AirShuttle - Transfer em Aeroportos

Projeto CAL- 2018/19 - MIEIC

Turma 2 Grupo A

Professor das Aulas Práticas: Rosaldo José Fernandes Rossetti

Autores

Mário Mesquita, up201705723 (up201705723@fe.up.pt)

Moisés Rocha, up201707329 (up201707329@fe.up.pt)

Paulo Marques, up201705615 (up201705615@fe.up.pt)

Porto, 24 de maiol de 2019

Índice

Descrição do problema

A empresa AirShuttle presta **serviços de transfer** entre o aeroporto Francisco Sá Carneiro e hotéis ou outros locais da região, disponibilizando um certo número de carrinhas van para o efeito.

Neste trabalho, pretende-se implementar um sistema que, dado um conjunto de pedidos de serviços a prestar, permita à empresa AirShuttle planear as suas deslocações. Este terá de otimizar o agrupamento dos passageiros por carrinha, de modo a que os caminhos sejam também otimizados.

O problema foi decomposto em **3 iterações principais**. No entanto, de modo a implementar algumas funcionalidades adicionais, surge uma última, para uma totalidade de 4 iterações.

1ª iteração - Carrinha única sem agrupamento de diferentes reservas

Inicialmente, é considerada a existência de uma única carrinha de capacidade limitada para efetuar o transporte e apenas considerados os passageiros com reserva efetuada, não misturando clientes de diferentes reservas. Deste modo, o planeamento reduz-se à escolha do caminho mais curto desde o aeroporto até ao destino final para cada cliente, seguindo a ordem cronológica das reservas.

As reservas serão pré-processadas de modo a garantir que cada entrada em Ri tem um número de passageiros inferior à capacidade das carrinhas, garantindo que cada reserva pode ser satisfeita totalmente por uma única carrinha numa única viagem.

Salienta-se que cada viagem só pode ser efetuada se existir pelo menos um caminho de ida e um de volta, visto a carrinha precisar de regressar ao aeroporto. Por outras palavras, em cada viagem, o aeroporto e o destino devem pertencer a um mesmo componente fortemente conexo do grafo.

Adicionalmente, deve ser considerado que, por diversos motivos (obras nas vias públicas, …), certos caminhos podem se tornar inacessíveis. Nestas situações, as arestas correspondentes no grafo devem ser ignoradas durante o pré-processamento.

Pelos motivos descritos acima, e a fim de identificar destinos com pouca acessibilidade e a existência (ou não) de vias alternativas, torna-se necessário efetuar uma análise da conectividade do grafo.

2ª iteração - Carrinha única com agrupamento de diferentes reservas

Nesta segunda fase, devem ser agrupados passageiros de diferentes reservas numa só viagem sempre que possível, criando um percurso que passa por vários pontos antes de regressar ao aeroporto.

Isto permitirá diminuir o número de viagens de regresso ao aeroporto e reduzir os tempos de espera das reservas seguintes.

Salienta-se, no entanto, que a prioridade será diminuir o tempo de espera total dos clientes e não o número de carrinhas utilizadas. Deste modo, nem sempre os veículos serão totalmente preenchidos, apenas se a situação assim o permitir.

3ª iteração - Frota de carrinhas com agrupamento de diferentes reservas

Nesta fase é feito o avanço de uma única carrinha para uma frota de carrinhas, com capacidade limitada, surgindo então múltiplas deslocações e combinações de clientes possíveis.

O sistema deve otimizar o agrupamento dos passageiros por carrinha, de modo a que os caminhos sejam também otimizados.

4ª iteração - Integração de passageiros sem reserva

Finalmente, é considerada a possibilidade de integrar nas viagens passageiros sem reserva prévia vindos do aeroporto.

Destaca-se que este novo serviço apenas deve ser prestado se os próximos passageiros com reserva não forem afetados significativamente, isto é, se o tempo acrescido da viagem (no caso de integrar novos passageiros) não obrigar a que os próximos clientes fiquem mais tempo à espera.

Formalização do problema

Dados de entrada

Ci - sequência de veículos que irão efetuar o transporte dos clientes, Ci(i) representa o seu i-ésimo elemento.

Ri - sequência de reservas efetuadas online, sendo Ri(i) o seu i-ésimo elemento. Cada reserva é caracterizada por:

- name - nome do cliente
- NIF - número de identificação fiscal para faturação
- dest - local de destino
- arrival - hora de chegada ao aeroporto
- num - número de pessoas na reserva

Gi = (Vi, Ei) - grafo não dirigido pesado, composto por:

- V - vértices, representando pontos da rede viária, com:
  - ID - identificador do vértice
  - Adj ⊆ E, arestas que partem do vértice.
  - Position - Posição real do ponto no mapa
- E - arestas com:
  - ID - identificador da aresta
  - w - peso da aresta, no contexto do projeto representa o tempo médio que se demora a percorrer cada aresta (w vem de weight)
  - orig ∈ V - origem da aresta
  - dest ∈ V - destino da aresta

Ai - vértice do aeroporto
Cv - capacidade dos veículos da empresa, não contabilizando o condutor.
Ra - Raio de ação dos veículos da empresa.
Tm - Janela de tempo para escolher os clientes.
Md - Máxima distância possível entre dois locais considerados perto um do outro.

Dados de Saída

Gf = (Vf, Ef) - grafo não dirigido pesado, tendo Vf e Ef os mesmos atributos que Vi e Ei (à exceção dos atributos específicos a cada algoritmo utilizado).

Cf - sequência de carrinhas com os seus serviços a realizar atualizados, sendo Cf(i) o seu i-ésimo elemento. Cada carrinha é caracterizada por:

- S - sequência de serviços a realizar, sendo S(i) o seu i-ésimo elemento. Cada serviço é caracterizado por:
    - vacant - número de lugares vagos que sobraram
    - B = { c ∈ Ci | 1 ≤ j ≤ |B| } - sequência de reservas (sem repetidos) que vão no serviço, sendo B(j) o seu j-ésimo elemento. Cada um caracterizado por:
        - dest - destino pretendido pela reserva.
        - arrival - hora de chegada ao aeroporto.
        - deliver - hora de chegada ao destino.
    - P = { e ∈ Ei | 1 ≤ j ≤ |P| } - sequência de arestas a percorrer (pode haver repetidos), sendo P(j) o seu j-ésimo elemento.
    - start - hora de início do serviço, por outras palavras, horas que a carrinha é esperada sair do aeroporto com os clientes.
    - end - hora prevista do fim do serviço.

Restrições

Aos dados de entrada

- Cv > 0, visto que não faz sentido os veículos apenas poderem transportar o condutor.
- ∀ e ∈ Ei, w(e) > 0, visto o peso de cada aresta representar o tempo necessário para a percorrer.
- ∀ r ∈ Ri, dest(r) deve pertencer ao mesmo componente fortemente conexo do grafo Gi que o vértice Ai, pois é necessário calcular caminho de retorno para o aeroporto.
- ∀ e ∈ Ei, e deve ser uma estrada rodoviária pública, pois são as únicas utilizáveis pelo veículo da empresa. As que não forem não são incluídas no grafo Gi.
- ∀ r ∈ Ri, num(r) > 0.
- Ra > 0, visto este valor o ser o raio de um círculo.
- Ai ∈ Vi, o aeroporto é um vértice do grafo.
- Tm > 0, pois é uma medida de tempo.
- Md > 0, pois é uma medida de distância.

Aos dados de saída

- | Ci | = | Cf |
- ∀ vf ∈ Vf, ∃ vi ∈ Vi tal que vi e vf têm os mesmos valores para todos os atributos (não contando com os atributos específicos dos algoritmos utilizados).
- ∀ ef ∈ Ef, ∃ ei ∈ Ei tal que ei e ef têm os mesmos valores para todos os atributos (não contando com os atributos específicos dos algoritmos utilizados).

- ∀ s ∈ S, 0 < vacant(s) < Cv pois não pode haver sobrelotação dos veículos e cada veículo em serviço tem de ter pelo menos um passageiro.
- ∀ s ∈ S, |B(s)| > 0 pois só fará sentido realizar um serviço se houver clientes para transportar.
- ∀ b1 ∈ B de cada serviço de cada veículo, ¬ ∃ b2 ∈ B de outro serviço, tal que b1=b2.
- Seja e1 o primeiro elemento de P. É preciso que e1 ∈ Adj(Ai), pois cada veículo parte do aeroporto.
- Seja $e_{|P|}$ o último elemento de P. É preciso que dest($e_{|P|}$) = Ai, pois cada veículo termina o percurso no aeroporto.
- ∀ c ∈ Cf, end(c) > start(c).
- ∀ c ∈ Cf, ∀ i ∈ [1 ; |S|-1], end(S[i]) tem de ser antes de start(S[i+1]).
- ∀ c ∈ Cf, ∀ s ∈ S(c), arrival(∀ b ∈ B(s)) <= start(s) < deliver(∀ b ∈ B(s)) < end(s).

Função objetivo

A solução ótima passa por minimizar o tempo total entre a chegada dos clientes ao aeroporto e a entrega nos respetivos destinos, ou seja, a seguinte função:

$$\sum_{c \in Cf} \sum_{s \in S} \sum_{b \in B} (deliver(b) - arrival(b))$$

Estruturas de dados e Classes utilizadas

Representação do Grafo

Grafo

A representação do grafo foi feita com base na classe *Graph*, definida em *Graph.h*. Esta estrutura é constituída por um **vetor** de vértices (std::vector<Vertex*>), assim como por um ***HashMap*** (std::unordered_map<int, Vertex*>) que associa o ID de cada Vértice ao próprio Vértice. O vetor é necessário para ser possível iterar sobre todo o conjunto de vértices constituintes do grafo. Por outro lado, o mapa justifica-se para conseguir aceder aos vértices pelo seu ID em tempo constante (O(1)), visto esta ser uma operação frequente. Possui ainda diversos métodos para a sua manipulação, assim como algoritmos de pesquisa, análise da conetividade, entre outros.

Vértice

Os vértices são representados pela classe *Vertex*, definida em *Vertex.h.* Estes possuem um ID (igual ao que consta no mapa fornecido, por questões de coerência com os destinos das reservas), uma posição (classe definida em *Position.h*) e um vetor de arestas que “saem” do próprio vértice (std::vertex<Edge>). Além disso, possuem outros campos auxiliares específicos aos algoritmos que são aplicados nesta estrutura, como o Dijkstra ou o A*.

Aresta

As arestas são representadas pela classe *Edge*, definida em *Edge.h*. Visto o grafo ser não dirigido, possuem informação sobre a sua origem (Vertex*), destino (Vertex*) e peso (em segundos, significando o tempo médio necessário para percorrer este caminho). Registam ainda campos necessários para outras estruturas de dados e algoritmos, como um ID (para o GraphViewer) e uma *averageSpeed* (A*).

AirShuttle

Planeamento de serviços

Foi definida uma classe *ServicesPlanner* que será responsável por aplicar os algoritmos de ordenação das reservas pelas carrinhas em serviços.

Assim, esta estrutura guarda :

- um grafo (Graph *), que representa o mapa onde as carrinhas da empresa irão atuar;
- um aeroporto (ID do vértice), sendo este o ponto inicial de todas as viagens;

- lista de carrinhas (std::multiset<Van>), de modo a registar todas as carrinhas disponibilizadas para o transporte de clientes. A estrutura de dados ***multiset*** justifica-se pela utilidade de guardar as carrinhas ordenadas por próximo tempo em que estarão disponíveis;
- lista de reservas (std::multiset<Reservation>), guardando todas as reservas que foram feitas por clientes no dia em questão. A escolha de uma árvore binária balanceada justifica-se para manter esta lista organizada por tempo de chegada ao aeroporto. Optamos pela implementação da std do ***multiset*** para permitir também elementos repetidos, isto é, reservas que cheguem à mesma hora;
- campos actionRadius, timeWindow e maxDist, configuráveis e a serem usados nos vários algoritmos implementados na tomada de decisões.

Carrinha

A estrutura *Van* representa uma carrinha que transportará os clientes, constituída por uma capacidade e um vetor de serviços (std::vector<Service>, representando os serviços que a carrinha deverá satisfazer).

Serviço

A classe *Service* representa um serviço na empresa AirShuttle, caraterizado por um valor de lugares vagos, um tempo de início e fim (classe definida em *Time*), um vetor de reservas (std::vector<Reservation*>, registando quais as reservas, por ordem, que foram adicionadas a este serviço) e uma lista de arestas (std::list<Edge>; uma lista visto apenas ser percorrida sequencialmente, significando o caminho percorrido pelo serviço).

Reserva

As reservas dos clientes na empresa AirShuttle são representadas pela classe *Reservation*, constituídas por: nome de cliente, NIF, número de pessoas, destino e tempo de chegada ao aeroporto. Possui ainda o campo de tempo de chegada ao destino, a ser atualizado após adicionada a um serviço.

Desenho dos mapas e itinerários

A classe *MapDrawer* é responsável por, utilizando o *GraphViewer*, desenhar no ecrã os mapas, grafos e itinerários resultantes dos algoritmos aplicados. Possui diversos métodos relativos à apresentação dos mapas, seja a partir de ficheiros, um grafo ou um objeto do tipo *ServicesPlanner*.

Perspetiva de solução

Apresentam-se de seguida os problemas encontrados ao longo das várias iterações. Serão identificadas as técnicas de concepção e os principais algoritmos desenvolvidos e considerados, assim como as estruturas de dados utilizadas. É providenciado pseudocódigo apenas para os algoritmos por nós desenvolvidos, visto já existirem várias implementações para os restantes.

Salienta-se que, apesar de fazer mais sentido o grafo ser dirigido e ter sido esta a situação considerada na primeira fase do projeto, face aos dados do mapa providenciados para a segunda etapa, **foi necessário considerar o grafo não dirigido**.

Pré-Processamento dos Dados de Entrada

Pré-processamento do grafo

Previamente às iterações, o grafo Gi é pré-processado, reduzindo o seu número de vértices e arestas, a fim de aumentar a eficiência temporal dos algoritmos nele aplicados.

Primeiramente, são ignoradas as arestas do grafo que se encontrem inacessíveis (por motivos como obras nas vias públicas, entre outros) ou demasiado afastadas do aeroporto (considera-se que a empresa atua num círculo de ação com raio Ra e centro no aeroporto).

De seguida, devido à natureza circular das viagens, impondo que todas as carrinhas regressem ao aeroporto após conclusão do seu serviço, vértices que não pertencem ao mesmo componente fortemente conexo do grafo onde o aeroporto se encontra são também ser eliminados.

Garante-se assim que, aquando da aplicação dos algoritmos, o grafo não tem vértices e/ou arestas que não podem ser percorridos pelas carrinhas.

Pré-processamento das reservas

A sequência de reservas Ri é percorrida à procura de destinos que não pertençam ao grafo pré-processado. Nos casos em que isto se verifica, as respetivas entradas são removidas.

Adicionalmente, as reservas em Ri cujo valor de passageiros ultrapasse a capacidade dos veículos (Cv) são divididas, efetivamente removendo a original e adicionando tantas novas reservas quantas forem necessárias para garantir um número de passageiros inferior a Cv em cada. Esta divisão permite garantir que cada reserva pode ser totalmente satisfeita por uma só carrinha e é feita no momento de adição das reservas ao *ServicesPlanner*.

Finalmente, Ri é ordenada pela hora de chegada dos clientes ao aeroporto, de modo a que os algoritmos seguintes a executar possam seguir a ordem temporal e respeitar a prioridade de minimizar o tempo de espera dos passageiros. Esta ordenação é garantida pela estrutura de dados utilizada, um *multiset*.

Pré-cálculo dos percursos de duração mínima

Foi estudada a opção de pré-calcular todos os percursos de duração mínima entre os vários vértices, utilizando o algoritmo de **Dijkstra** para cada vértice ou o algoritmo de **Floyd–Warshall**.

A aplicação de um destes métodos permitiria aumentar consideravelmente a eficiência do restante programa, retirando a necessidade de calcular a duração mínima entre os vários pontos em cada novo percurso. No entanto, uma vez calcular a distância mínima entre todos os pares de vértices e não apenas entre os vértices interessantes para determinado percurso, a utilização destes algoritmos foi desconsiderada. No contexto deste problema, em média, apenas uma pequena parte de todas as combinações possíveis de vértices será usada, pelo que não se considerou justificável. No entanto, reconhecemos que noutras situações este pré-cálculo é necessário.

Não obstante, de modo a auxiliar os algoritmos seguintes a serem aplicados, nesta fase do pré-processamento é realizado o Dijkstra a partir do aeroporto, pré-calculando todas as distâncias e caminhos até qualquer ponto.

Identificação dos problemas encontrados

Na primeira iteração, com uma carrinha única, o problema reduz-se a encontrar o **caminho mais curto** entre o aeroporto e o destino final de cada reserva, pela ordem cronológica de horas de chegada dos clientes.

Adicionando a capacidade de combinar passageiros de diferentes reservas, na segunda iteração, as viagens passam a poder conter vários pontos de destino por onde a carrinha deve passar. O problema inerente é então o de passar por todos os pontos de interesse e regressar ao aeroporto, percorrendo a menor distância possível. Este começa a assemelhar-se ao famoso problema NP-difícil designado por ***Travelling Salesman Problem***.

A passagem, na terceira iteração, para uma frota de carrinhas, possibilita a combinação de diferentes reservas em diferentes veículos. O algoritmo a implementar deve conseguir decidir o número de carrinhas a ser usadas e a organização ótima de passageiros dentro delas, tendo em conta a função objetivo. O problema toma então parecenças com o ***Vehicle Routing Problem***, uma generalização do problema que tínhamos anteriormente (*Travelling Salesman*), considerando apenas um depósito (o aeroporto).

A iteração final vem trazer novas funcionalidades ao sistema, possibilitando a integração nas viagens de passageiros sem reserva. O algoritmo a implementar deve ser capaz de prever as situações em que estes serviços podem ser efetuados sem afetar os clientes com reserva.

Percurso de duração mínima entre dois pontos

Este é o problema base a resolver neste projeto, visto que se visa gerar e organizar múltiplas rotas.

Dos múltiplos algoritmos que podem ser usados para resolver esta questão, o de **Dijkstra** destaca-se pela sua facilidade de implementação e, apesar de ser um algoritmo para procurar o melhor caminho de um vértice para qualquer outro, também por ser possível otimizá-lo, fazendo-o parar quando encontra o vértice de destino.

No entanto, o algoritmo de Dijkstra faz a sua procura num círculo que se vai expandindo em torno do vértice de origem e, como se esperam receber grafos de grande dimensão, se os dois vértices não estiverem muito próximos, o algoritmo de Dijkstra pode tornar-se altamente ineficiente, pois a quantidade de vértices inúteis explorados pode chegar a crescer exponencialmente.

Numa tentativa de remediar esta situação, foi considerada a opção de implementar outros algoritmos mais eficientes, tais como o **A*** ou até o Dijkstra bidirecional. O A* surge como uma opção muito apelativa, visto alcançar soluções ótimas num tempo consideravelmente mais reduzido que o Dijkstra quando o grafo representa vias rodoviárias, como é o caso neste projeto.

A solução aplicada considera uma **combinação de Dijkstra com A***.

Inicialmente, é realizado um Dijkstra a partir do aeroporto, calculando a distância mínima até todos os outros pontos. Estes valores são guardados, visto todas as carrinhas iniciarem e acabarem o seu percurso neste ponto, aumentado a eficiência do algoritmo.

Assim, o percurso desde e até o aeroporto está já calculado. Em percursos entre vértices que não envolvam o aeroporto, é utilizado o A*, visto ser mais eficiente no cálculo direto entre 2 pontos.

Algoritmo de Dijkstra

Este algoritmo, concebido por Edsger W. Dijkstra em 1956, soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo. A sua aplicação resulta numa árvore de caminhos mais curtos desde o vértice inicial (aeroporto) até todos os outros pontos no grafo.

De modo a aplicar este método, cada vértice precisa de registar os seguintes campos adicionais:

- distance - duração mínima até à origem
- path - vértice anterior no caminho mais curto
- pathEdge - aresta anterior que coneta o *path* a este vértice

É ainda utilizada uma fila de prioridade para registar quais os vértices a serem processados. Seguindo uma abordagem *greedy*, esta deve ser mutável, procurando em cada passo maximizar o ganho imediato (neste caso, minimizar a duração da viagem).

O algoritmo subdivide-se em **preparação** e **pesquisa**.

A preparação requer percorrer todos os vértices do grafo, inicializando os valores de dur e path, sendo resolúvel em tempo linear relativamente ao número de vértices, O(|V|).

A pesquisa exige que sejam percorridos todos os vértices e arestas (|V| + |E|), sendo que a cada passo podem ser realizadas operações de extração, inserção ou *decrease-key* na fila de prioridade (O(log|V|), uma vez que |V| é o tamanho máximo da fila). Assim, a complexidade temporal da pesquisa é O((|V|+|E|)*log(|V|)).

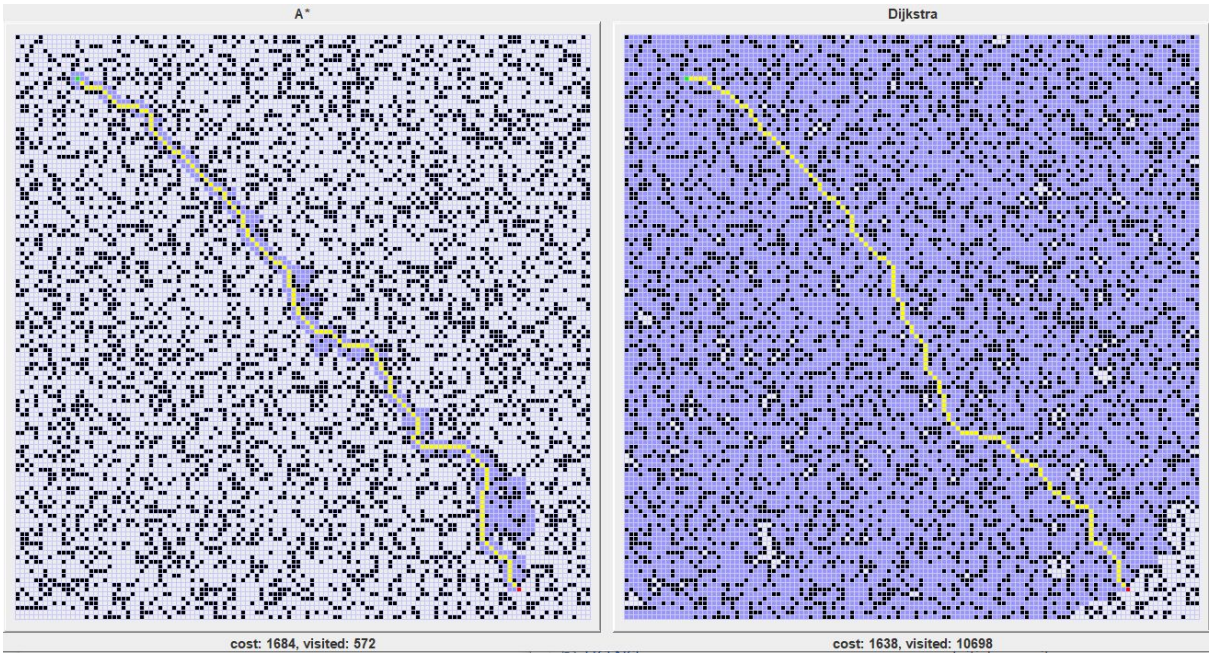
Algoritmo A*

O algoritmo **A***, concebido por Peter Hart, Nils Nilsson e Bertram Raphael em 1968, pode ser visto como uma extensão do algoritmo de Dijkstra. Este algoritmo utiliza heurística para guiar a sua pesquisa, obtendo assim melhores resultados, especialmente em grafos mais densos como é o caso da representação das estradas num mapa.

Para este feito, numa tentativa de descobrir quais os vértices “promissores”, faz uso da distância euclidiana entre dois vértices como função heurística, um valor facilmente obtível num mapa.

Assim, em comparação com o algoritmo de Dijkstra, será mais rápido, visto ter de analisar menos vértices. A sua complexidade é, no geral, O((|E| + |V|) * log(|V|)).

No entanto, apesar de mais rapidamente alcançável, a solução obtida com o A* não é garantidamente a mais óptima, ao contrário do que acontece noutros algoritmos. Esta situação encontra-se exposta na imagem abaixo, onde este algoritmo é comparado com o Dijkstra:



Analisando a imagem verifica-se que o A* visitou consideravelmente menos vértices do que o Dijkstra (572 contra 10698), acabando de executar muito mais rapidamente. No entanto, o caminho mais curto encontrado possui custo 1684, um valor superior ao custo retornado pelo Dijkstra de 1638.

Percurso de duração mínima entre todos os pares de pontos

Como mencionado na fase do pré-processamento, surgiu e foi estudada a hipótese de pré-calcular os percursos de duração mínima entre todos os pares de vértices, com a possibilidade de aumentar significativamente a eficiência do restante programa.

Para este efeito foram consideradas duas abordagens, a aplicação do método de Dijkstra para cada vértice e o algoritmo de Floyd-Warshall.

Algoritmo de Dijkstra

Este algoritmo, como já previamente mencionado, executa com uma complexidade temporal de O((|V|+|E|)*log(|V|)). Visto ser necessário registar as durações mínimas para todos os vértices, terá de ser executado |V| vezes, pelo que a complexidade total será de O(|V|*(|V|+|E|)*log(|V|)).

Surge como uma boa opção em grafos esparsos, como normalmente é o caso das redes viárias, visto o número de arestas ser da ordem do número de vértices. Nesta situação, então, a sua complexidade será $O(|V|^2*\log(|V|))$, sendo mais eficiente do que o algoritmo que será mencionado de seguida.

Algoritmo de Floyd-Warshall

Este algoritmo, desenvolvido por Robert Floyd em 1962, é utilizado para calcular as distâncias mínimas entre todos os pares de vértices num grafo. Apesar de não retornar informação sobre os percursos de distância mínima, é possível adaptá-lo para que tal aconteça.

Utiliza conceitos de programação dinâmica na sua execução, registando os valores de distância mínima e predecessor no caminho mais curto em duas matrizes de adjacência distintas.

A sua complexidade temporal escala cubicamente com o número de vértices ($\Theta(|V|^3)$), pelo que, relativamente aos outros métodos possíveis, é preferível em situações de grafos densos, em que o número de arestas é da ordem do quadrado do número de vértices. Nestas condições, surge como uma melhor opção do que a aplicação repetida do método de Dijkstra.

Caminho de duração mínima passando por vários destinos

Dado um conjunto de destinos que um veículo deve visitar num dado serviço, este deve sair do aeroporto, percorrer cada um deles e regressar ao ponto inicial, constituindo assim uma viagem circular.

Deparamo-nos então com um problema similar ao do caixeiro-viajante (***Travelling Salesman Problem***), um problema NP-difícil de elevada complexidade.

Para resolver este problema, existem tanto algoritmos que alcançam **soluções exatas** como algoritmos que chegam a **soluções aproximadas**, através de aproximação e heurística.

Soluções aproximadas

Dos algoritmos que chegam a soluções aproximadas, uma opção interessante é o algoritmo de **nearest neighbour**. Este baseia-se em escolher um vértice aleatório de início e escolher o vértice não visitado mais próximo como o seu próximo destino.

Esta abordagem gananciosa alcança uma solução em tempo reduzido. Contudo, esta não é ótima, acrescendo o facto de ser diferente conforme o vértice inicial escolhido como ponto de partida.

É de notar que, na situação considerada, visto o aeroporto ser sempre o ponto de partida, a implementação deste algoritmo exigiria que o vértice inicial fosse fixo, eliminando a aleatoriedade neste passo.

Soluções exatas

Dos algoritmos de solução exata aplicáveis ao *TSP*, destaca-se o algoritmo de **Held-Karp**, baseado em programação dinâmica.

No entanto, apesar de alcançar uma solução exata, possui uma complexidade temporal extremamente elevada $\theta(n^2 2^n)$, chegando até a ser menos eficiente do que o ***bruteforce***, o teste de todas as permutações, $\theta(n!)$ para n < 8.

Assim, se for tomada a opção de resolver este problema por métodos de solução exata, será considerada a hipótese de, conforme o número de vértices a visitar, alternar entre o *bruteforce* (para n < 8) e o algoritmo de Held-Karp (para n >= 8).

Organização das várias carrinhas conforme as reservas

Perante a organização das reservas pelas carrinhas, há dois pontos a considerar:

- O agrupamento das reservas.
- A escolha da carrinha

Destaca-se a importância de não deixar os clientes muito tempo à espera por uma carrinha, pelo que as reservas devem ser agrupadas de modo a que as horas de chegada ao aeroporto não sejam muito díspares. Será sempre dada prioridade aos clientes que cheguem primeiro ao aeroporto, existindo um tempo limite que estes podem ficar à espera havendo carrinhas disponíveis para os levar.

Aquando da integração de um novo cliente, será procurado um balanço entre a **adição a veículos já ocupados** (quando a situação assim o permitir), aumentando o tempo que esse veículo tomará a regressar ao aeroporto e estar novamente disponível, e o **despacho de uma nova carrinha**, diminuindo o número de veículos para atender os próximos clientes, mas sem afetar nenhuma das outras viagens já planeadas.

Por ordem de chegada dos clientes ao aeroporto, estes devem começar a preencher as carrinhas. Se não houverem veículos semi-preenchidos com lugares vagos, estes devem imediatamente ocupar uma nova carrinha. Em caso contrário, deve ser considerada a hipótese de integrar carrinhas já existentes. Poderá chegar-se à conclusão que não é viável, pelo que devem ocupar uma nova carrinha. Esta ordem de decisão garante a prioridade dos clientes que chegam primeiro ao aeroporto.

Um fator importante nesta decisão será considerar uma janela de tempo para tentar popular uma carrinha até à sua capacidade máxima. O limite temporal deve ser definido aquando da chegada do primeiro cliente que integra o veículo, garantindo que nem este nem nenhum dos seguintes passageiros ficarão demasiado tempo à espera. Prioriza-se

assim o tempo de espera dos clientes e não o preenchimento das carrinhas, sendo, de facto, possível as carrinhas iniciaram viagem com lugares vagos.

Contudo, mesmo que duas reservas distintas cheguem ao aeroporto na mesma altura e exista uma carrinha com capacidade suficiente para as transportar, pode não ser viável agrupá-las, por possuírem destinos consideravelmente afastados. Deste modo, deve existir uma tentativa de agrupar as reservas por "zonas", considerando um valor máximo de tempo entre quaisqueres dois destinos concorrentes num serviço. Durante a adição de um cliente a uma carrinha já ocupada, será verificado se a nova organização das viagens (contando com esse cliente) cumpre a regra estabelecida. Em caso negativo, este será atribuído a uma nova carrinha, na tentativa de agrupar os próximos clientes da mesma zona.

Salienta-se que, se a integração de um passageiro num veículo não afetar nenhum dos próximos clientes, por apenas chegarem ao aeroporto num espaço temporal muito afastado, este deve ser adicionado, independentemente de satisfazer ou não os critérios acima mencionados. Deste modo, clientes que possivelmente ficariam à espera no aeroporto pela próxima carrinha disponível são logo postos em viagem, permitindo terminar as viagens mais rapidamente, visto não ser necessário a carrinha regressar para os transportar.

Integração de passageiros sem reserva

Para além de apenas prestar serviços com reserva, será interessante tentar ao máximo prestar serviços a clientes que cheguem **sem reserva** prévia. O problema desta situação está assente no facto de o plano de viagens ser criado no início do dia, levando a que, no caso de se desejar tratar esses clientes como os com reserva, a reorganização dos serviços tivesse que acontecer quando as carrinhas se encontram já em atividade. Isto introduziria no sistema o risco de atrasar a viagem de alguns clientes, podendo até reduzir a satisfação destes.

Em consequência, será priorizada a satisfação dos clientes com reserva, apenas se adicionando clientes sem reserva se a sua integração não afetar o horário das viagens dos passageiros já previstos.

Assim, a estratégia para tentar adicionar clientes sem reserva será a seguinte:

1. Procurar carrinhas com lugares vagos que tenham um serviço num horário próximo do novo pedido de serviço (pode ser perguntado ao cliente quanto ele está disposto a esperar por uma carrinha). Se não houverem resultados, recusar o cliente.

2. Ordenar os serviços encontrados pelo nível de impacto que a adição do novo destino provocaria. Destaca-se que, se o destino do cliente já estiver a ser coberto por algum outro serviço, este praticamente não afetará o planeamento anterior e o cliente poderá ser aceite na carrinha (pode até ser perguntado se o cliente não se importa de apenas ser levado até algum nó

no caminho da carrinha, se este estiver próximo do destino do cliente, garantindo que os clientes com reserva não são afetados de modo algum).

3. Percorrendo a lista de serviços por ordem, começando pelo serviço em que a adição seria menos impactante, verificar se, depois da integração do novo cliente, o veículo continua a chegar a tempo de prestar o seu próximo serviço na lista. Caso isto se verifique para algum serviço, o cliente sem reserva deve ser integrado e o algoritmo termina.

4. Se o cliente não tiver sido aceite em nenhum dos serviços na lista do passo anterior, este deve ser recusado.

Possibilidade de viagens das residências ao aeroporto

O objetivo principal será sempre levar os clientes do aeroporto para o seu destino. Porém, devido à **natureza circular das viagens**, poderia ser considerada a situação de levar passageiros das suas residências ao aeroporto.

Este tipo de serviços poderia ser efetuado com reserva. No entanto, visto que o planeamento das rotas trata-se de um problema do caixeiro-viajante, um problema já NP-difícil, e que teria de acrescer o cuidado adicional de apenas apanhar esses clientes quando já se descarregou pessoas suficientes para haver espaço, impondo uma ordem específica na visita de cada destino, a complexidade do problema tornar-se ia demasiado elevada para ser analisada.

Assim, a alternativa a esta situação é considerar que estes pedidos apenas podem ser feitos sem reserva, existindo a possibilidade de os rejeitar se necessário. E, como não se quer afetar os clientes com reserva, esse recolher seria feito apenas após a entrega de todos os passageiros aos seus destinos. Isto obrigaria a alterar a estrutura dos dados de saída, de modo a manter o registo de quais serviços são de carga e quais são de descarga, levando à reestruturação dos algoritmos usados e a um aumento considerável do problema em estudo.

Para evitar a reestruturação dos restantes algoritmos, assim como dos dados de saída, poder-se-ia apenas aceitar um pedido de ida para o aeroporto por serviço prestado, simplificando bastante o problema.

No entanto, devido a estas sucessivas simplificações, o serviço a prestar deixaria de fazer sentido, pelo que a sua implementação foi desconsiderada.

Porém, no caso de ser implementado este serviço, a estratégia para tentar integrar este serviço seria a seguinte:

1. Procurar serviços ainda sem recolha cuja carrinha não necessite de se desviar consideravelmente para recolher o novo cliente. Se não houverem resultados, recusar o serviço.

2. Ordenar os serviços encontrados pelo nível de impacto que a adição do novo destino provocaria.

3. Percorrendo a lista de serviços por ordem, começando pelo serviço em que a adição seria menos impactante, verificar se, depois da integração do novo cliente, o veículo continua a chegar a tempo de prestar o seu próximo serviço na lista. Deve ainda ser verificado se a carrinha consegue, de facto, chegar ao aeroporto a tempo de satisfazer o novo cliente que pediu a recolha. Caso isto se verifique, este deve ser integrado e o algoritmo termina.

4. Se o cliente não tiver sido aceite em nenhuma das carrinhas na lista do passo anterior, este deve ser recusado.

Análise da conectividade

Como já mencionado anteriormente, durante a fase de pré-processamento do grafo, devem ser removidos os vértices que não são alcançáveis a partir do aeroporto.

Neste âmbito, torna-se necessário proceder à análise da conectividade do grafo, identificando qual o componente fortemente conexo ao qual o aeroporto pertence. Qualquer vértice que não faça parte deste componente deve ser eliminado.

Na primeira fase do projeto, visto ser considerado um grafo dirigido, foram estudados e considerados os algoritmos de **Kosaraju** e **Tarjan**. No entanto, uma vez ser necessário considerar o grafo não dirigido (devido ao mapa fornecido para a segunda fase deste projeto), o algoritmo aplicado foi uma simples **Depth-First Search.**

Com efeito, após aplicar este algoritmo, marcando os vértices visitados, verificou-se existirem pontos inalcançáveis a partir do local selecionado como aeroporto. Procedeu-se então à eliminação destes vértices do grafo, assim como de todas as arestas desde e até eles.

Algoritmo de Kosaraju

Para encontrar todos os componentes fortemente conexos do grafo, um dos algoritmos considerados foi o de **Kosaraju**, que consiste no seguinte:

1. Fazer uma pesquisa em profundidade no grafo, numerando os vértices em pós-ordem;

2. Inverter todas as arestas do grafo;

3. Realizar uma segunda pesquisa em profundidade, começando pelos vértices de numeração superior ainda por visitar.

No final deste algoritmo, cada árvore obtida é um componente fortemente conexo. Contudo, apenas interessa o componente respetivo ao aeroporto, pelo que o algoritmo pode ser otimizado e parar aquando da descoberta deste.

O método baseia-se em duas pesquisas em profundidade, sendo também necessário inverter todas as arestas do grafo. Cada uma destas operações possui complexidade O(V + E), sendo realizadas em sequência. Deste modo, a complexidade do algoritmo é também O(V + E), ou seja, corre em tempo linear.

Algoritmo de Tarjan

Além do método de Kosaraju mencionado anteriormente, o algoritmo de **Tarjan** surge também como uma opção para determinar os componentes fortemente conexos de um grafo.

Apesar de executar similarmente em tempo linear, este baseia-se numa única pesquisa em profundidade, sendo, portanto, mais eficiente.

Por este motivo, será considerada a opção de implementar também este algoritmo.

Depth-First Search

Para analisar a conectividade de um grafo não dirigido basta aplicar uma DFS a partir do vértice inicial (aeroporto), marcando os vértices percorridos como visitados. Este algoritmo consiste em:

1. Marcar todos os vértices do grafo como não visitados.

2. Marcar o vértice atual como visitado (na primeira iteração será o aeroporto).

3. Para todas as arestas que têm origem neste vértice, se o vértice de destino ainda não tiver sido visitado, visitá-lo recursivamente, repetindo o ponto 2.

Pelo caráter recursivo deste algoritmo, garante-se que os vértices serão visitados “em profundidade” em oposição a “em largura”, tentando sempre explorar cada “ramo” do grafo o mais longe possível antes de regressar atrás à decisão anterior (*backtracking*). Como a pesquisa se inicia a partir do aeroporto, apenas serão encontrados os vértices que fazem parte do mesmo componente fortemente conexo.

A DFS visita todos os vértices inicialmente para os marcar como não visitados (|V|) e, durante a pesquisa, percorre todas as arestas de cada vértice encontrado (|E|). Assim, a complexidade deste algoritmo será, em média, O(|V| + |E|).

Algoritmos implementados

Destacam-se neste capítulo os principais algoritmos que foram projetados e implementados por nós. Não são mencionados algoritmos como o de Dijkstra ou A* visto serem usados os originais, com modificações pouco relevantes.

Salienta-se que, na terceira iteração, existiu uma tentativa de experimentar diferentes tipos de algoritmos para combinar os passageiros. A sua forma mais simples segue uma abordagem “first come first served”, tentando sempre agrupar os passageiros pela ordem em que chegam ao aeroporto. No entanto, foram consideradas várias métricas para tentar melhorar o algoritmo, como considerar uma janela de tempo desde que o primeiro cliente chega para tentar agrupar os clientes seguintes, ou apenas agrupar clientes cujos destinos estão “próximos” uns dos outros.

Análise da conectividade - DFS

Pseudocódigo

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

Análise da Complexidade

Este algoritmo começa por percorrer todos os vértices no grafo (|V|) para os marcar como não visitados. De seguida, a partir do aeroporto, irá recursivamente visitar todos os pontos alcançáveis através das arestas que encontra, efetivamente percorrendo o componente fortemente conexo do aeroporto e marcando-o como visitado. Nesta fase, o número de arestas é no máximo |E|, se percorrer todas as arestas do grafo.

Assim, em média, este algoritmo terá uma complexidade de O(|V| + |E|).

Pré-Processamento

Pseudocódigo

```
1   // Gi - graph representing the map
2   // Ri - sequence of reservations
3   // a - airport
4   PRE_PROCESS_DATA(Gi, Ri, a):
5       DFS_CONNECTIVITY(Gi, a)
6       REMOVE_UNVISITED_VERTICES(Gi)
7
8       for each r ∈ Ri do
9           if FIND_VERTEX(Gi, dest(r)) == NULL then
10              REMOVE_RESERVATION(Ri, r)
11
12      DIJKSTRA_SHORTEST_PATH(Gi)
```

Análise da Complexidade

Inicialmente é feita uma análise da conectividade com o algoritmo mencionado previamente DFS_CONNECTIVITY, cuja complexidade é O(|V| + |E|).

Seguidamente, são eliminados do grafo todos os vértices marcados como não visitados. Esta operação requer as seguintes etapas:

- Percorrer todos os vértices e eliminar os que não estão marcados como visitados;
- Percorrer todas as arestas e eliminar as que tinham destino em algum dos vértices eliminados.

A primeira fase apresenta complexidade $O(|V|^2)$, visto percorrer toda a lista de vértices (|V|) e a operação de eliminação de um vetor possuir também complexidade linear.

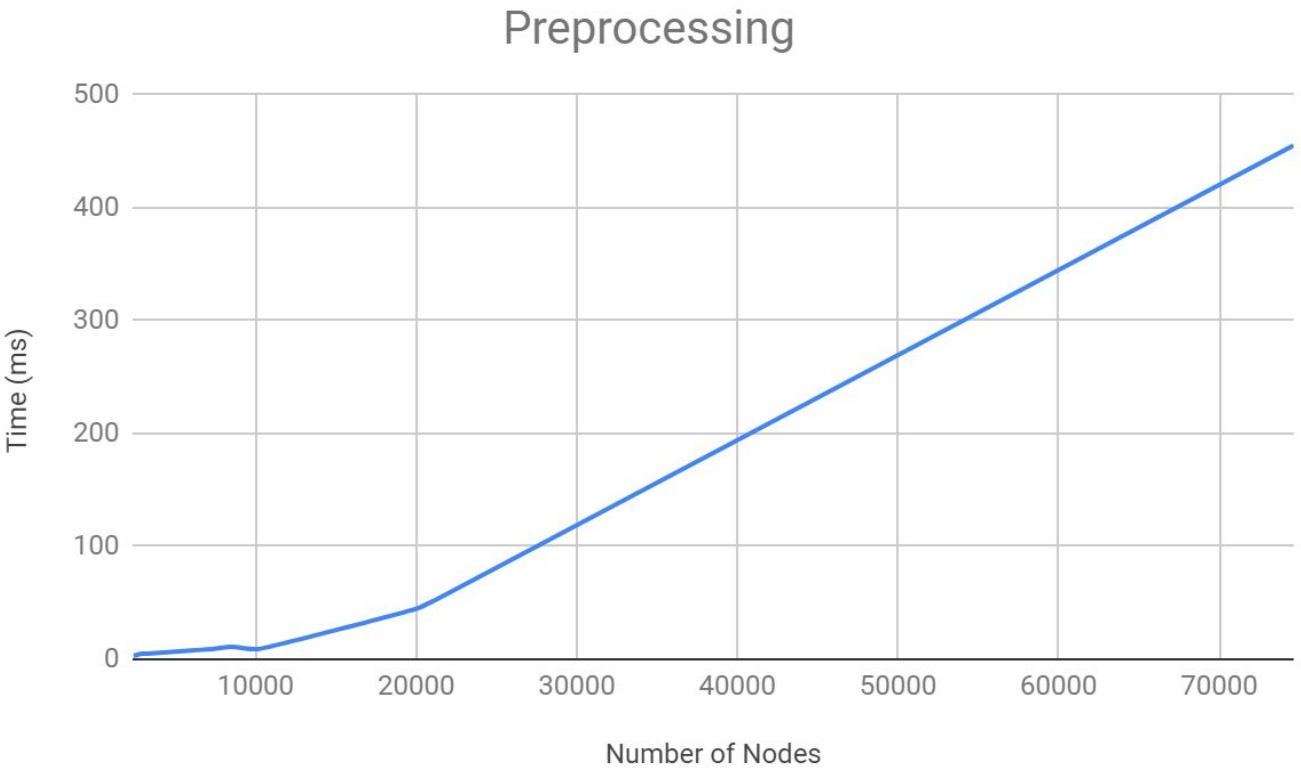
A segunda fase irá percorrer todas as arestas (|E|) e para cada uma verificar o destino com os vértices removidos, possivelmente eliminando a aresta. Visto ser utilizada uma *Hash Table* para registar quais os vértices removidos, esta verificação é conseguida em tempo constante. A eliminação da aresta possui complexidade O(|e|), onde |e| é o número de arestas a sair do vértice respetivo. No entanto, face ao contexto do problema, este valor será sempre consideravelmente reduzido, muito inferior ao número total de arestas. Por este motivo, admite-se que a segunda fase corre em tempo O(|E|).

Assim, a remoção dos vértices não visitados tomará tempo $O(|V|^2 + |E|)$.

Segue-se a necessidade de percorrer toda a lista de reservas (|R|) e, caso o vértice associado já não exista, remover a reserva. Pelo uso de *Hash Tables*, a procura do vértice de destino é feita em O(1). Como as reservas estão guardadas num *multiset*, a remoção toma tempo O(log(|R|)). Assim, a complexidade desta etapa é O(|R| * log(|R|)).

Finalmente, de modo a auxiliar a aplicação dos próximos algoritmos, é realizado o algoritmo de Dijkstra para calcular a distância mínima de todos os pontos até ao aeroporto. Visto ter sido usada uma fila de prioridade mutável, este executa em O((|V|+|E|)*log(|V|)).

Concluindo, tendo em conta os fatores dominantes, na sua totalidade o algoritmo possui complexidade $O(|V|^2 + |E|*\log(|V|) + |R|*\log(|R|))$.



1ª iteração - Carrinha única sem agrupamento de diferentes reservas

Pseudocódigo

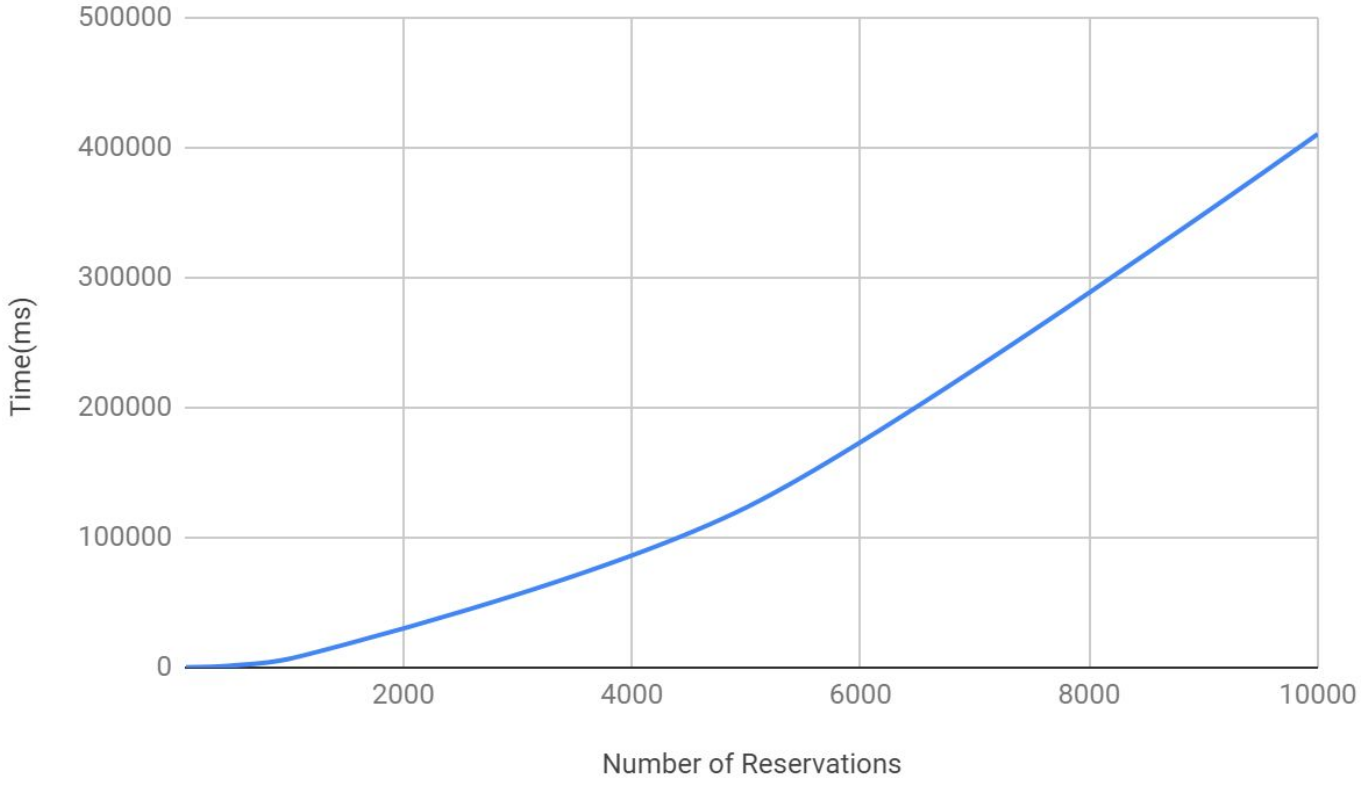
```
// Gi - graph representing the map
// Ri - sequence of reservations
// v - van to be used
// a - airport
PLAN_SINGLE_VAN_NOT_MIXING_PASSENGERS(Gi, Ri, v):

    CLEAR_SERVICES(v)

    for each r ∈ Ri do
        // Get path to destination and back to airport
        path_edges = GET_PATH_EDGES(Gi, a, dest(r))
        return_path = REVERSE_PATH(path_edges)
        path_edges = JOIN_PATHS(path_edges, return_path)

        // Create and add new service to the van
        // SERVICE(vacant, start_time, reservation, path)
        new_service = SERVICE(van_capacity(v) - num_people(r), LAST_SERVICE_END(v), r, path_edges)
        ADD_SERVICE(v, new_service)
```

Análise da Complexidade

O algoritmo inicia-se com a remoção de todos os serviços da carrinha v, um valor que será sempre menor ou igual a |R|.

De seguida, são percorridas todas as reservas (|R|) de modo a organizá-las em serviços. Para cada uma, inicialmente esta será removida da lista de reservas (log(|R|)). Logo a seguir, é calculado o caminho desde o aeroporto até ao destino (obtível pela aplicação anterior do Dijkstra no pré-processamento), sendo necessário iterar sobre todos os vértices do caminho (|v|). Segue-se a obtenção do caminho de regresso, bastando inverter o caminho anterior (|v|), e, finalmente, a junção dos dois caminhos (|v|). Assim, a organização das reservas em serviços possui uma complexidade de O(|R| * (log(|R|) + |v|)).

Na totalidade, a complexidade do algoritmo é então **O(|R| * (log(|R|) + |v|))**, sendo |v| o número médio de vértices a percorrer para alcançar o destino das reservas.


500000
400000
300000
200000
100000
0
Time(ms)
2000
4000
6000
8000
10000
Number of Reservations

2ª iteração - Carrinha única com agrupamento de diferentes reservas

Pseudocódigo

```
// Gi - graph representing the map
// Ri - sequence of reservations (multiset organized by arrival time)
// Vi - vans to be used (multiset organized by next available time)
// a - airport
// tm - time window to wait for passengers
// md - max distance to be considered when grouping passengers
PLAN_SINGLE_VAN_MIXING_PASSENGERS(Gi, Ri, Vi, a, tm, md):

    // use first van available
    van = BEGIN(Vi)
    REMOVE_ELE(Vi, BEGIN(Vi))

    // must assign all reservations
    while not IS_EMPTY(Ri) do

        // get earliest reservation not already assigned
        early_res = BEGIN(Ri)
        REMOVE_ELE(Ri, BEGIN(Ri))

        service = Ø
        PUSH_BACK(service, early_res)

        // group earliest client with next reservations
        num_slots = capacity(van) - num_people(early_res)
        aux = BEGIN(Ri)
        while num_slots > 0 AND aux != END(Ri) do
            inTimeWindow = arrival(aux) < arrival(early_res + tm)
            inRange = EUCLIDIAN_DISTANCE(dest(aux), dest(early_res)) < md
            canFit = num_slots >= num_people(aux)

            // no more reservations fit time window
            if not inTimeWindow then
                break

            // valid match
            if inRange AND canFit then
                PUSH_BACK(service, aux)
                num_slots -= num_people(aux)
                REMOVE_ELE(Ri, aux)

            // get next reservation
            aux = NEXT_ELE(Ri, aux)

        // calculate best path through all the destinations
        path = CALCULATE_PATH_FROM_SERVICE(Gi, a, service)

        // assign deliver time to each reservation
        departure_time = TARDIEST_RESERVATION_TIME(service)
        total_time = ASSIGN_TIME_OF_DELIVER_TO_RESERVATIONS(path, service, departure_time)

        // update next time available of used van
        next_time_available(early_van) = departure_time + total_time

        // add service to van
        ADD_SERVICE(early_van, SERVICE(num_slots, departure_time, service, path))

    // re-insert van in the multiset
    INSERT(Vi, early_van)
```

```
// Gi - graph
// service - list of reservations to attend
CALCULATE_PATH_FROM_SERVICE(Gi, a, service):

    // vector of vertices to traverse
    vertices = GET_VERTICES_FROM_SERVICE(Gi, service)

    // calculate best path through all the vertices
    // vector of edges
    path = CALCULATE_PATH(Gi, a, vertices)

    return path
```

```
// Gi - graph
// a - airport vertex
// vertices - list of vertices to calculate path
CALCULATE_PATH(Gi, a, vertices):

    path = Ø

    vertices_copy = vertices

    // from airport trip
    closest = GET_CLOSEST_VERTEX(a, vertices_copy)
    ERASE(vertices_copy, closest)

    // path calculated with dijkstra in pre-processing
    from_aiport = GET_PATH_EDGES(Gi, a, closest)
    APPEND(path, from_aiport)

    // nearest neighbour approach using A*
    current_source = closest
    while not IS_EMPTY(vertices_copy) do
        closest_to_source = GET_CLOSEST_VERTEX(current_source, vertices_copy)
        ERASE(vertices_copy, closest_to_source)

        ASTAR(Gi, current_source, closest_to_source)
        temp_path = GET_A_STAR_PATH(Gi, current_source, closest_to_source)
        APPEND(path, temp_path)

        current_source = closest_to_source

    // return to airport trip
    ASTAR(Gi, current_source, airport)
    temp_path = GET_A_STAR_PATH(Gi, current_source, airport)
    APPEND(path, temp_path)

    return path
```

```
1   // path - list of edges that will be traversed
2   // service - list of reservations to attend
3   // departure_time - time at which the van leaves
4   ASSIGN_TIME_OF_DELIVER_TO_RESERVATIONS(path, service, departure_time):
5
6       total_time = 0
7       for e ∈ path do
8           total_time += weight(e)
9
10          for r ∈ service do
11              if assigned(r) then
12                  continue
13
14              if dest(r) == id(dest(e)) then
15                  SET_DELIVER_TIME(r, departure_time + total_time)
16                  assigned(r) = true
17      return total_time
```

Análise da Complexidade

AssignTimeOfDeliverToReservations

Este algoritmo pretende determinar o tempo de entrega para todas as reservas em determinado serviço, à medida que vai percorrendo as arestas do caminho. Assim, irá iterar sobre todo o *path* (|E|), procurando em cada passo as reservas cujo destino é igual ao vértice atual para marcar o tempo de entrega dessa reserva. Deste modo, a complexidade do algoritmo é **O(|E| * |r|)**, sendo |r| o número de reservas no serviço.

CalculatePath

O CALCULATE_PATH inicia-se copiando a lista de vértices numa reserva, de tamanho |v|, o número de vértices no serviço (que será sempre insignificante comparando com o número total de vértices |V|).

De seguida, é percorrida a lista de vértices à procura do ponto mais próximo do aeroporto. Após encontrado, são utilizados os valores já calculados do Dijkstra para determinar as arestas do caminho desde o aeroporto até esse ponto. Toda esta etapa toma tempo |v|.

A próxima fase é então uma abordagem Nearest Neighbour de calcular os caminhos mais ótimos entre os vários vértices. Assim, enquanto existirem vértices por percorrer (|v|), é determinado o próximo ponto mais próximo e aplicado o algoritmo A* desde o ponto anterior até ao encontrado, obtendo e adicionado as arestas encontradas ao *path*. No total, esta etapa executa em O(|v| * (|v| + (|V|+|E|)*log(|V|))). No entanto, como v é um subconjunto de V, admite-se executar em O(|v| * (|V| + |E|) * log(|V|)).

A fase final pressupõe a aplicação do A* desde o vértice final de volta para o aeroporto, tomando a complexidade deste algoritmo.

Conclui-se então que o CALCULATE_PATH é dominado pela fase de Nearest Neighbour, pelo que a sua complexidade é **O(|v| * (|V| + |E|) * log(|V|))**.

CalculatePathFromService

Este método calcula o conjunto de vértices que determinado serviço deve percorrer e segue a aplicação do CALCULATE_PATH sobre eles. Assim, a complexidade deste algoritmo é dominada pelo segundo passo, ou seja, **O(|v| * (|V| + |E|) * log(|V|))**.

PlanSingleVanMixingPassengers

Este algoritmo inicia-se removendo a primeira carrinha do multiset (log(|C|)) e irá percorrer toda a lista de reservas para lhes atribuir serviços (|R|).

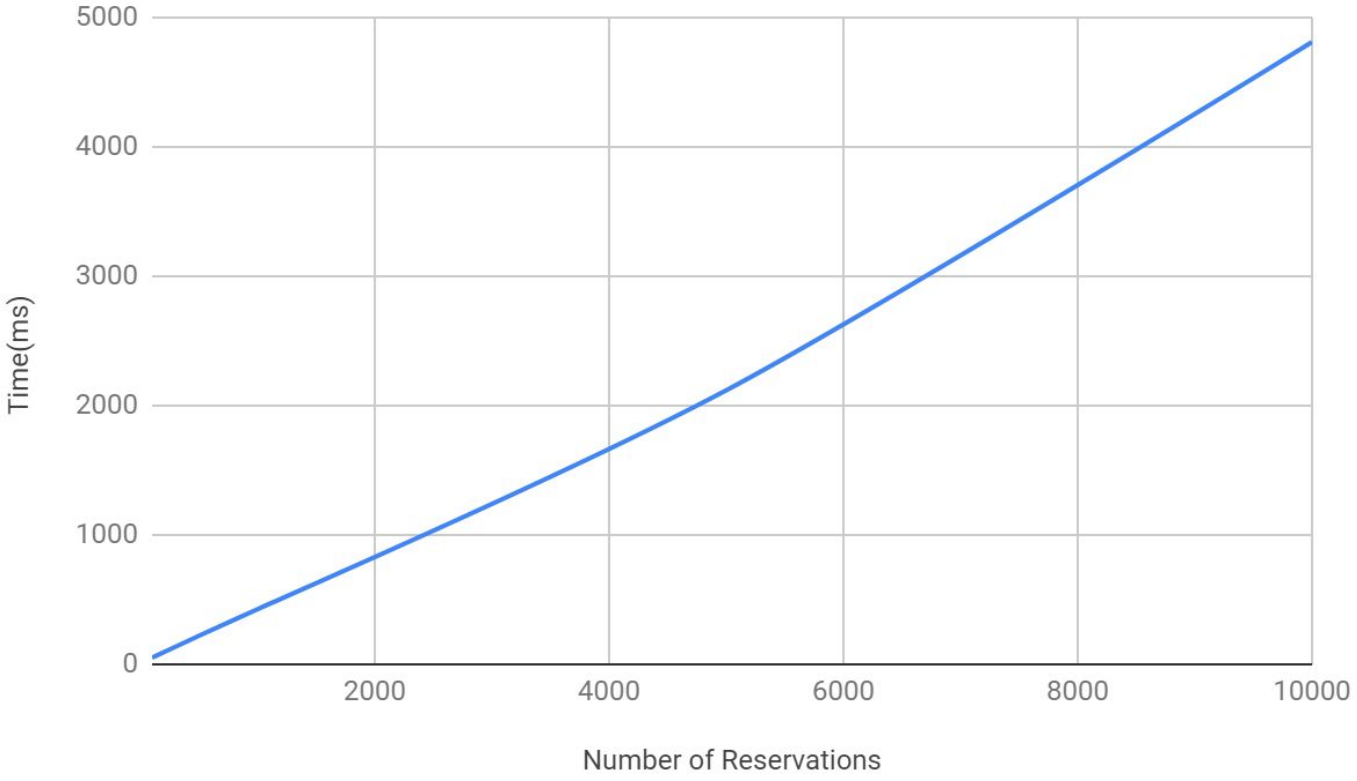
Em cada iteração, inicialmente é obtido e removida a próxima reserva (conforme ordem de chegada ao aeroporto), em tempo log(|V|).

De seguida, é percorrida a restante lista de reservas à procura de possíveis combinações, adicionado a um vetor as reservas a combinar. Esta operação tomará tempo O(|R|).

Segue-se o cálculo do caminho entre os vários vértices das reservas agrupadas, usando o algoritmo CALCULATE_PATH_FROM_SERVICE mencionado acima.

Finalmente é atribuído o tempo de chegada ao destino às reservas, pelo algoritmo ASSIGN_TIME_OF_DELIVER_TO_RESERVATIONS já referido.

Assim, a complexidade total deste algoritmo, tendo em conta os fatores dominantes, é **O((log(|C|) + |R| * (|R| + |v| * log(|V|) * (|V| + |E|))).**


5000
4000
3000
2000
1000
0
Time(ms)
2000
4000
6000
8000
10000
Number of Reservations

3ª iteração - Frota de carrinhas com agrupamento de diferentes reservas

Pseudocódigo

```
// Gi - graph representing the map
// Ri - sequence of reservations (multiset organized by arrival time)
// Vi - vans to be used (multiset organized by next available time)
// a - airport
// tm - time window to wait for passengers
// md - max distance to be considered when grouping passengers
PLAN_VANS_FLEET_MIXING_PASSENGERS(Gi, Ri, Vi, a, tm, md):

    CLEAR_SERVICES(Vi)

    // must assign all reservations
    while not IS_EMPTY(Ri) do

        // get earliest reservation not already assigned
        early_res = BEGIN(Ri)
        REMOVE_ELE(Ri, BEGIN(Ri))

        // get earliest van available
        early_van = BEGIN(Vi)
        REMOVE_ELE(Vi, BEGIN(Vi))

        // group earliest client with next reservations
        occ_capacity = 0
        service = MIX_CLIENTS_WITH_EARLIEST(Gi, Ri, early_res, early_van, tm, md, &occ_capacity)

        // calculate best path through all the destinations
        path = CALCULATE_PATH_FROM_SERVICE(Gi, a, service)

        // assign deliver time to each reservation
        departure_time = TARDIEST_RESERVATION_TIME(service)
        total_time = ASSIGN_TIME_OF_DELIVER_TO_RESERVATIONS(path, service, departure_time)

        // update next time available of used van
        next_time_available(early_van) = departure_time + total_time

        // add service to van
        ADD_SERVICE(early_van, SERVICE(capacity(early_van) - occ_capacity, departure_time, service, path))

        // re-insert van in the multiset
        INSERT(Vi, early_van)
```

```
// Gi - graph representing the map
// Ri - sequence of reservations (multiset)
// res - first reservation that entered the van
// van - van to be used
// md - max distance to be considered when grouping passengers
// occ - variable passed by reference to return unmber of occupied seats in the van
MIX_CLIENTS_WITH_EARLIEST(Gi, Ri, res, van, tm, md, occ):

    // calculate time limit the van can wait for clients
    limit = arrival(res) + tm
    if limit < next_time_available(van) then
        limit = next_time_available(van)


    // vector of reservations for the new service
    service = Ø
    PUSH_BACK(service, res)

    // look for reservations arriving within the time window limit
    // that are close in distance to the earliest reservation
    // and group them togheter
    origin = position(FIND_VERTEX(Gi, dest(res)))
    occ_capacity = num_people(res)
    current_reservation = BEGIN(Ri)
    while (arrival(current_reservation) < limit) AND (current_reservation != END(Ri)) do

        node_pos = position(FIND_VERTEX(Gi, dest(current_reservation)))

        // found valid match within acceptable distance and which fits in the van
        if ((EUCLIDIAN_DISTANCE(origin, node_pos) < md) AND (occ_capacity + num_people(current_reservation) <= capacity(van))) then
            PUSH_BACK(service, current_reservation)
            occ_capacity += num_people(current_reservation)
            REMOVE_ELE(Ri, current_reservation)

            // cant fit more reservations
            if occ_capacity >= capacity(van) then
                break
        else
            current_reservation = NEXT_ELE(Ri)

    return service
```

Análise da Complexidade

MixClientsWithEarliest

O algoritmo começa a ganhar complexidade no ciclo em que irá ser tentado misturar a reserva *res* com outros clientes. Este irá iterar sobre a lista de reservas que falta atribuir (limite superior |R|) enquanto o tempo de chegada da reserva atual estiver dentro do limite estabelecido. Para cada uma, verifica se o seu destino está “próximo” do destino da reserva *res*, assim como se o número de clientes ainda cabe na carrinha. Em caso válido, adiciona-a ao vetor de serviços e remove a reserva da lista. Esta remoção tomará tempo |R| no pior caso, mas, visto a carrinha possuir capacidade limitada (admitimos ser 10), irá apenas ser executada um número máximo de 10 vezes no pior caso. Assim, não consideramos esta remoção como impactante para a complexidade geral.

Concluindo, este algoritmo possui complexidade O(|R|).

PlanVansFleetMixingPassengers

Este algoritmo irá iterar sobre todas as reservas, isto é, sobre |R| elementos.

Em cada iteração, inicialmente determina qual a próxima reserva que deve ser atendida e qual a próxima carrinha disponível, removendo estes elementos das respetivas listas. Visto serem usados multisets, esta fase toma tempo O(log(|R|) + log(|C|)).

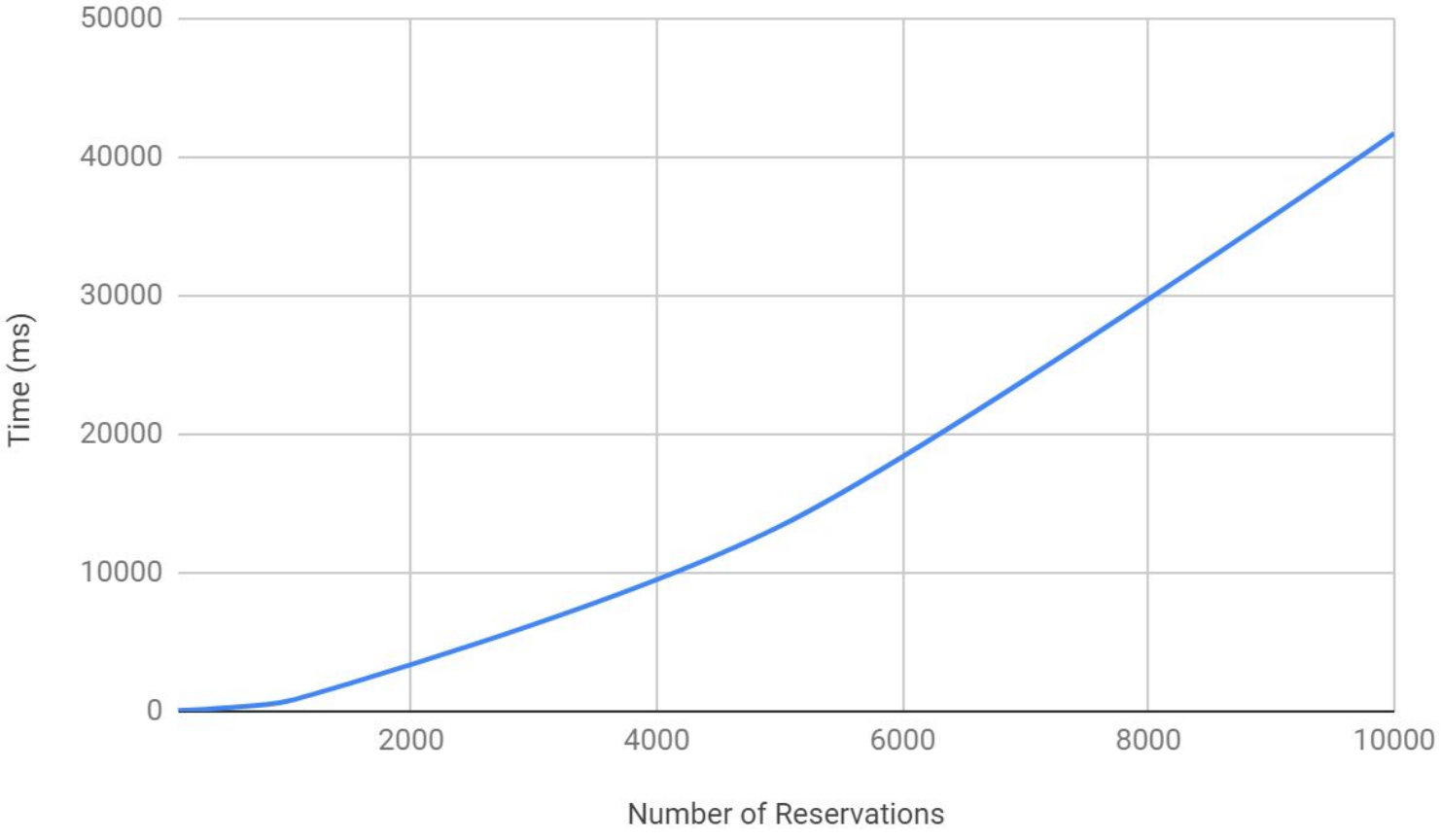
De seguida, é utilizado o algoritmo MIX_CLIENTS_WITH_EARLIEST para obter um vetor de reservas que podem ser combinadas, que, como mencionado acima, possui complexidade O(|R|).

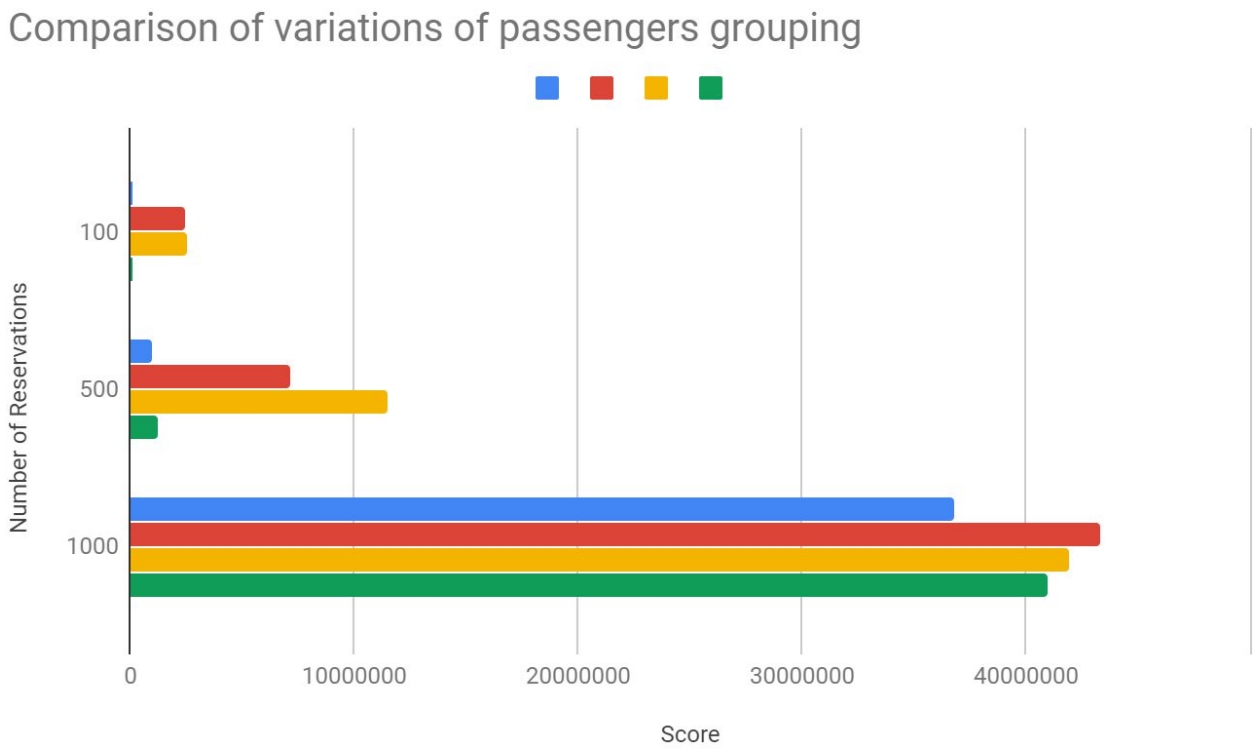
Procede-se à determinação do caminho a partir do vetor de reservas, utilizando o algoritmo CALCULATE_PATH_FROM_SERVICE que executa em **O(|v| * (|V| + |E|) * log(|V|))**.

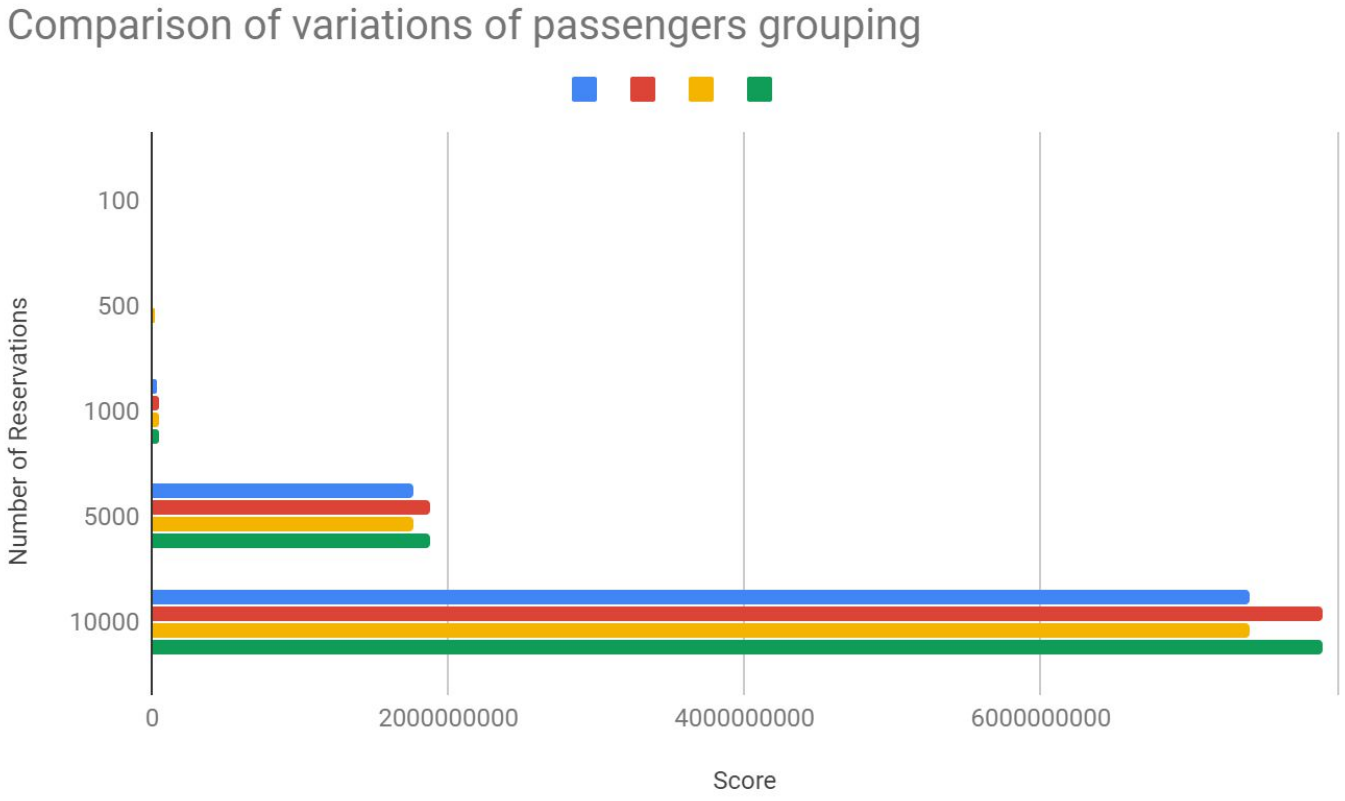
É então aplicado o algoritmo ASSIGN_TIME_OF_DELIVER_TO_RESERVATIONS, que como foi visto executa em O(|E | * |r|).

Finalmente , adiciona-se o serviço à carrinha e re-insere-se esta no multiset, tomando tempo O(log(|C|)).

Concluindo, considerando os fatores dominantes no algoritmo, a sua complexidade é **O(|R| * (log(|C|) + |R| + |v| * log(|V|) * (|V| + |E|)))**.







Este gráfico pretende comparar diferentes formas de agrupar passageiros. Foram consideradas as métricas de **considerar uma janela de tempo** e **agrupar reservas com**

destinos próximos. O esquema de cores no gráfico é o seguinte, onde S representa “sim” e N “nao”, relativamente às métricas consideradas: azul S/S, vermelho N/N, amarelo N/S, verde S/N.

O score é calculado com base na função objetivo, onde um maior score significa um maior tempo de espera dos clientes desde que chegaram ao aeroporto até chegarem ao destino.

Verificamos que, para conjuntos de dados pequenos, o uso das métricas consideradas realmente fazem diferença e apresentam melhores resultados face a não usá-las. No entanto, para um número de reservas grande (5000 e 10000), as variações azul e amarelo, assim como, vermelho e verde, ficam cada vez mais similares.

Concluímos então que, face a *datasets* de elevada dimensão, o fator de **janela de tempo** deixa de ser relevante. No entanto, a métrica de **agrupar destinos próximos** apresenta-se como cada vez importante, minimizando a função objetivo.

4ª iteração - Integração de passageiros sem reserva

Pseudocódigo

```
1   // Gi - graph
2   // Vi - list of vans with services assigned
3   // a - airport
4   // client - new client with no reservation to integrate
5   // waiting_time - time the client is willing to wait for a van
6   INTEGRATE_CLIENT_NO_RESERVATION(Gi, Vi, a, client, waiting_time):
7
8       // list of pairs in the format <service, next_service>
9       // required to later acquire next_service more efficiently
10      close = Ø
11
12      // find services which may be a possible match
13      for each v ∈ Vi do
14          for each s ∈ services(v) do
15
16              // time diff from arrival of new client to start of service
17              time_diff = start(s) - arrival(client)
18
19              // is at correct time and has vacant spots
20              if 0 <= time_diff <= waiting_time AND vacant(s) >= num_people(client) then
21                  next = NEXT_SERVICE(services(v), s)
22
23                  // if there is no service after this one, can always integrate
24                  if next == NULL then
25                      NEW_DEST_INTEGRATION(Gi, a, client, s)
26                      return
27
28                  ENLIST(close, PAIR(s, next))
29
30      // find services which already go through same destination
31      for each pair ∈ close do
32          s = first(pair)
33          for each r ∈ reservations(s) do
34              if dest(r) == dest(client) then
35                  SAME_DEST_INTEGRATION(client, s)
36                  return
37
38      // find services which are not affected by adding client
39      for each pair ∈ close do
40          s = first(pair)
41          next = second(pair)
42          path_vertices = GET_SERVICE_DESTINATIONS(Gi, s)
43          INSERT(path_vertices, FIND_VERTEX(Gi, dest(client)))
44
45          path_edges = CALCULATE_PATH(Gi, a, path_vertices)
46          total = TRAVEL_TIME(path_edges)
47
48          if start(service) + total <= start(next) then
49              NEW_DEST_INTEGRATION(Gi, a, client, s)
50              return
```

```
// Gi - graph
// a - airport
// c - client to integrate
// s - service in which to integrate client
NEW_DEST_INTEGRATION(Gi, a, c, s):

    // determine set of vertices the service must go through
    vertices = GET_SERVICE_DESTINATIONS(Gi, s)
    INSERT(vertices, FIND_VERTEX(Gi, dest(c)))

    // recalculate path, now going through new destination
    path_edges = CALCULATE_PATH(Gi, a, vertices)
    total = TRAVEL_TIME(path_edges)

    // update service info
    PUSH_BACK(reservations(s), c)
    path(s) = path_edges
    end(s) = start(s) + total
    vacant(s) -= num_people(c)
```

```
// Gi - graph
// a - airport
// c - client to integrate
// s - service in which to integrate client
// path - path going throught new vertex
// total - time of new path
NEW_DEST_INTEGRATION(Gi, a, c, s, path, total):

    // update service info
    PUSH_BACK(reservations(s), c)
    path(s) = path_edges
    end(s) = start(s) + total
    vacant(s) -= num_people(c)
```

```
1   // c - client to integrate
2   // s - service in which to integrate client
3   SAME_DEST_INTEGRATION(c, s):
4
5       // find reservation with same destination and add client after it
6       for each r ∈ reservations(s) do
7           if dest(r) == dest(c) then
8               INSERT_AFTER(reservations(s), c, r)
9               break
10
11      vacant(s) -= num_people(c)
```

Análise da Complexidade

SameDestIntegration

Começando pela análise do algoritmo **SAME_DEST_INTEGRATION**, este irá percorrer todas as reservas de um serviço e, caso encontre alguma cuja destino seja igual ao destino do novo cliente, insere o novo cliente imediatamente após. O número de reservas num serviço será sempre muito inferior ao número total de reservas, pelo que se considera |r| (número médio de reservas por serviço) como diferente de |R| (número total de reservas). Assim, uma vez que a operação de inserção no vetor, que toma tempo O(|r|), apenas é executada uma vez, a complexidade deste algoritmo é O(|r|).

NewDestIntegration

O algoritmo **NEW_DEST_INTEGRATION** é dominado pela necessidade de recalcular o caminho do serviço, tendo em conta um novo destino. Assim, esta operação tomará tempo **O(|v| * (|V| + |E|) * log(|V|))**.

Caso o caminho tenha já sido calculado, como acontece no algoritmo final utilizado, definimos um overload da função **NEW_DEST_INTEGRATION** que toma este caminho como argumento, não sendo necessário recalcular. Assim, toma tempo constante O(|r|), devido à inserção no vetor de reservas.

IntegrateClientNoReservation

O algoritmo final, **INTEGRATE_CLIENT_NO_RESERVATION**, inicia percorrendo todos os serviços de todas as carrinhas (|S|), procurando possíveis candidatos que possam integrar o novo cliente e guardando-os numa lista (inserção em O(1)).

Seguidamente, para cada serviço nesta lista de tamanho máximo (|S|), são procuradas reservas que possuam o mesmo destino do novo cliente. No pior caso, esta

fase terá que percorrer todas as reservas (|R|). Caso encontre alguma, o cliente é adicionado à reserva usando o SAME_DEST_INTEGRATION (executado apenas uma vez, de complexidade O(|r|)) e o algoritmo termina. Caso termine nesta fase, a complexidade seria então de O(|R|).

A última etapa do algoritmo requer percorrer todos os serviços identificados como possíveis candidatos (|S|) e verificar se algum deles é realmente válido. Assim, para cada serviço, será calculado o caminho tendo em conta a adição do novo cliente, usando o CALCULATE_PATH. Esta operação exigirá tempo O(|v| * (|V| + |E|) * log(|V|)). Seguidamente, caso seja uma combinação válida, o passageiro é integrado, tomando tempo O(|r|). Assim, esta última fase possui complexidade O(|S| * |v| * (|V| + |E|) * log(|V|)).

Na sua totalidade, a complexidade do algoritmo no pior caso será então **O(|R| + |S| * |v| * (|V| + |E|) * log(|V|))**.

Função objetivo

Pseudocódigo

```
1  // Vi - list of vans with services assigned
2  OBJECTIVE_FUNCTION(Vi):
3      total = 0
4      for each v ∈ Vi do
5          for each s ∈ services(v) do
6              for each r ∈ reservations(s) do
7                  total += (deliver(r) - arrival(r))
8      return total
```

Análise da Complexidade

Este algoritmo exige que sejam percorridos todos os veículos, todos os serviços de cada veículo e todas as reservas de cada serviço. No entanto, visto cada reserva estar atribuída a um único serviço e, por sua vez, cada serviço estar apenas atribuído a uma carrinha, na totalidade serão simplesmente percorridas todas as reservas uma única vez cada. Assim, a complexidade deste algoritmo é O(|R|), onde |R| é o número de reservas atribuídas.

Nota: A análise empírica dos algoritmos foi realizado num computador Legion y520 com o processador Intel i7-7700hq com 8gbs de ram e o OS: Ubuntu 18.04.2.

Casos de utilização

A aplicação a implementar contém uma interface simples de texto para interagir com o utilizador. Para tal, utiliza um sistema de menus com as várias opções a serem disponibilizadas.

Permite, também, guardar em memória um mapa (sob a forma de grafo), carregado pelo utilizador, no qual incidirão as diferentes funcionalidades da aplicação. Destas destacam-se:

- visualização do grafo representante do mapa através do GraphViewer;
- verificação dos dados de cada serviço de cada carrinha e visualização do respetivo caminho no GraphViewer.

Conclusão

A situação em questão, o planeamento das deslocações de uma empresa de transfers, foi analisada com sucesso, tendo sido decomposta em quatro iterações que visam aproximar o problema com um nível de complexidade incremental. A última iteração destaca-se das restantes por introduzir funcionalidades adicionais ao planeamento das rotas, não sendo essencial.

Foram identificados os vários problemas inerentes ao tema, tendo sido expostos com detalhe e apresentadas propostas de solução. Estas foram baseadas nas ideias e algoritmos discutidos nas aulas teóricas de CAL, assim como em conteúdos fora do âmbito da unidade curricular.

Implementamos as várias iterações que nos propusemos a fazer na primeira parte do trabalho, considerando a sua complexidade teórica e empírica. Para este fim, foram consideradas várias **variações** nos algoritmos implementados, nomeadamente na forma de agrupamento de passageiros e no cálculo de caminhos entre vértices. Foi então feito um estudo de, face um problema real similar ao nosso contexto, quais as técnicas de agrupamento e de cálculo de percurso mais ótimas.

Destaca-se a utilização de um grafo para representar a rede de estradas, tendo sido esta estrutura de dados a base dos algoritmos aplicados. Surgiram, portanto, vários problemas relacionados com esta estrutura, salientando-se o **caminho mais curto entre dois vértices**, o **caminho mais curto entre todos os pares de vértices**, o **caminho mais curto passando por uma sequência de vértices** e **análise da conectividade**.

Ao longo das quatro iterações consideradas, foram identificados dois problemas NP-difíceis que se assemelham ao problema em questão, o ***Travelling Salesman Problem*** e o ***Vehicle Routing Problem***.

Salienta-se que, conforme a situação e o problema a resolver, foram estudados e projetados algoritmos da autoria do grupo, ou foram adaptados algoritmos já existentes. Destes últimos, destacam-se: **DFS**, **Dijkstra**, **A***, **Floyd-Warshall**, **Nearest Neighbour**, **Held-Karp**, **Kosaraju**, **Tarjan**.

Foram também utilizados vários conceitos relevantes na área da concepção de algoritmos, dos quais: ***bruteforce,*** **recursividade**, **programação dinâmica**, **algoritmos gananciosos**, **divisão e conquista**, **retrocesso**.

Nesta última fase foram sentidas várias dificuldades relacionadas com os mapas de estradas fornecidos. Devido à sua conectividade, vimo-nos forçados a admitir o grafo não dirigido, alterando os algoritmos que estavamos a considerar implementar. Além disso, deparamo-nos com vários problemas durante a análise dos algoritmos porque, ao aplicar o pré-processamento, removendo vértices não alcançáveis, a dimensão do grafo reduzia

consideravelmente. Obtivemos inicialmente resultados pouco adequados e mesmo disparatados, até nos apercebermos que o problema estava nos mapas fornecidos.Com efeito, estes problemas provocaram atrasos na realização do trabalho, impedindo-nos de aplicar variações aos algoritmos implementados e comparar com os algoritmos originais.

É de notar que, durante a implementação, ocorreram-nos várias ideias de como aprimorar os algoritmos, tal como a **otimização dos parâmetros inseridos** para que o programa encontre uma solução cada vez melhor. Por exemplo, para aumentar o agrupamento de reservas por carrinha, podia-se aumentar a **janela de tempo** para encontrar reservas ou até o **raio** à volta da primeira reserva para encontrar mais reservas para irem na mesma carrinha. Outro ponto seria a **quantidade de carrinhas**, aumentando o número de carrinhas dá-se a possibilidade de as pessoas terem de esperar menos, mas aumenta a probabilidade de desperdícios. Verificamos que, por vezes, consegue-se que a solução tenha o mesmo resultado usando um número mais baixo de carrinhas.

Face aos resultados da análise empírica, concluímos que a segunda iteração surge como a mais eficiente em termos de organização das carrinhas, seguida da terceira e finalmente da primeira (a nível da função objetivo). Estes resultados podem ser resultantes da função objetivo não ser a mais adequada a este contexto, devendo ter sido considerado pesos ou fatores de penalização no seu cálculo. Os resultados obtidos estão disponíveis em https://docs.google.com/spreadsheets/d/1MQD5Zjlb52lD_kDbeXiAfO8Paz3D666BcqBKX5sMhJg/edit?usp=sharing .

Cada membro do grupo dedicou um esforço similar ao projeto, tendo a divisão das tarefas sido feita da seguinte maneira:

- **Mário Mesquita:** Descrição do problema, Função objetivo, Pré-processamento dos dados de entrada, Identificação dos problemas encontrados, Algoritmo de Dijkstra, Percurso de duração mínima entre todos os pares de pontos, Organização das várias carrinhas conforme as reservas, Análise da conectividade, Conclusão, Revisão e reescrita do relatório, Pseudocódigo, Implementação da 1ª iteração, Implementação da 4ª iteração, Análises de complexidade teórica.

- **Moisés Rocha:** Dados de entrada, Dados de saída, Restrições, Caminho de duração mínima passando por vários destinos, Organização das várias carrinhas conforme as reservas, Integração de passageiros sem reserva, Possibilidade de viagens de residências ao aeroporto, implementação da terceira iteração, interface.

- **Paulo Marques:** Dados de entrada, Restrições dos dados, Função objetivo, Percurso de duração mínima entre dois pontos, Casos de utilização, Manutenção das referências bibliográficas, Revisão e reescrita do relatório. Implementação da segunda iteração, análise empírica dos algoritmos implementados, implementação do algoritmo A*, conclusão.

Bibliografia

- Slides fornecidos no âmbito da cadeira de Conceção e Análise de Algoritmos pelos professores: R. Rossetti, L. Ferreira, L. Teófilo, J. Filgueiras, F. Andrade
- Thomas H. Cormen... [et al.]; Introduction to algorithms. ISBN: 978-0-262-53305-8
- Dijkstra's Algorithm, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- Brilliant - Dijkstra's Algorithm, https://brilliant.org/wiki/dijkstras-short-path-finder/
- A* Search Algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm
- Computerphile – Dijkstra's Algorithm, https://www.youtube.com/watch?v=GazC3A4OQTE
- Computerphile – A* (A Star) Search Algorithm, https://www.youtube.com/watch?v=ySN5Wnu88nE
- Tarjan's Algorithm, https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
- Kosaraju's Algorithm, https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm
- Floyd-Warshall Algorithm, https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- Travelling Salesman Problem, https://en.wikipedia.org/wiki/Travelling_salesman_problem
- Christian Nilsson - Heuristics for the Traveling Salesman Problem, https://web.tuke.sk/feicit/butka/hop/htsp.pdf
- Amanur Rahman Saiyed - The Traveling Salesman problem, http://cs.indstate.edu/~zeeshan/aman.pdf
- Consistent Heuristics, https://en.wikipedia.org/wiki/Consistent_heuristic
- Vehicle Routing: https://en.wikipedia.org/wiki/Vehicle_routing_problem
- Dantzig, George Bernard; Ramser, John Hubert (October 1959) https://andresjaquep.files.wordpress.com/2008/10/2627477-clasico-dantzig.pdf
- GeeksForgeeeks Strongly Connected Components: https://www.geeksforgeeks.org/strongly-connected-components/
- Aplicação em java para a comparação de Dijkstra com A*:
- https://github.com/kevinwang1975/PathFinder
- GraphViewer: http://jung.sourceforge.net/