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, 26 de abril 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 pode ser 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 dirigido pesado, composto por:

- V - vértices, representando pontos da rede viária, com:
  - ID - identificador do vértice
  - Adj ⊆ E, arestas que partem do vértice.
  - Latitude -Coordenada real do ponto no mapa
  - Longitude - Coordenada 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)
  - 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 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).

Funções 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:

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

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 a serem desenvolvidos. É providenciado pseudocódigo apenas para os algoritmos por nós desenvolvidos, visto já existirem várias implementações para os restantes.

Pré-Processamento dos Dados de Entrada

Pré-processamento do grafo

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

Primeiramente, devem ser 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 devem também ser eliminados.

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

Pré-processamento das reservas

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

Adicionalmente, as reservas em Ri cujo valor de passageiros ultrapasse a capacidade dos veículos (Cv) devem ser 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 poderá ser totalmente satisfeita por uma só carrinha.

Finalmente, Ri deve ser 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.

Pré-cálculo dos percursos de duração mínima

Será 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, a utilização destes algoritmos será computacionalmente exigente, uma vez calcular a distância mínima entre todos os pares de vértices e não apenas entre os vértices interessantes para determinado percurso.

Assim, a aplicabilidade ou não destes algoritmos requer uma análise da eficiência do programa tendo em conta a dimensão do mapa considerado.

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***. Visto o grafo ser dirigido, é uma versão assimétrica do problema.

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, tentaremos 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.

Algoritmo de Dijkstra

Este algoritmo, concebido por Edsger W. Dijkstra em 1956, soluciona o problema do caminho mais curto num grafo dirigido (como é o caso) 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:

- dur - duração mínima até à origem
- path - vértice anterior no caminho mais curto

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|)).

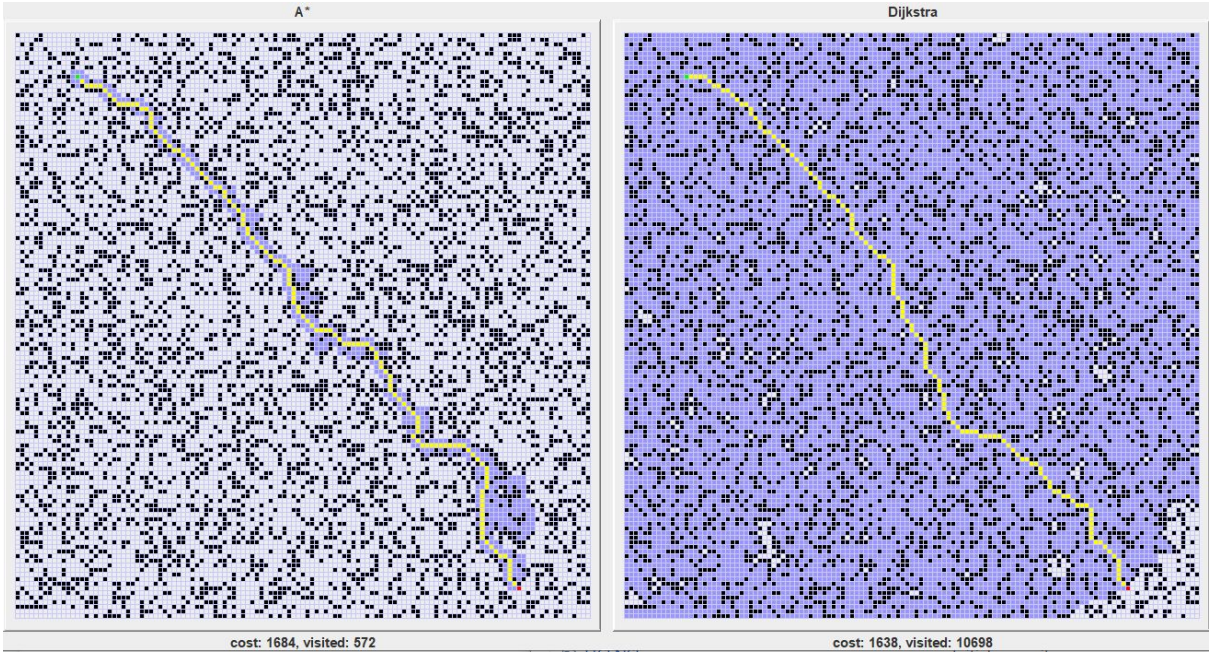
Algoritmo A*

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.

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.

A implementação deste algoritmo será considerada na próxima fase do projeto.

Percurso de duração mínima entre todos os pares de pontos

Como mencionado na fase do pré-processamento, surge a hipótese de pré-calcular os percursos de duração mínima entre todos os pares de vértices, aumentando significativamente a eficiência do restante programa.

Para este efeito serão consideradas duas abordagens, a aplicação do método de Dijkstra para cada vértice e o algoritmo de Floyd-Warshall. Serão ambas estudadas relativamente à sua eficiência.

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. Visto o grafo ser dirigido, é uma versão assimétrica do problema.

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.

Pseudocódigo

```
FIND_FIRST_ARRIVAL(s):
    min = ∞
    for c ∈ B(s) do
        if arrival(c) < min then
            min = arrival(c)
    return min


// Ci-sequence of vans;
// r-reservation looking for a service;
// Tm-time window to wait for a van to be filled
FIND_SEMI_FILLED_SERVICES(Ci, r, Tm):
    services = ∅
    for c ∈ Ci do
        for s ∈ S(c) do
            f = FIND_FIRST_ARRIVAL(s)
            if (vacant(s) >= num(r)) and (f <= enter(r) <= f+Tm) then
                INSERT(services, s)
    return services


// Ri-sequence of reservations;
// Ci-sequence of vans;
// Tm-time window to wait for a van to be filled;
// Md-maximum distance between vertices considered near
ORGANIZE_VANS(Ri, Ci, Tm, Md):

    R = ∅ //sorted by 'enter' first and 'time' second

    for each r ∈ Ri do
        enter(r) = time(r)
        INSERT(R, r)

    while R != ∅ do
        r = EXTRACT-MIN(R)

        found_match = false
        serv = FIND_SEMI_FILLED_SERVICES(Ci, r, Tm) //Find semi-filled services at the time the new client arrives

        if serv != null then
            for s ∈ serv do

                ADD_CLIENT_TO_SERVICE(r, s) //add current client to service, updating all values required

                if end(s) < start(NEXT-RESERVATION(r)) or DESTINATIONS_NEAR(s, Md) then
                //addition does not interfere with next client or successive destinations are near each other
                    found_match = true
                    break

                REMOVE_CLIENT_FROM_SERVICE(r, s) //was not a valid match

        if not found_match then
            c = FIND_EMPTY_VAN(ci, enter(r))
            if c != null do
                CREATE_NEW_SERVICE(c, r)
            else
                enter(r) = NEXT_VAN_ARRIVAL(ci, enter(r))
                INSERT(R, r) //add element again to list
```

ADD_CLIENT_TO_SERVICE deverá adicionar determinado cliente de uma reserva a um serviço, atualizando todos os campos necessários desse serviço (vacant, end, …).

REMOVE_CLIENT_FROM_SERVICE deverá remover determinado cliente de um serviço, atualizando todos os campos necessários desse serviço (vacant, end, …).

FIND_FIRST_ARRIVAL encontra qual a primeira hora de chegada de todos os clientes num determinado serviço.

FIND_SEMI_FILLED_SERVICES devolve todos os serviços já existentes na lista de carrinhas Ri que poderão potencialmente transportar o cliente r, considerando uma janela de tempo para preencher cada serviço de Tm.

NEXT-RESERVATION procura a reserva seguinte à que é passada como argumento, por ordem temporal de chegada ao aeroporto.

DESTINATIONS_NEAR verifica se todos os destinos de um determinado serviço s estão, no máximo, à distância temporal de Md.

FIND_EMPTY_VAN procura uma carrinha que esteja completamente livre no momento especificado como argumento.

CREATE_NEW_SERVICE cria uma novo serviço numa determinada carrinha c para um cliente r, às horas especificados por enter(r).

NEXT_VAN_ARRIVAL determina qual a hora em que haverá uma carrinha disponível para transportar um cliente r que chegou às horas enter(r).

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.

Pseudocódigo

Sendo R um pedido de viagem sem reserva com:

- dest - destino
- time - altura em que é feito o pedido

e T quanto tempo o cliente quer esperar:

```
NO_RESERVATIONS(Cf, R, T):
    // Carrinhas com serviços com lugares vagos nos proximos momentos
    CLOSE ← ∅ //List
    for each c in Cf:
        for each s in S(c):
            if T ⩽ start(s) - time(R) ⩽ 0 && vacant(s) > 0:
                ENLIST(CLOSE,s)

    // Serviços que passam pelo destino do passageiro
    COINCIDE ← ∅  // List
    for each s in CLOSE:
        for each a in P(s):
            if dest(R) = dest(a):
                ENLIST(CLOSE, s)

    if |COINCIDE| > 0:
        return COINCIDE

    // Serviços que não são afetados pela a adição do novo cliente
    RES ← ∅
    for each s in CLOSE:
        AL ← ∅ // List
        for each b in B(s):
            ENLIST(AL, dest(b))
        ENLIST(AL, dest(R))

        N ← CALC-PATH(AL)

        if (start(s) + dur(N) ⩽ start(SERVICE-AFTER(s)) || SERVICE-AFTER(s) == NIL):
            ENLIST(RES, s)
    return RES
```

CALC-PATH() será o algoritmo que calcula o caminho ótimo passando por pontos especificados numa lista (AL). Retorna uma lista de arestas a seguir com dur, que representa o tempo que o caminho vai demorar a percorrer, junto.

SERVICE-AFTER() retorna o serviço que vai ser efetuado pela mesma carrinha a seguir ao serviço dado.

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.

Algoritmo de Kosaraju

Para encontrar todos os componentes fortemente conexos do grafo será utilizado o algoritmo 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.

Casos de utilização

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

Permitirá, 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;
- cálculo do caminho ótimo entre dois pontos;
- verificação da alcançabilidade de um determinado destino pelas carrinhas da empresa;
- geração de uma rota percorrendo um ou mais vértices;
- organização e atribuição dos serviços de uma dada lista de carrinhas, tendo em conta uma lista de reservas (as duas listas devem ser fornecidas).

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.

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: **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**.

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.

- **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.

- **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.

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