Inicialmente itera-se sobre os passageiros de cada aeroporto para descobrir a acessibilidade de cada destino. Caso não existam caminhos possíveis para o destino de um passageiro de um aeroporto, esse passageiro pode ser posto de parte, visto que não há como o transportar.

Ainda em relação à lista de passageiros de cada aeroporto, é importante que estejam agrupados por destino, do mais longínquo para o mais curto. Assim, não só é mais rápido procurar os passageiros, como os primeiros a serem agrupados coincidem com os primeiros da lista.

Implementação do Pré-Processamento dos Passageiros
Visto que o grafo é pre-processado antes dos passageiros, é garantida de antemão a acessibilidade de cada destino de cada passageiro. Contudo, relativamente à ordenação de passageiros por distância decrescente, esse aspeto ficou por implementar visto que seria necessário calcular o caminho mais curto para todos os passageiros de todos os aeroportos e o tempo de execução seria demasiado longo e penoso.

5.2 Problemas Encontrados

O problema base de qualquer das etapas propostas é a pesquisa do caminho mais curto que permita efetuar a rota mais eficientemente, isto é, gastando menos combustível (rota com menor duração) e, numa fase posterior de optimização, maximizando o lucro.

5.2.1 Travelling Salesman Problem with Pickup and Delivery

De facto, a primeira etapa em que é considerado um único avião é apenas uma simplificação do problema e uma fase de teste dos algoritmos base implementados. A gestão de voos mesmo só com um avião, não se assemelha a problemas genéricos como o **Travelling Salesman Problem (TSP)**. Este conhecido problema procura encontrar o caminho mais curto entre uma série de pontos visitando-os apenas uma vez e regressando ao ponto inicial. Apesar de em cada rota o avião ter que voltar ao seu aeroporto de origem, o facto de poder apanhar passageiros com destinos de viagem diferentes em aeroportos intermédios, implicaria que o avião, em alguns casos, precisasse de regressar a aeroportos já visitados para satisfazer essas viagens, o que não é contemplado pelo TSP.

Assumindo a capacidade do avião como ilimitada, conseguiríamos obter um resultado iterando sobre o caminho do TSP duas vezes, a primeira onde todos os passageiros embarcavam e a posterior que os deixaria no destino requerido, mas logicamente, não seria o caminho mais rápido. Ao longo dos anos surgiram diversas propostas de resolução deste problema (**Travelling Salesman Problem with Pickup and Delivery - TSPPD**). A maioria dos algoritmos exatos são do tipo 'branch and bound' ou 'branch and cut' ou 'dynamic programming', mas a sua exequibilidade apenas se aplica a grafos com algumas

dezenas de nós, algo impensável numa escala como uma rede de aeroportos. Já as soluções heurísticas, não se desviam muito do método primeiramente mencionado, exceptuando algumas mudanças como saltos na iteração ou prioritização de grupos de vértices, que tentam otimizar a solução.

Resumindo, a natureza do problema impede o uso de soluções como os algoritmos de Held-Karp ou o Nearest Neighbor, dada a necessidade de entrada de pessoas nos aeroportos. Ou seja, a complexidade aumenta significativamente quando se trata um TSPPD. Para além disso, o custo de descolagem e aterragem ser elevado também contribui para a falta de realidade nas soluções que provêm deste tipo de algoritmos. Sendo assim, implementar uma solução para o problema acima mencionado teria pouco interesse, dada a disparidade em relação às soluções reais. A nossa primeira iteração acaba então por contemplar restrições como a capacidade, tornando o problema num **Vehicle Routing Problem (VRP)**, que por simplicidade começará apenas com uma carrinha e será aprofundado de seguida.

Implementação de Resoluções para o TSPPD
Não foi implementada uma resolução para o TSPPD, visto que seria demasiado custoso optimizar o caminho dos voos de um aeroporto tendo em conta os caminhos de todos os outros aeroportos, e a alteração de uma rota num aeroporto implicaria uma reavaliação de todos os outros, o que só aumenta o custo de uma operação que por si já era muito custosa.

5.2.2 Dial-a-Flight

Como visto anteriormente, o problema pode ser aproximado ao **Vehicle Routing Problem (VRP)**, em particular a uma das suas extensões, o **Vehicle Routing Problem with Pickup and Delivery (VRPPD)**. Pertence também à família de problemas de **Transit on Demand (TOD)**, que procuram respostas a situações como o **Dial a Ride Problem(DARP)**. Neste caso concreto, existem outro tipo de implicações, daí o problema proposto ser análogo ao **Dial a Flight Problem(DAFP)**. Há pouca documentação sobre o DAFP, visto ser um problema recente. Grande parte dos trabalhos têm como base otimizações do DARP. O facto de cada situação implicar restrições diferentes e do grau de complexidade do problema ser tão elevado não tem contríbuido para o surgimento de soluções generalizadas, mas é certo que o DAFP tem começado a receber bastante atenção nos últimos anos.

Por terem uma dificuldade acrescida, os problemas de VRPPD não têm algoritmos tão eficientes, principalmente quando a ordem de vértices começa a ultrapassar as centenas. Como mencionado na subsecção anterior, grande parte destes algoritmos envolvem 'branch and bound', 'branch and cut' e 'dynamic programming'. Dada a natureza e dimensão do Flight Net, é então necessário o recurso a uma solução heurística, com restrições diferentes de uma simples VRPPD.

Toda a formulação da heurística do problema é um desafio por si só, dado

que as aplicações no mundo real são escassas. Isto deve-se à forma como as companhias aéreas organizam as suas viagens. O que geralmente acontece, é os clientes escolherem o voo mais conveniente e não o voo a fazer o inverso. Esta situação seria concretizável num sistema de Air-Taxi, onde as pessoas pudessem requerer o destino.

Para o problema é fundamental escolher o critério de foco, que pode variar entre o transporte de todos os passageiros, a geração de lucros ou a menor rota possível em cada viagem. A escolha do algoritmo varia substancialmente em função destas opções. O retorno do avião ao aeroporto de partida é também um dos principais fatores que pode alterar a estratégia de gestão.

Em relação à janela temporal, o problema invoca todo um aglomerado de necessidades, quer no que diz respeito ao horário da tripulação, quer no simples planeamento dos voos. O agendamento de viagens traz uma complexidade ainda mais acrescida, porque o problema tornar-se-ia num **Vehicle Routing Problem with Pickup and Delivery and Time Window (VRPPDTW)**. Como o foco desta situação incide principalmente no agrupamento dos clientes e das viagens, e o concílio de horários implicaria cálculos inconcebíveis a grande escala, é necessária uma simplificação a nível cronológico e na gestão do tráfego aéreo. Apesar de tudo, a consideração pelos horários de trabalho da tripulação é fundamental e não descartável para o realismo da simulação. É então necessário impedir um avião de circular por mais de um número determinado de horas no ar, e forçar a troca de toda a tripulação sempre que haja uma paragem.

Os problemas de combustível acabam por ter especificades idênticas à da tripulação, na medida em que o avião não pode andar mais de um certo tempo no ar, obrigando o mesmo a fazer uma paragem caso não tenha o suficiente para a viagem completa. Numa situação real o consumo seria variável em função da altitude, peso, condições atmosféricas, entre outros, por isso para o problema é aproximado a uma função linear em função da distância e tempo.

Outra das grandes condicionantes num voo comercial é a satisfação de um cliente. Como foi referido em cima, os tempos de espera (fator que poderia influenciar esta matéria) não vão ser considerados. No entanto, tendo em conta que as descolagens e aterragens consomem bastante tempo, uma viagem com imensas paragens pode revelar-se problemática. Já os preços e condições do avião acabam por não ter relevância para o caso em questão.

Por fim, há ainda o entrave da inacessibilidade de aeroportos. Fatores como a meteorologia podem ser facilmente considerados, mas o mesmo não se pode dizer de influências internacionais, greves ou outro qualquer fator imprevisível que possa alterar o planeamento dos voos.

De modo geral, aplicando as particularidades do mundo real, a complexidade do problema tornar-se-ia demasiado grande para a conseguirmos resolver, porém é condição necessária a implementação de algumas destas restrições, umas que serão tratadas no pré-processamento do grafo e outras que serão aprofundadas posteriormente na proposta de solução, para obter resultados minimamente realistas.

Implementação de Resoluções para o Dial-a-Flight
Soluções concretas para este género de problema não foram propostas, tanto devido ao enorme desafio que é, como devido ao tempo disponibilizado para projetar uma proposta de solução.

5.2.3 Possiveis soluções

As diferenças entre restrições e funções objetivo dos vários problemas do tipo Dial a Flight, fazem com que não existamm propriamente soluções generalizadas. Para além disso, é um problema que só recentemente começou a receber atenção. Ou seja, há mais propostas de solução do que algoritmos efetivamente reconhecidos.

Assim, tendo em conta as especificidades do problema e as suas adaptações à vida real, decidimos projetar um método heurístico para distribuir os passageiros. A função objetivo pode tomar diferentes formas dependendo do que queremos minimizar. Inicialmente, todos os passageiros irão ser transportados ao seu destino. Ao contrário de um autocarro, por exemplo, um avião não pode parar em muitos aeroportos numa rota por duas razões. A primeira são os custos monetários e temporais de uma descolagem/aterragem, que fazem com que uma viagem curta tenha um índice de preço por distância maior. A segunda é a satisfação dos clientes, que havendo a possibilidade de fazer um voo direto, não vão aceitar parar 5 vezes por motivos de organização dos aviões.

Tendo em conta esses fatores, priorizar voos diretos e viagens longas pareceu o mais adequado. Ou seja, havendo um passageiro a querer viajar para um determinado destino, o voo opta sempre pelo caminho mais curto (direto, se possivel, ou com as escalas mais eficientes, caso não o seja). Desta forma, o nosso algoritmo optará sempre pelo caminho mais curto para o destino do passageiro. Se o caminho mais curto tiver escalas pelo meio, outras pessoas que queiram ir para esses aeroportos poderão também apanhar o avião numa das paragens, sem exceder a lotação máxima. Assim, os voos longos vão agrupando alguns passageiros de viagens curtas, reduzindo o número de aviões. Esta abordagem funcionará perfeitamente considerando que a lotação dos aviões se encontra sempre a 100%. Contudo, é nesse mesmo pormenor que o algoritmo não é tão eficiente. Se existirem poucas pessoas a querer ir para um local, ou passageiros que não obtiveram lugar noutros voos e estão agora desemparelhados, a realização dessas viagens não vai ser tão rentável.

Para colmatar estas falhas, poderão serão feitas otimizações à heurística proposta. Uma das possibilidades consiste em ignorar todos voos que não cumprissem um número determinado de lotação. Isto para uma companhia aérea seria ótimo, porque vai gerar o maior lucro por distância percorrida e será a opção mais lucrativa. Alguns passageiros ficariam sem voo, mas as viagens iriam garantir a maior eficácia possivel. Em suma, isto é negativo para um grupo de clientes. Outra opção seria agrupar os destinos menos requeridos juntos. Estas viagens seriam cansativas para os passageiros, mas transportar-se-ia toda a gente com uma certa eficiência. Resumidamente, as pessoas que não tivessem avião após a distribuição inicial, seriam agrupadas num número redu-

zido de voos. Neste caso, toda a gente seria transportada, apesar de não ser tão rentável nem satisfatório para um passageiro.

Em relação à tripulação, a cada chegada a um aeroporto é forçada uma troca. Os tripulantes que chegam são inseridos numa fila de espera onde ficam a descansar, enquanto que a tripulação com mais tempo de espera assume o lugar na continuação da rota. Os limites de horas são processados previamente no grafo. Outras condicionantes como combustível e estado atmosférico serão também consideradas.

5.2.4 Solução implementada

Após analisarmos os resultados do transporte de um avião, conseguimos ter a noção da eficiência efetiva do algoritmo. Assim, tal como planeado, partimos para a fase 2, onde introduzimos o transporte de passageiros de outros aeroportos. Este acréscimo veio aumentar em larga escala o número de pessoas transportados como veremos de seguida. Inicialmente, o nosso plano contemplava o transporte de todos os passageiros, no entanto, rapidamente percebemos que era irreal processar um número tão grande de voos, tendo em conta a complexidade temporal do mesmo. Assim, descartámos o transporte de múltiplos aeroportos, focando-nos sempre num em específico. Gostaríamos de ter também complementado o programa com algumas otimizações que potenciassem o algoritmo a ainda melhores resultados, no entanto, a falta de tempo após entrega dos dados tornou esse desafio em algo não exequível.

A implementação é idêntica ao proposto no plano original. Inicialmente é calculado o caminho mais curto usando o algoritmo de Dijktra. Depois, é gerado para cada escala, um segmento da rota(voo), ao qual é atribuido a tripulação há mais tempo no aeroporto, em deterimento da que estava antes em trabalho. Posto isto, é então feita a distribuição dos passageiros. Os prioritários são sempre os que têm destino e origem em comum com a rota. Após estes, é então feita iterativamente a distibuição(de passageiros com origem e/ou destinos diferentes) do aeroporto de origem para uma das escalas(da mais longe para a mais curta). Este processo é repetido, avançando a origem para a escala seguinte. Termina por fim, com o preenchimento total do voo, ou com o fim de todas as possibilidades.

Capacity	200	850
Total of passengers transported from the origin to the final destination	12084	114850
Total of passengers transported not from the origin to the final destination	112322	3100750
Total of passengers transported	124406	3215600
Average passengers from the origin per flight	43.2667	42.0077
Average passengers with other origin or destination per flight	87.7604	203.085
Average passengers per flight	122.781	237.224
Average passengers in the first half processing	939089	2455350
Average passengers in the second half processing	304973	760243
Total number of flights	28984	27996
Total number of routes(a group of flights)	5861	5528
Total number of routes from the source	2793	2734

Após examinar cuidadosamente os resultados, conseguimos retirar algumas conclusões. Numa fase inicial, o algoritmo é bastante eficiente, levando uma quantia elevada de passageiros por avião, à medida que os passageiros vão sendo distribuidos, há menos gente para completar os voos, o que diminui a percentagem de lotação dos aviões, como é possivel comparar entre a primeira e segunda metade do processamento.

Conseguimos também diminuir o número de voos em algumas centenas, sendo que isto se torna mais claro, quando a capacidade do avião é maior. No entanto, a taxa de ocupação de um avião maior é notoriamente menor, com uma média de 30%, opondo os 60% num avião médio.

No entanto, o maior dado que a tabela demonstra, é a quantidade de passageiros que são transportados para "preencher"os espaços que os da origem não conseguiram cobrir. Assim, é possivel concluir que o algoritmo de distribuição foi eficaz e que o lucro da empresa seria maximizado nessa componente.

6 Análise Algorítmica

De modo a tentar optimizar o problema e a encontrar a sua solução ideal, vamos explorar alguns algoritmos que pretendem resolver as especificidades analisadas anteriormente.

6.1 Pesquisa

Algoritmos de pesquisa de grafos permitem reconhecer todos os nós descendentes de um nó de origem, sendo particularmanente utéis no seu pré-processamento e na análise da conectividade.

6.1.1 Algoritmo de Pesquisa em Profundidade

DFS (Deapth-First Search) é uma estratégia de pesquisa de grafos que consiste em explorar as suas arestas de em profundidade, a partir de um vértice de origem que pertence necessariamente ao grafo. Sempre que possível, analisa todas as arestas do vértice descoberto mais recentemente e, quando todas elas já tenham sido exploradas, retrocede de modo a pesquisar as arestas não visitadas do vértice anterior. O processo é repetido até que todos os vértices acessivéis a partir do vértice de origem tenham sido encontrados.

Pseudo-Código

Pesquisa em Profundidade

```
1: function DFS(Graph G(V, E), Vertex v)
2:     visited(v) ← true
3:     for Vertex w ∈ ADJ (G, v) do
4:         if !visited(w) then DFS(G, w)
```

Complexidade
A complexidade temporal do algoritmo é $O(|E| + |V|)$ porque esta estratégia garante que cada aresta é encontrada apenas uma vez, assumindo o uso de uma lista de adjacências e que o membro *visited* é inicializado com $false$ em cada vértice.

A complexidade espacial do algoritmo é correspondente ao número de chamadas recursivas, que são no máximo o número total de vértices, logo é $O(|V|)$.

Implementação
O DFS e várias outras variações dele foram implementados no projeto, sendo usado em vários outros algoritmos como o Kosaraju e o Tarjan.

6.2 Caminho mais curto

Algoritmos de caminho mais curto procuram encontrar a menor distância entre um dado número de vértices num grafo, de acordo com o respetivo peso das suas arestas. Esse processo é a base do nosso problema, interessando conhecer sempre o caminho mais curto entre 2 aeroportos ou entre um aeroporto e todos os outros, de modo a minimizar a distância percorrida que, consequentemente, minimiza o tempo gasto, intrínsecamente ligado ao combustível utilizado em cada rota.

6.2.1 Algoritmo de Dijkstra

Dado um vértice inicial de um grafo G(V,E) com arestas de peso não-negativo, o algoritmo de Dijkstra recorre a uma abordagem gananciosa para resolver o problema do caminho mais curto entre esse vértice e todos os outros. Também pode ser usado para encontrar o caminho mais curto entre 2 vértices, parando o processamento quando esse caminho é encontrado.

A ideia principal do algoritmo consiste em guardar um conjunto de vértices por visitar e a partir de um vértice inicial s com distância 0, marcá-lo como vértice atual e calcular as possíveis distâncias entre os seus vértices adjacentes, somando o peso de cada aresta que os conecta. O vértice é marcado como alcançado e removido do conjunto de vértices por visitar, passando o vértice atual a ser o que corresponde à aresta de menor distãncia. O processo é repetido até a distância mínima ser infinita, caso se pretenda encontrar a distância entre um vértice de origem e todos os outros vértices, ou até que o vértice de destino seja marcado como visitado, caso se pretenda encontrar a distância entre 2 vértices específicos. Neste último caso, bastaria parar o algoritmo no momento em que o vértice de destino correspondesse ao vértice de menor distância.

O pseudo-código seguinte corresponde ao Dijkstra de um ponto para vários, estando destacada a condição de paragem para o caso do Dijkstra entre 2 pontos.

Pseudo-Código

Algoritmo de Dijkstra

```
1:  function DIJKSTRA(Graph G(V, E), Vertex s)
2:      Q ← ∅
3:      for Vertex v ∈ G do
4:          dist(v) ← ∞
5:          path(v) ← NULL
6:          visited(v) ← false
7:          queueIndex(v) ← 0
8:      dist(s) ← 0
9:      INSERT(Q, (s))
10:     while Q ≠ ∅ do
11:         Vertex v ← EXTRACT-MIN(Q)
12:         visited(v) ← false
13:         if *Between two points* v == destinationVertex then
14:             break
15:         for Edge e ∈ ADJ(v) do
16:             Vertex* w = destination(e)
17:             length ← dist(v) + weight(e)
18:             oldDist = dist(w)
19:             if oldDist > length then
20:                 dist(w) ← length
21:                 path(w) ← v
22:                 if oldDist == INF then
23:                     Vertex w ← INSERT(Q)
24:                 else DECREASE-PRIORITY(Q, (w))
```

O uso de uma fila de prioridade de mínimo em vez de um set, apesar de não fazer parte do algoritmo original proposto por Dijkstra, garante a ordem de progressão pretendida e que o vértice com a distância mínima tem maior prioridade. As operações da fila de prioridade de INSERT, EXTRACT-MIN e DECREASE-PRIORITY, realizam, respetivamente, a inserção de um elemento, remoção do elemento da cabeça da fila e a diminuição de prioridade.

Complexidade

Quanto à complexidade temporal, as operações de inserção e remoção de elementos da fila ocorrem $|V|$ vezes e podem ser realizadas em tempo logarítmico sobre o tamanho da fila $|V|$, logo temos $O(|V| * log(|V|))$. Já a operação de diminuição de prioridade implica que seja feita no máximo $|E|$ vezes e pode ser realizada em tempo logarítmico sobre o tamanho da fila $|V|$, logo temos $O(|E| * log(|V|))$. Assim, a complexidade temporal do algoritmo será $O((|E| + |V|) * log|V|)$. No entanto, caso seja usada uma Fibonacci Heap essa complexidade poderá ser me-

lhorada para $O(|V| * log|V|)$ dado que a operação de diminuição de prioridade ocorre em tempo constante $O(1)$.

A complexidade espacial do algoritmo é $O(|V|)$ dado que a memória ocupada pela fila de prioridade tem um tamanho máximo equivalente ao número total de vértices do grafo.

Implementação
O algoritmo de Dijkstra para a distância entre 2 pontos foi o principal algoritmo para calcular o caminho mais curto do programa. Em comparação com outros e a par do A-Star, foi o que se provou mais rápido e consistente. Assim, é o utilizado na distribuição dos passageiros por rotas. Tendo em conta a grande importância do cálculo do caminho mais curto na nossa função heurística, o algoritmo de Dijkstra é então uma pedra basilar do programa. Os resultados obtidos(apresentados no fim da secção) corroboram a análise teórica realizada. Foi considerado e implementado o Dijkstra de um para vários pontos na heurística, mas como era necessária a variação do aeroporto de origem para idas e voltas, o algoritmo perderia a sua utilidade. No entanto, não fosse essa particularidade, seria muito mais eficaz, pois evitaria o processamento do Dijkstra a cada iteração.

6.2.2 Algoritmo A*

O algoritmo A* pode ser interpretado como uma extensão ao algoritmo de Dijkstra, usando heurísticas para otimizar os seus resultados. A cada iteração, o A* escolhe o vértice para qual se extender a partir do valor de $f(v) = h(v)+g(v)$, em que $h(v)$ representa uma função heurística que determina o caminho mais curto e $g(v)$ a distância desde o vértice original até ao vértice v. O algoritmo termina quando o vértice final é encontrado ou quando não há mais caminhos a serem extendidos.

Apesar de visitar consideravelmente menos vértices que o Dijkstra, devido a se basear em heurísticas, a solução encontrada pode não ser a solução ótima. No caso de $h(v)$ representar a distância hamiltoniana (no globo) entre os vértices, equivale a aplicar o algoritmo de Dijkstra mas com os pesos das arestas modificados, escolhendo sempre o vértice com a menor distância hamiltoniana e o menor peso por aresta. Isto garante que se escolhem caminhos cada vez mais perto do vértice final, representando então uma melhoria em relação ao algoritmo de Dijkstra. Sendo o nosso grafo pesado tendo em conta a distância, este algoritmo torna-se pertinente na nossa solução.

Pseudo-Código

Algoritmo A*

```
1:  function ASTAR(Graph G(V, E), Vertex s, Vertex f)
2:      Q ← ∅
3:      for Vertex v ∈ G do
4:          dist(v) ← ∞
5:          path(v) ← NULL
6:          visited(v) ← false
7:          queueIndex(v) ← 0
8:      dist(s) ← Heuristic(s, v)
9:      gScore(s) ← 0
10:     INSERT(Q, (s))
11:     while Q ≠ ∅ do
12:         Vertex v ← EXTRACT-MIN(Q)
13:         visited(v) ← false
14:         if v == f then
15:             break
16:         for Edge e ∈ ADJ(v) do
17:             Vertex* w = destination(e)
18:             if visited(w) == true then
19:                 continue
20:             tentativeGScore ← GScore(v) + weight(e)
21:             if dist(w) == NULL then
22:                 INSERT(Q, (v))
23:             if tentativeGscore < GScore(w) then
24:                 path(w) ← v
25:                 GScore(w) ← tentativeGScore
26:                 dist(w) ← GScore(w) + Heuristic(w, f)
27:                  DECREASE-PRIORITY(Q, (w))
```

EXTRACT-MIN remove o elemento da cabeça da fila, que corresponde ao elemento com menor *hdist*.

Complexidade

A complexidade temporal do algoritmo A* depende da heurística utilizada, sendo que, no pior caso, é de $O(|V| + |E|)$, dado que tem de percorrer todos os vértices e arestas do grafo para encontrar o caminho mais curto. Contudo, assumindo uma heurística em que $\forall v \in V, h(v) = 0$, o algoritmo é semelhante ao de Dijsktra, logo a sua complexidade temporal e espacial também o serão.

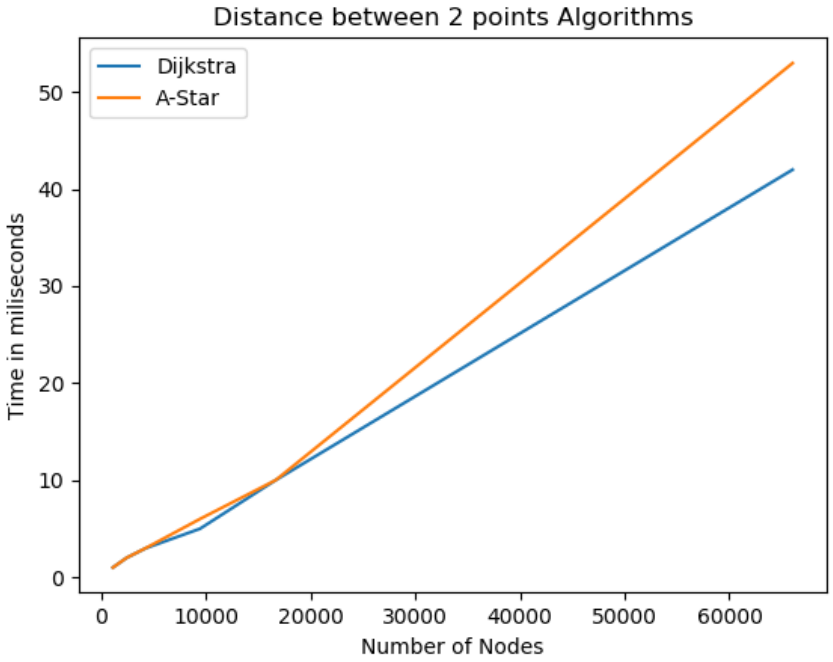
Considerando uma heurística baseada na distância hamiltoniana, temos que o número de vértices a serem percorridos irão sempre representar a distância no globo entre cada vértice e o vértice final, pelo que no máximo são percorridos

$\sqrt{|V|}$ vezes. Assim, a complexidade temporal do algoritmo é $O(\sqrt{|V|} * log(|V|))$.

A complexidade espacial continua a ser $O(|V|)$, sendo que é usada uma fila de prioridade com tamanho máximo equivalente a todos os vértices do grafo.

Implementação

À semelhança do algoritmo de Dijkstra, este também foi dos mais eficientes. Acabámos por optar pelo Dijkstra na solução heurística devido a pequenas diferenças, no entanto, também é possivel obter um caminho com este método no programa. Há ligeiro afastamento sobre o Dijkstra quando falamos da sua aplicação empírica, mas também acabou por estar dentro do modo previsto.



O cálculo do tempo em função do número de nodes a processar na procura do caminho entre 2 pontos acaba por ser idêntico nos dois algoritmos. A linearidade das funções confirma-se apesar da ligeira diferença das regressões. Assim, o algoritmo principal esolhido acabou por ser o de Dijkstra.

6.2.3 Algoritmos Bidirecionais

Em casos de pesquisa do caminho mais curto entre 2 vértices, algoritmos Bidirecionais constituem uma otimização interessante face aos de pesquisa unidirecional, como o Dijkstra ou o A*.

O algoritmo de Dijkstra bidirecional parte da ideia base de aplicar o algoritmo de Dijkstra em ambos os vértices inicial e final, parando quando as duas

pesquisas se encontram num ponto intermédio, isto é, quando um determinado vértice já foi analisado na outra direção. No entanto, devido a esta condição de paragem, o resultado pode não ser o optimal, dado que o vértice processado por ambas as pesquisas pode não representar o caminho mais curto, apenas o vértice que foi processado por ambas primeiro. Considerando uma condição de paragem mais forte, como por exemplo, se terminar quando a soma dos valores do topo das filas de prioridade de cada pesquisa (top_f para $forward$ e top_r para $reverse$) for superior à menor distância já percorrida ($path$), conseguimos garantir a escolha do caminho mais curto na região de encontro dos 2 algoritmos:

$$top_f + top_r >= path$$

O algoritmo assume a existência de um grafo transposto ao grafo original onde se pretende fazer a pesquisa, como dado de entrada R. fP e bP são sets que guardam os vértices processados. midPoint é o vértice do meio do grafo, onde as pesquisas se encontram.

Pseudo-Código

Algoritmo de Dijkstra Bidirecional

```
function DIJKSTRA(Graph G(V, E), Vertex s, Vertex f, Graph R(V, E))
    fQ ← ∅, bQ ← ∅
    fP ← ∅, bP ← ∅
    mu ← NULL
    for Vertex v ∈ G do
        dist(v) ← ∞
        path(v) ← NULL
        visited(v) ← false
        queueIndex(v) ← 0
    for Vertex v ∈ R do
        dist(v) ← ∞
        path(v) ← NULL
        visited(v) ← false
        queueIndex(v) ← 0
    dist(s) ← 0
    INSERT(fQ, (s))
    INSERT(bQ, (f))
    while Q ≠ ∅ ∧ R ≠ ∅ do
        Vertex vf ← EXTRACT-MIN(fQ)
        Vertex vb ← EXTRACT-MIN(bQ)
        insert(fP)
        insert(bP)
        if dist(vf) + dist(vb) >= mu then
            break
        for Edge e ∈ ADJ(vf) do
            Vertex* w = destination(e)
            length ← dist(vf) + weight(e)
            oldDist = dist(w)
            if oldDist > length then
                dist(w) ← length
                path(w) ← vf
                if oldDist == INF then
                    Vertex w ← INSERT(fQ)
                else DECREASE-PRIORITY(fQ, (w))
            revW ← find(R, w)
            if (find(bP, revW) == true) ∧ (length + dist(revW) < mu) then
                mu ← length + dist(revW)
                midPoint ← w
        for Edge e ∈ ADJ(vb) do
            Vertex* w = destination(e)
            length ← dist(vb) + weight(e)
            oldDist = dist(w)
            if oldDist > length then
                dist(w) ← length
                path(w) ← vb
                if oldDist == INF then
                    Vertex w ← INSERT(bQ)
                else DECREASE-PRIORITY(bQ, (w))
            forW ← find(G, w)
            if (find(fP, forW) == true) ∧ (length + dist(forW) < mu) then
                mu ← length + dist(forW)
                midPoint ← w
```

Já o algoritmo de A*, seguindo o mesmo procedimento de o aplicar nos

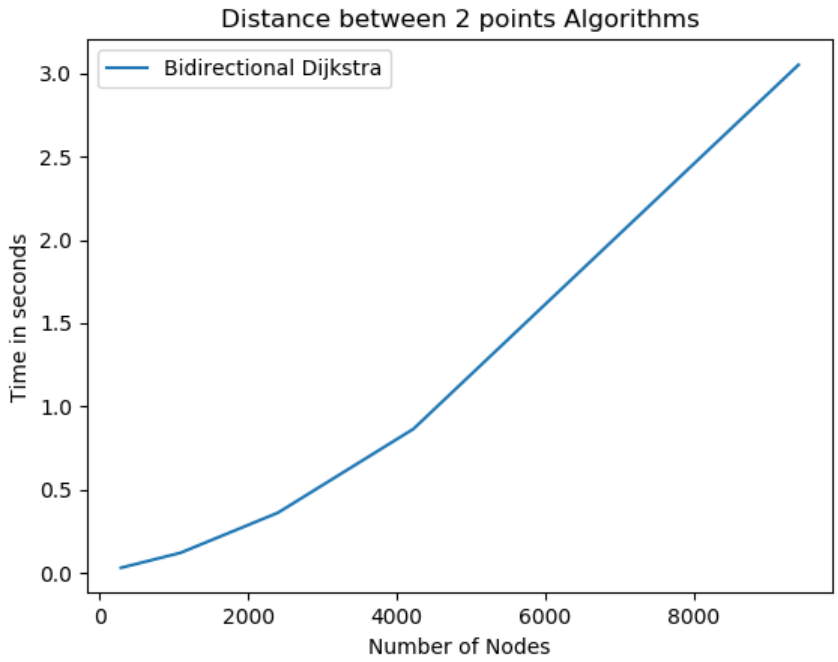
vértices inicial e final, não é tão simples quanto aparenta ser. Caso a consistência da função potencial não seja garantida ($h_f + h_r = const$, em que h_f representa a função potencial no sentido $forward$ e em que h_r representa a função potencial no sentido $reverse$), os dois algoritmos podem operar sobre diferentes funções de distância, ou seja, quando as pesquisas se encontram, o caminho mais curto não é garantido. De modo a ultrapassar esse incómodo, uma alternativa será usar funções potenciais consistentes, que representam as funções média das funções potenciais arbitradas:

$$p_f = \frac{h_f - h_r}{2} \qquad p_r = \frac{h_r - h_f}{2} = -p_f$$

Assim, a condição de paragem deverá verificar se a soma dos valores do topo das filas de prioridade de cada pesquisa é superior à soma da menor distância já percorrida com a função potencial média p_r, assumindo que $p_f = 0$

$$top_f + top_r >= path + p_r$$

Implementação
Apenas o Dijkstra Bidirecional foi efetivamente implementado, ficando o A* Bidirecional por implementar, dado que percebemos à partida que não seria um algoritmo tão eficiente. O cálculo do tempo é substancialmente mais elevado, numa ordem completamente diferente dos normais.



Como é possivel reparar no gráfico, este algoritmo tem um custo temporal de alguns segundos, o que em comparação com os mencionados anteriormente, é extremamente elevado, elimando qualquer hipótese de ser usado na função heurística. .

6.2.4 Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall procura encontrar o caminho mais curto entre todos os pares de vértices, usando uma abordagem de programação dinâmica, sendo mais útil em casos de grafos densos e tendo em conta casos de arestas com pesos negativos. A sua implementação poderá ser bastante pertinente em fase de pré processamento de grafos.

Dado um sub-conjunto de V em que $\{1, 2, ..., k\}$, para qualquer par de vértices $(i, j) \in V$, consideram-se todos os caminhos de i para j com vértices intermédios em $\{1, 2, ..., k\}$, sendo p o caminho simples com menor peso de todos eles. O algoritmo de Floyd-Warshall explora a relação entre o caminho p e os caminhos mais curtos de i para j, pertencentes ao sub-conjunto $\{1, 2, ..., k-1\}$. Definindo recursivamente o peso do caminho mais curto entre i e j com todos os vértices intermédios no conjunto $\{1, 2, ..., k\}$ como $d_{ij}^{(k)}$, temos então:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & if\ k = 0, \\ \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right) & if\ k \geq 1. \end{cases} \quad (1)$$

Quando $k = 0$, não existem vértices intermédios pelo que a distância entre i e j será o peso da aresta que os conecta; quando $k \geq 1$, existe pelo menos um vértice intermédio entre os pontos pelo que a distância mais curta será o valor mínimo entre a distância entre i e j e a soma das distâncias dos sub-caminhos entre i e j que passam por k. Aplicando esse processo recursivamente, é obtida uma matriz de distâncias mínimas entre todos os pares de vértices. O algoritmo pode ser modificado de modo a guardar os vértices que constituem esses caminhos.

Pseudo-Código

Algoritmo de Floyd-Warshall

```
1: d : minimum distance, matrix |V| x |V| initialized with ∞
2: p : predecessor in path, matrix |V| x |V| initialized with NULL
3:
4: function FloydWarshall(Graph G(V, E, w))
5:     for Edge (i, j) ∈ E do
6:         d_ij ← w_ij
7:         p_ij ← i
8:     for Vertex j ∈ V do
9:         d_jj ← 0
10:    for k from 1 to |V| do
11:        for i from 1 to |V| do
12:            for j from 1 to |V| do
13:                length ← d_ik + d_kj
14:                if d_ij > length then
15:                    d_ij ← length
16:                    p_ij ← p_kj
```

Complexidade
A complexidade temporal do algoritmo é $O(|V|^3)$ dado que todas as operações têm complexidade constante e contém 3 ciclos for agrupados, precisos para comparar cada par de vértices com os correspondentes caminhos do grafo.

Em termos de complexidade espacial, o algoritmo requer $O(|V|^3)$ caso se criem k matrizes. Essa complexidade pode ser reduzida até $O(|V|)$ se se atualizar a matriz em cada iteração em vez de se criar uma nova.

Implementação
Como a distribuição dos voos dizia respeito a apenas 1 dos aeroportos, o algoritmo de Floyd-Warshall acabaria por não ser tão eficiente, pois iriam ser

processados vários caminhos que nunca seriam usados.

6.3 Conectividade

A análise da conectividade de um grafo consiste em verificar se a partir de qualquer vértice é possível chegar a qualquer outro vértice. Assim, no grafo pré-processado podem existir sub-grafos definidos como um conjunto de vértices conexos sem nenhum caminho para vértices de outros sub-grafos, ou seja, grafos independentes dentro do grafo pré-processado.

Deste modo, a análise da conectividade é pertinente para o pré-processamento do grafo, dado que é necessário remover aeroportos inacessíveis. A acessibilidade de um aeroporto reflete se é possível ou não estabelecer rotas entre ele e pelo menos um outro aeroporto da rede. Por outras palavras, é necessário saber se na rede de aeroportos existem vários sub-grafos independentes, e caso existam, remover todos os sub-grafos com apenas um vértice, ou seja, remover todos os aeroportos sem ligações.

Para além de remover todos os aeroportos inacessiveis, este pré-processamento também permite computar em tempo constante se é possivel uma rota entre dois aeroportos, visto que basta verificar se ambos são do mesmo sub-grafo. Assim, melhora-se muitíssimo a performance do cálculo da melhor rota entre dois aeroportos caso estes não estejam ligados, o que outrora seria um processo muito custoso.

Se o grafo fosse dirigido, poderia ser implementado o algoritmo de Tarjan ou de Kosaraju. Porém, no contexto deste projeto, faz mais sentido um grafo não dirigido, devido à natureza bilateral das rotas aéreas. Quando comparado com casos reais, o mais comum é o factor que impossibilita uma rota também impossibilitar a rota inversa. Assim, para tornar a decisão uniforme, foi decidido assumir que qualquer rota ou impedimento dela seria bilateral, e, consequentemente, foi decidido usar um grafo não dirigido.

Estando impossibilitado de usar o algoritmo de Tarjan ou de Kosaraju, foi necessário usar um algoritmo mais genérico (neste caso o depth first search, DFS) e modificá-lo para alcançar o pretendido.

Implementação da Conectividade

Para o âmbito do projeto, e tendo em conta a natureza das viagens aéreas, inicialmente decidimos que um grafo não dirigido fortemente conexo seria o ideal, dado que um aeroporto tem sempre voos em 2 sentidos. Contudo, o grafo fornecido nem era dirigido nem era fortemente conexo. Caso fosse não dirigido, bastaria aplicar DFS num vértice (aeroporto) ainda não visitado, e agrupar todos os vértices visitados por este DFS, até que todos os vértices do grafo tenham sido visitados. No entanto, o grafo fornecido é dirigido, e foi necessário reformular todo o código já feito, apesar do tempo escasso. Com estas alterações implementamos então algoritmos que identificassem os componentes fortemente conexos de um grafo dirigido, como o de Tarjan ou o de Kosaraju.

Outras divergências do pleanemento da conectividade residem na seleção de aeroportos para remover. Inicialmente, foi projetado que apenas aeroportos

sem rotas seriam removidos, porém esse decisão foi repensada e foi determinado que qualquer aeroporto que não pertencesse ao componente fortemente conexos mais populoso devia ser removido. Esta mudança tornou obsoleta a questão de saber se uma rota entre dois aeroportos é possivel (que também foi debatido no planeamento da conectividade), visto que é sempre possivel, caso contrário não podiam estar no mesmo componente fortemente conexo.

6.3.1 Algoritmo de Kosaraju

O algoritmo de Kosaraju descobre todos os componentes fortemente conexos de uma grafo ao percorrer todos os seus vértices com um DFS, depois ao reverter todas as arestas do grafo e de seguida ao voltar a percorre-lo, desta vez pela ordem contrária. Resumidamente:

- Percorre o grafo com DFS, começando num vértice aleatório e adicionando cada vértice a uma pilha quando todas as suas arestas forem percorridas.
- Reverte todas as arestas do grafo.
- Percorre o grafo revertido com DFS, começando pelo vértice que se situa no topo da pilha. Atribui a todos os vértices alcançados por este DFS o mesmo identificador de SCC, e marcá-los como visitados. Repete este processo até a pilha esvaziar.

Pseudo-Código

Algoritmo de Kosaraju

```
1:  S = 0
2:  L = Stack()
3:  function KOSARAJU(G(V, E))
4:      for v ∈ V do
5:          SCC(v) ← NULL
6:      for v ∈ G do
7:          DFS_S(G, v)
8:      while !L.EMPTY do
9:          v ← L.pop()
10:         DFS(G, v, v)
11:
12: function DFS_S(G(V, E), v)
13:     if u ∈ S then
14:         return
15:     for u ∈ ADJ(G, v) do
16:         DFS_S(G, u)
17:     L.push(v)
18:
19: function DFS(G(V, E), v, raiz)
20:     if SCC(v) ≠ NULL then
21:         return
22:     SCC(v) ← raiz
23:     for u ∈ ADJ(G, v) do
24:         DFS(G, u, raiz)
```

Complexidade
Como o grafo revertido já é computado previamente, apenas entra no calculo da complexidade os dois DFS's executados, cuja complexidade temporal é $O(|V| + |E|)$.

Implementação
O algoritmo de Kosaraju foi implementando, sendo alternativa ao algoritmo de Tarjan.

6.3.2 Algoritmo de Tarjan

O algoritmo de Tarjan também identifica o componente fortemente conexo de cada vértice de um grafo. Resumidamente:

- Percorrer o grafo com DFS, começando num vértice aleatório. Ao passar por cada vértice, é lhe atribuido dois valores iguais e únicos incrementados sequencialmente, o id e o valorBaixo,(ou seja, ao primeiro vértice é atribuido dois números n, ao seguinte dois $n+1$, etc.), é marcado como visitado e é adicionado a uma pilha.
- Quando ocorre retrocesso no DFS, se o vértice anterior está incluido na pilha, atualizar o valorBaixo do vértice atual para o valorBaixo do anterior caso seja menor.
- Após percorrer todas as suas arestas, se o seu id é igual ao seu valorBaixo, remover arestas da pilha até remover o vértice atual.

Pseudo-Código

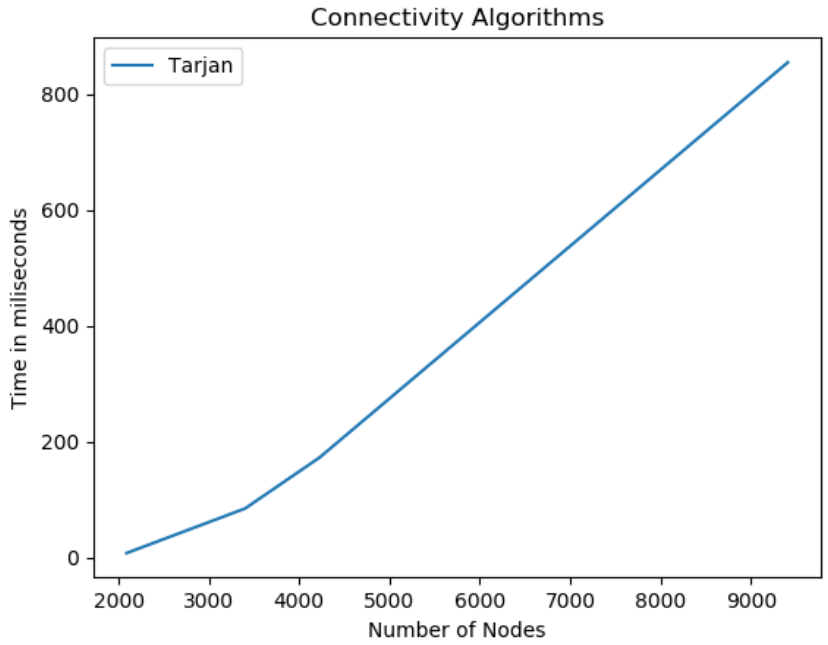
Algoritmo de Tarjan

```
1:  numberid ← 1
2:  L ← Stack()
3:
4:  function TARJAN(G(V, E)))
5:      for v ∈ V do
6:          id(v) ← NULL
7:          SCC(v) ← NULL
8:      for v ∈ V do
9:          if id(v) ≠ NULL then
10:             DFS_T(G, v)
11:
12: function DFS_T(G(V, E), v)
13:     L.push(v)
14:     low(v) ← numberid + +
15:     id(v) ← low(v)
16:     for u ∈ ADJ(G, v) do
17:         if id(u) = NULL then
18:             DFS_T(G, u)
19:             low(v) ← min{low(v), low(u)}
20:         else if v ∈ L then
21:             low(v) ← min{low(v), id(u)}
22:     if low(v) = id(v) then
23:         while (u ← L.POP()) ≠ v do
24:             SCC(u) ← v
25:         SCC(v) ← v
```

Complexidade

Também este algoritmo tem complexidade temporal de $O(|V| + |E|)$, visto que se baseia num DFS geral.

Implementação
A opção que tomámos para processar os nodes da componente fortemente conexa, acabou por ser o Tarjan. Após compararmos os dois algoritmos, o Tarjan revelou ser ligeiramente mais eficiente, para além de ser bastante mais organizado e fácil de examinar. Assim, é este o algoritmo que usamos para processar os nodes não acessíveis da SCC aquando do recurso à função heurística.



6.4 Heurísticas

Como analisamos anteriormente, tendo em conta que soluções exatas para problemas do tipo VRP são bastante custosas e muitas vezes de elevada complexidade de implementação devido a restrições e condições de processamento impostas, recorrer a heurísticas que aproximam boas soluções é um dos métodos mais eficientes na procura de respostas para este tipo de problemas. Heurísticas de construção 'clássicas' constroem rapidamente soluções viáveis passivéis de serem melhoradas através de métodos de pesquisa locais, como *Tabu Search* e *Simulated Annealing*, ou populacionais, como *Genetic Search*.

Para além da solução descrita previamente é relevante mencionar por exemplo o algoritmo de Clarke e Wright. Apesar deste algoritmo nem sempre retornar a solução ótima, oferece boas alternativas. Neste caso, após certas modificações,

o seu uso poderá ornar-se vantajoso como otimização à proposta de solução que concebemos, dada a sua adaptabilidade. 5.2.3.

6.4.1 Algoritmo de Clarke e Wright

Partindo de um grafo $G(V, E)$, o algoritmo de Clarke e Wright (também conhecido como *Savings Algorithm*) procura encontrar soluções viáveis para o VRP, baseando-se na poupança de custos associados ao peso das arestas do grafo, através da criação de rotas posteriormente otimizadas pela junção de caminhos que envolvam diminuição desses mesmos custos.

Dado um conjunto de pontos de interesse, calcula-se a poupança s_{ij} para cada par desses pontos, que representa a diminuição do custo obtida caso seja efetuada uma rota entre i e j em vez de 2 rotas individuais para cada ponto. Sendo D o ponto de partida e d o custo/distância entre 2 pontos temos que:

$$s_{ij} = d_{Di} + d_{Dj} - d_{ij}$$

Após o cálculo das poupanças, esses valores devem ser ordenados de forma decrescente e os pares de pontos são considerados por essa ordem um de cada vez. A rota entre esses pontos é unida se essa junção não implicar eliminar uma conexão efetuada anteriormente e se a capacidade total dos veículos não for violada.

O algoritmo tem ainda duas versões, uma paralela e uma sequencial. A paralela implica uma análise das poupanças única criando diferentes rotas de uma só vez; já a sequencial cria uma rota de cada vez que analisa as poupanças calculadas. Em geral, a versão paralela apresenta melhores resultados que a versão sequencial, dependendo sempre, no entanto, do caso particular a considerar.

7 Conclusões preliminares

De modo geral, o objetivo de otimizar os voos de forma a obter uma solução que conseguisse ser em simultâneo realista e eficiente foi cumprido. No entanto, a gestão das rotas provou ser um problema de complexidade muito mais elevada do que a imaginada inicialmente, o que nos forçou a realizar um enorme trabalho do pesquisa.

O grande problema ao qual procurámos responder foi o **Vehicle Routing Problem with Pickup and Deliveries - VRRPD**, no entanto, devido às suas especificidades, foi aplicado na extensão **Dial a Flight - DAF**.

Na edificação da solução foram explorados alguns algoritmos estudados nas aulas, como por exemplo o Dijkstra ou o Floyd-Warshall, dada a necessidade de calcular as rotas mais rápidas. Porém, grande parte dos obstáculos que advêm do problema foram pensados com base em artigos e algoritmos sobre o tema. Neste grupo incluem-se por exemplo o Clarke e Wright ou os algoritmos heurísticos projetadas.

A implementação está planeada em fases graduais, sendo que as últimas consistem em otimizações complementares às imperfeições dos algoritmos iniciais. Nestas etapas são usados conceitos como força bruta, algoritmos gananciosos, divisão e conquista, programação dinâmica, recursividade e retrocesso.

Apesar das dificuldades que o problema trouxe ao de cima e de não haver muito material de estudo sobre o tema, consideramos que chegámos a uma solução bastante eficaz, dadas as restrições que um sistema de gestão de voos implica. A redação do relatório foi fundamental para a organização de ideias e algoritmos e deverá facilitar a concretização da nossa solução, algo que vamos poder avaliar ao por o projeto em prática.

7.1 Contribuições

Ao longo do desenvolvimento do trabalho proposto para esta fase, adotamos um método colaborativo e de discussão de ideias de modo a encontrarmos as melhores soluções e ultrapassarmos as dificuldades que foram surgindo, pelo que a contribuição de cada elemento foi equilibrada entre todos. Sendo assim, como todos contribuímos em cada parte do trabalho, a lista de tarefas que cada um executou é partilhada e corresponde à integridade do relatório.

8 Conclusão

Tal como imaginávamos, grande parte dos nossos receios iniciais acabaram por se confirmar. A definição de particularidades como a fila de tripulações e a geração de passageiros para todos os aeroportos, que tinham como objetivo tornar a simulação o mais realista possivel, acabaram por ter bastante peso na formação do grafo, povoamento das estruturas de dados e na heurística sem si. Assim, para considerar todas as variáveis no processamento dos voos, tivemos de recorrer a demasiadas componentes do programa. O sucesso obtido acabou por contrastar com o gasto temporal e espacial.

Apesar dos obstáculos, conseguimos implementar uma grande variedade de algoritmos. Para cada necessidade, procurámos ter em conta as diversas opções, fosse com o objetivo de avaliar a conectividade do gráfico ou a calcular o melhor caminho. Experimentámos e comparámos as eficiências de cada, de modo a optar pelo que melhor se adaptava ao nosso problema.

Como mencionado previamente, as rotas no FlightNet atingem a sua eficiência máxima quando vão pelo caminho mais curto. Assim, formulámos a parte gráfica com a capacidade de destacar essa rota. Para além disso, considerámos que os aeroportos interditos devido ao mau tempo ou à sua inacessibilidade deveriam estar também marcados.

Em relação aos dados de entrada, foi penosa a entrega em atraso do formato do grafo. As alterações tardias à estrutura do grafo impediram que quer a função heurística, quer o GraphViewer fossem mais aprofundados, prejudicando o resultado final.

De modo geral, conseguimos responder aos problemas que o Dial of Flight impõe. Implementamos as restrições propostas inicialmente com sucesso, com exceção das situações mais complexas descritas anteriormente. A eficiência da heurística também se confirmou, provando ser bastante eficaz inicialmente, perdendo rendimento nos últimos vértices processados. Foi também implementada gradualmente, como previsto, no entanto, sem as otimizações que gostariamos de ter experimentado.

A complexidade elevada do problema e a necessidade de trabalhar com imensas informações ao mesmo tempo fez com que este projeto fosse extremamente desafiante em comparação com outros temas, algo que aumentou exponencialmente o nosso interesse e vontade em pesquisar e implementar a solução. No entanto, também acabou por nos limitar em diversas particularidades restritas do tema, que não teriamos de considerar outrora. Apesar de tudo, consideramos que fizemos o melhor possivel e que alcançámos um resultado bastante satisfatório. Podemos também dizer que o nosso conhecimento sobre algoritmia aumentou significativamente após semanas de pesquisa e de trabalho e fica uma vontade de explorar mais problemas avançados!

9 Referências

T. Cormen, C. Leiserson, R. Rivest e C. Stein. Introduction to Algorithms. Cambridge, MA: MIT Press, 2009.

M. A. Weiss. Data Structures and Algorithm Analysis in C++, 3/E. New York, NY: Addison Wesley, 2007.

J. Cordeau, G. Laporte, J. Potvin e M. Savelsbergh. Transportation On Demand, 2004. Disponível: https://www2.isye.gatech.edu/~ms79/publications/TransOnDemand.pdf

M. Nowak, The Pickup and Delivery Problem with Split Loads, Engenharia Industrial e de Sistemas, Georgia Institute of Technology, 2005. Disponível: https://smartech.gatech.edu/bitstream/handle/1853/7223/nowak_maciek_a_200508_phd.pdf?sequence=1&isAllowed=y

I. Dumitrescu, S. Røpke, J. Cordeau, e G. Laporte. The Traveling Salesman Problem with Pickup and Delivery: Polyhedral Results and a Branch-and-Cut Algorithm. Mathematical Programming, 121(4), 269-305, 2010. Disponível: https://doi.org/10.1007/s10107-008-0234-9

P. Hart, N. Nilsson e B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, 1968. Disponível: https://ieeexplore.ieee.org/document/4082128

V. Cinar, T. Öncan e H. Süral. A Genetic Algorithm for the Traveling Salesman Problem with Pickup and Delivery Using Depot Removal and Insertion Moves. pp. 431-440, 2010. Disponível: https://link.springer.com/chapter/10.1007%2F978-3-642-12242-2_44

J. Cordeau, M. Iori, N. Mendes, D. Nelli e R. Tedeschi. Minimizing User Inconvenience and Operational Costs in a Dial-a-Flight Problem for Air Safaris. 2020. Disponível: https://chairelogistique.hec.ca/wp-content/uploads/2020/10/DAFP.pdf

S. Yanik, B. Bozkaya e R. deKervenoael. A new VRPPD model and a hybrid heuristic solution approach for e-tailing, European Journal of Operational Research, vol. 236, no. 3, pp. 879-890, 2014. Disponível: http://dx.doi.org/10.1016/j.ejor.2013.05.023

G. Martinovic, I. Aleksi e A. Baumgartner. Single-Commodity Vehicle Routing Problem with Pickup and Delivery Service, Mathematical Problems in Engineering, vol. 2008, pp. 1-17, 2008. Disponível: https://www.hindawi.com/journals/mpe/2008/697981/.

A. Goldberg e C. Harrelson. Computing the shortest path: A* search meets graph theory. SODA '05, 2005. Disponível: https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/GH05.pdf

A. Goldberg, C. Harrelson, H. Kaplan e R. Werneck. Efficient Point-to-Point Shortest Path Algorithms. 2005. Disponível: https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf

A. Suryadibrata, J. Young e R. Luhulima. Review of Various A* Pathfinding Implementations in Game Autonomous Agent, IJNMT (International Journal of New Media Technology), vol. 6, no. 1, pp. 43-49, 2019. Disponível: 10.31937/ijnmt.v6i1.1075.

R. Larson e A. Odoni, Urban operations research. Englewood Cliffs, N.J.: Prentice-Hall, 1981. Disponível: http://web.mit.edu/urban_or_book/www/book/

G. Rand. The life and times of the Savings Method for Vehicle Routing Problems. ORiON, vol. 25, no. 2, 2009. Disponível: 10.5784/25-2-78.