U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEIC 2º ANO

Conceção e Análise de Algoritmos

Maio 2019

City Sightseeing: trilhas turísticas urbanas

Turma 7 - Grupo G

João Praça - up201704748

Leonor Sousa- up201705377

Sílvia Rocha - up201704684

Índice

Descrição do Tema

Pretende-se implementar uma aplicação que permita a otimização de percursos de city sightseeing na cidade do Porto. Esta aplicação pretende efetuar uma melhoria ao típico serviço de turismo encontrado nas grandes cidades, onde se oferecem viagens de autocarro que passam por determinados pontos turísticos com rotas previamente definidas. O melhoramento consiste numa personalização das rotas do autocarro consoante as preferências dos clientes.

Deste modo, a aplicação implementada deverá, para um determinado número de pontos de interesse selecionados (e ruas que unem os respetivos pontos), encontrar o caminho mais curto e/ou mais rentável para a empresa em questão.

1ª Fase de Implementação

Numa primeira fase, todos os turistas serão agrupados num grupo único, efetuando a mesma rota, que passará por todos os pontos selecionados. Nesta fase, não será considerado o limite de passageiros do autocarro.

Assim sendo, nesta fase apenas se deverá calcular o caminho mais curto entre um ponto de partida e um ponto de chegada, passando por um determinado número de pontos de interesse. Este serviço é muito semelhante ao fornecido pelo “Google Maps”. O cálculo deste percurso deverá ainda ter em conta a possibilidade de uma determinada rua se encontrar em obras e, por isso, temporariamente inutilizável.

Para facilitar o desenvolvimento desta fase, esta será subdivida em três fases:

Fase 1.1

Inicialmente, apenas se irá calcular o percurso entre o ponto de partida e o ponto de chegada.

Fase 1.2

Em segunda lugar, será calculado o percurso entre o ponto de partida e o ponto de chegada, passando pelo vários pontos de interesse, mas por uma ordem específica (a ordem de entrada). (Sistema “Google Maps”)

Fase 1.3

Em último lugar será calculado o percurso entre o ponto de partida e o ponto de chegada, passando pelos vários pontos de interesse, pela ordem que permite que o percurso seja o mais curto possível.

2ª Fase de Implementação

Numa segunda fase, os turistas deverão ser agrupados em pequenos grupos de acordo com similaridade de pontos de interesse. Deste modo, é atribuído um autocarro diferente a cada grupo de turistas, devendo ser otimizados os percursos para cada grupo de turistas.

Nesta fase, o limite de turistas por autocarro continua a ser ignorado.

3ª Fase de Implementação

Na última fase de implementação, acrescenta-se a restrição associada ao limite de passageiros de cada autocarro.

Assim, os grupos de turistas serão também otimizados tendo em conta, por um lado, a similaridade de pontos de interesse, e por outro, a relação entre o tamanho do grupo com o limite de passageiros. Esta restrição tem um enorme importância na gestão do serviço: alocar-se pequenos grupos de turistas por uma grande quantidade de autocarros em vez de alocar grupos maiores a uma menor quantidade de autocarros pode resultar numa solução não rentável por dois motivos: a distância percorrida no total é maior ou, apesar da distância até ser menor, implica o pagamento a um número maior de motoristas, resultando num maior prejuízo para a empresa. Deste modo, esta fase implica a minimização não só da distância percorrida pelos vários autocarros, mas também do número de autocarros utilizados.

Formalização do Problema

Dados de Entrada

- Gmin - Tamanho mínimo de um grupo, a partir do qual é rentável transportá-lo-
- B - sequência de autocarros disponíveis para um serviço, sendo Buses[i] o seu i-ésimo elemento; cada elemento é caracterizado por um id, pela sua capacidade total (na 1ª e 2ª implementação, a capacidade é igual a ∞).
- G - grafo dirigido pesado, composto por:
    - V - vértices, que representam os pontos de interesse; caracterizam-se por um identificador e por Adj ⊆ E (conjunto de arestas que partem do vértice);
    - E - arestas, que representam as várias ruas que unem os pontos de interesse; caracterizam-se por um identificador, pelo seu peso (distância entre os pontos que une) e por dest ∈ V (vértice de destino).
- S ∈ V - vértice inicial
- T ∈ V - vértice final
- Tourists - sequência de turistas/clientes que utilizarão o serviço, sendo Tourists[i] o seu iésimo elemento; cada elemento é caracterizado por um id e por um conjunto de pontos de interesse, representados por um identificador.
- EI - sequência de arestas indisponíveis; representam as ruas que estarão em obras;

Dados de Saída

- Q[O] - sequência dos vários percursos que deverão ser efetuados por cada autocarro; cada percurso (O) corresponde a uma sequência ordenada dos vértices (pontos de interesse) que contém todos os vértices indicados pelos turistas que vão realizar essa viagem; na primeira fase, size(Q)=1.
- A - sequência de autocarros utilizados para efetuar os percursos selecionados, sendo A[i] o seu i-ésimo elemento; o autocarro A[i] é alocado ao percurso Q[i].
- P - peso total das arestas visitadas na sequência Q (distância total da viagem)

Restrições nos dados de entrada

- ∀ e ∈ E, peso(e) ≥ 0 -> O peso (distância entre dois pontos de interesse) de uma aresta representa uma grandeza positiva pelo que o seu valor não poderá ser inferior a 0.

- Gmin ≥ 0 -> o tamanho de um grupo é sempre um número positivo ou nulo.
- size(Tourists) ≥ Gmin
- size(B) ≥ 1 - a empresa tem que ter, pelo menos, um autocarro.
- S,T ∈ V - os vértices inicial e final tem que pertencer aos vértices do grafo
- ∀ e ∈ EI, e ∈ E
- O grafo G deverá ser fortemente conexo -> cada vértice (ponto de interesse) deverá ter uma aresta (rua) que permita chegar a esse vértice e uma outra aresta (rua) que permita abandonar esse vértice. Simultaneamente, a partir de qualquer grafo deverá ser possível chegar a todos os outros grafos.
- O grafo G deverá ser conexo o suficiente de forma a permitir que, ao remover todas as arestas pertencentes a EI, ele permaneça fortemente conexo.

Restrições nos dados de saída

- P ≥ 0 -> Em consequência da restrição apresentada nos dados de entrada, também a soma dos pesos das arestas visitadas terá de ser um valor maior ou igual a 0.
- O[0] = S -> O primeiro vértice da sequência O tem de corresponder ao vértice S, o ponto de partida da viagem.
- O[i] = T -> O último vértice da sequência O tem de corresponder ao vértice T, o ponto de chegada da viagem.
- ∀ p ∈ O, p ∈ V - Todos os vértices da sequência O têm de constar no grafo que representa o mapa da cidade.
- ∀ a ∈ A, a ∈ B - Todos os autocarros da sequência A têm de constar no conjunto de autocarros possuídos pela empresa (B).
- 1≤ size(A) ≤ size(B) - Não podem ser alocados mais autocarros do que aqueles disponíveis na empresa.
- ∀ A[i],A[j] ∈ A, se i≠j então A[i] ≠ A[j] -> A não pode conter autocarros repetidos
- size(A) = size(Q) -> deverão ser alocados tantos autocarros quantos os percursos a serem efetuados.

Função objetivo

A solução ótima para o problema já descrito no presente relatório passa por obter o percurso mais curto (rentável) entre S e T (vértice inicial e final) passando por todos os vértices indicados, ou seja, o que tenha P, peso total, menor. Deverá ser também minimizado o número

de autocarros utilizados (no caso das últimas duas fases de implementação). Deste modo, o objetivo será a minimização das seguintes funções:

f = P

g = size(A)

A minimização da função g terá prioridade.

Perspetiva da Solução

Estruturas de Dados

Grafo

Para representar um grafo será utilizada a estrutura de dados contida em “Graph.h”, ficheiro previamente fornecido pelo professor da unidade curricular, através da plataforma moodle.

Esta estrutura de dados (classe “Graph”) é composta por um vetor de vértices e por um conjunto de métodos que permitem adicionar e remover vértices, adicionar e remover arestas (funcionalidade útil na questão das obras), pesquisar um determinado elemento, entre outros.

Cada um dos vértices, representados pela classe “Vertex” é caracterizado pelo conjunto de arestas que dele partem e por um conjunto de outros atributos, úteis nos algoritmos que são aplicados no grafo. Cada vértice contém também métodos para remover ou adicionar arestas.

Por fim, cada aresta, representada pela classe “Edge”, contém um atributo que representa o seu peso e um outro que representa o seu vértice de destino.

Bus

Será também implementada uma classe Bus, que terá como atributos:

- int id;
- int capacidade;
- vector<Tourist *> tourists;

E ainda os seguintes métodos:

- void addTourist(Tourist * t);
- void removeTourist(Tourist * t);

Tourist

Incluiremos ainda uma classe Tourist, que terá como atributos:

- int id;
- string nome;
- vector<Node *> pois;

E ainda os seguintes métodos:

- void addPoi(Node * n);

- void removePoi(Node * n);

Empresa

Por fim, será implementada uma classe Empresa, que terá como atributos:

- vector<Bus*> buses;
- vector<Tourist*> tourists;
- vector< vector<Node*> > routes;

E ainda os seguintes métodos:

- void addBus(int id, int capacidade);
- void removeBus(int id);
- void addTourist(int id, string nome);
- void removeTourist(int id);

Se necessário, durante a implementação do projeto, poderão ser adicionados e/ou alterados alguns atributos ou métodos das classes descritas. No entanto, nesta fase, esta implementação parece-nos suficiente para o projeto em questão.

Principais Algoritmos

Fase 1

Fase 1.1

Para implementação desta fase, será utilizado o Algoritmo de Dijkstra para calcular caminho mais curto entre o vértice inicial e o vértice final.

Fase 1.2

Nesta fase, o percurso de vários pontos será dividido em vários subpercursos de 2 pontos apenas, sendo que a cada um destes subpercursos será aplicado o algoritmo desenvolvido na fase 1.1.

Para melhor perceber esta situação, veja-se o seguinte exemplo: para calcular o melhor percurso A->B->C->D->E, calcular-se-ão os melhores percursos A->B, B->C, C>D e D->E, sendo que a resposta final seria a concatenação das várias sub-respostas obtidas.

Fase 1.3

Por fim, nesta fase, aplicar-se-ão métodos combinatórios para calcular o percurso sem uma ordem de pontos intermédios específica.

Para o exemplo acima, em que tínhamos o percurso de A para E, passando por B, C e D, serão testados as seguintes ordens: A->B->C->D->E, A->B->D->C->E, A->C>B->D->E, A->C->D->B->E, A->D->B->C->E e A->D->C->B->E, recorrendo ao algoritmo da fase 1.2. Será então selecionada a melhor opção (a que tiver menor peso) de entre as 6 obtidas.

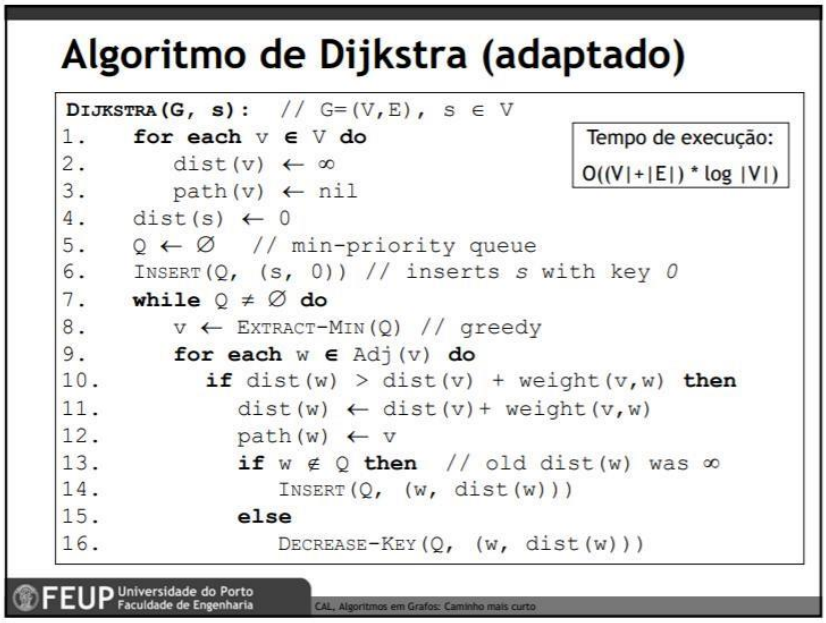
No entanto, este algoritmo utiliza uma abordagem de força bruta, uma vez que avalia todos os caminhos possíveis e retorna o mais curto. Com recurso a programação dinâmica, podemos reduzir o número de combinações de vértices, eliminando aquelas que não se adequam à melhor solução. Assim, o melhor caminho será calculado por etapas, combinando resultados obtidos para partes menores do problema principal, previamente calculados e guardados.

O algoritmo será desenvolvido tendo como base o pseudocódigo seguinte, sendo S o conjunto de vértices a visitar (inclui vértice inicial, POIs e vértice final) e C(S, j) o comprimento do caminho mais curto que passa pelos vértices de S apenas uma vez, começando no vértice inicial e terminando em j, o vértice final.

```
Os problemas são ordenados por |S|:

   C({1},1) = 0:
   Para s = 2 até n:
           Para todos os subconjuntos S ⊆ {1,2,...,n} de tamanho s e contendo 1:
               C(S,1) = ∞:
               Para todo j ∈ S, i ≠ j:
                      C(S,j) = min{C(S - {j},i) + d_ij: i ∈ S, i ≠ j}
   Retornar min_j C({1,...,n},j) + d_ij
```

Para toda esta fase (1), aplica-se ainda a possibilidade de utilizar o Algoritmo A*, visto que poderá melhorar a rapidez do processo, rapidez esta necessária num algoritmo que por si só já se pode revelar bastante demorado.



Pseudocódigo para Algoritmo de Dijkstra - Retirado dos Slides fornecidos pelo Professor da unidade curricular, na página moodle da mesma

Fase 2

Esta fase não implica grandes cálculos a nível algorítmico. Os grupos de turistas deverão ser feitos de acordo com a similaridade dos pontos de interesse. Deste modo, serão efetuados vários vetores de turistas, em que, para cada vetor, todos os turistas possuem a mesma lista de pontos de interesse a visitar. A cada vetor de turistas corresponde um vetor de pontos de interesse (com um ponto de partida e de chegada já definidos), ao qual depois será aplicado o algoritmo da fase 1.

Apesar de ser uma opção ainda em análise, poderá ser aplicada uma tolerância na igualdade de pontos de interesse. A tolerância será aplicada da seguinte forma:

A tolerância deverá ser calculada para cada caso consoante um número mínimo de pontos de interesse (por exemplo 5). Desta forma se o primeiro turista a ser adicionado a um determinado vetor (autocarro/percurso) tiver 3 pontos de interesse, a tolerância é decrementada em 3 (tolerância = 2). Deste modo, um próximo turista só poderá ser adicionado a este mesmo vetor, se o seu percurso não contiver mais do que 2 pontos de interesse diferentes (e a mais) do que os do primeiro turista. A tolerância vai sendo decrementada à medida que se adicionam turistas com novos pontos de interesse. Quando a tolerância atinge o valor 0, só são permitidos turistas com uma totalidade de pontos de interesse já selecionados para esse percurso.

Fase 3

Nesta fase e, depois de serem efetuados os grupos da fase 2, serão, mais uma vez agrupados os grupos calculados, se tal for necessário para minimizar o número de autocarros necessários. Deste modo, poderão surgir dois tipos de situações nos grupos:

1. Grupo muito grande, não cabe em qualquer autocarro disponível: será necessário dividir o grupo; uma divisão eficaz passaria pelo novo cálculo de semigrupos usando o algoritmo da fase 2 com uma tolerância menor.
2. Grupos muito pequenos, precisam de um número demasiado elevado de autocarros comparado com o número de turistas: será necessário agrupar os grupos; uma divisão eficaz passaria pelo novo cálculo de semigrupos usando o algoritmo da fase 2 com uma maior tolerância.

Casos de Utilização

Adicionar um turista

Esta funcionalidade permite adicionar um novo turista à lista de turistas para que ele possa ser incluído nos cálculos da divisão em grupos e dos melhores percursos.

Remover um turista

Esta funcionalidade permite remover um turista.

Adicionar um autocarro à frota da empresa

Esta funcionalidade permite a adição de um novo veículo à frota da empresa.

Remover um autocarro da frota da empresa

Permite a remoção de um veículo da frota da empresa.

Adicionar pontos de interesse a um determinado turista

Esta funcionalidade permite que um turista já inserido possa acrescentar alguns pontos de interesse posteriormente.

Remover pontos de interesse de um determinado turista

Esta funcionalidade permite que um turista já inserido possa remover alguns pontos de interesse posteriormente.

Assinalar uma determinada rua como estando em obras

Esta funcionalidade permite informar o sistema de cálculo do percurso que uma determinada rua se encontra inutilizável.

Assinalar as obras numa determinada rua como terminadas

Esta funcionalidade permite informar o sistema de cálculo do percurso que uma determinada rua se encontra novamente utilizável.

Cálculo do caminho mais curto/rentável sem grupos

Esta funcionalidade irá permitir o cálculo do caminho mais rentável entre dois pontos, passando por vários POIs, sem alocação de turistas em grupos.

Cálculo do caminho mais curto/rentável com grupos sem limite

Esta funcionalidade do programa irá permitir calcular a divisão dos diferentes grupos de turistas de acordo com os POIs escolhidos, sem número limite de lugares em cada autocarro, e o cálculo do caminho mais rentável em cada um dos casos.

Cálculo do caminho mais curto/rentável com grupos limitados

Esta será a principal funcionalidade do programa que irá permitir calcular a divisão dos diferentes grupos de turistas, tendo em conta a capacidade dos autocarros disponíveis e os POIs escolhidos, e o cálculo do caminho mais rentável em cada um dos casos.

Discussão sobre estruturas de dados utilizadas

As estruturas de dados planeadas na 1ª parte do projeto foram implementadas com sucesso no programa apresentando apenas algumas alterações relativamente ao descrito.

A primeira alteração efetuada foi relativamente à estrutura do Grafo. A intenção inicial era utilizar o ficheiro Graph.h que nos tinha sido fornecido previamente, no entanto, concluímos que a estrutura de dados como se encontrava era incompleta face ao que necessitávamos. Deste modo, foram acrescentados alguns atributos e métodos. De entre eles, as funções necessárias para a implementação dos algoritmos de Dijkstra e A* para o cálculo do caminho mais curto.

Uma outra alteração foi na classe Bus que não tem o atributo capacidade, uma vez que foi definido que todos os autocarros teriam a mesma capacidade, passando assim este atributo para a classe Company.

Para além disso, foram implementados vários métodos na classe Company fundamentais para o correto funcionamento do programa, nomeadamente para cálculo de rotas, leitura de vértices e arestas de ficheiros, criação do grafo e procura de pontos de interesse/vértices no mesmo.

Foi também adicionada uma nova classe, PoI, que representa os pontos de interesse, e que tem como atributos id, x, y e type, que correspondem, respetivamente, ao id identificativo do ponto de interesse, as suas coordenadas no mapa, e o tipo de ponto associado. São os apontadores dos objetos desta classe que são guardados, como info, nos vértices do grafo.

Por fim, foram adicionadas todas as classes e ficheiros fornecidos relacionados com o GraphViewer, que permite ao utilizador visualizar os caminhos calculados pelo programa.

Discussão sobre a conectividade do grafo

Na análise efetuada inicialmente, aquando da primeira entrega deste trabalho, foi mencionada a importância de usar grafos fortemente conexos reforçando ainda que mesmo após a operação de adição de obras em algumas das estradas isso teria de ser possível. No entanto, julgamos que os grafos fornecidos não cumprem estes critérios.

Nos grafos fornecidos, o caso mais comum é que cada vértice tenha uma ou nenhuma aresta adjacente. Este facto dificultou claramente a fase de testes dado que se só existe um caminho desde A até B dificilmente se consegue verificar o correto funcionamento de algoritmos para cálculo do caminho mais curto, por exemplo.

Deste modo, implementámos um pequeno ficheiro de teste que consideramos ter a conectividade necessária para o correto funcionamento do projeto. Neste ficheiro cada vértice possui várias arestas adjacentes o que já obriga à análise dos vários caminhos possíveis e determinação do melhor de entre estes.

Discussão sobre os principais casos de uso implementados

Os principais usos do programa planeados na 1ª parte do trabalho foram implementados nesta 2ª parte, com exceção da opção “Cálculo do caminho mais curto/rentável com grupos sem limite”, uma vez que consideramos não se tratar de uma situação realista e relevante para a demonstração do programa.

As primeiras oito operações são relativas à empresa e à gestão dos dados da mesma: adicionar e remover turistas, autocarros e pontos de interesse, e ainda assinalar obras em determinadas ruas, isto é, remover certas arestas do grafo.

A opção 9, “Cálculo do caminho mais curto/rentável sem grupos”, devolve ao utilizador um caminho que inicia e termina em pontos previamente indicados, passando por todos os pontos de interesse associados aos turistas inscritos na empresa, sem qualquer restrição de associação em grupos. Esta opção utiliza um algoritmo que calcula todas as possibilidades de caminho, isto é, todas as combinações de pontos de interesse, e seleciona aquele cuja distância total é menor.

A opção 10, “Cálculo do caminho mais curto/rentável com grupos limitados”, devolve ao utilizador vários caminhos, que iniciam e terminam em pontos previamente indicados, representativos da rota de cada autocarro. Os turistas são agrupados pela similaridade de pontos de interesse, e posteriormente é calculado o melhor caminho para cada autocarro, passando por todos os pontos de interesse dos turistas que nele viajam.

Para além disso, adicionámos uma 11ª opção, “Calculo do caminho mais curto/rentável sem grupos usando algoritmo ganancioso”, com o mesmo propósito da 9, mas utilizando um algoritmo que utiliza programção dinâmica mas que é ganancioso. Esta opção foi adicionada com o objetivo de demonstrar a diferença de eficiência temporal entre o algoritmo de força bruta e o algoritmo ganancioso.

As três opções anteriores utilizam como algoritmo para encontrar o melhor caminho entre 2 pontos o algoritmo de Dijkstra. No entanto, como também foi implementado o algoritmo A*, adicionámos três novas opções (12, 13 e 14), que têm, respetivamente, o mesmo objetivo das opções 9, 10 e 11, mas com recurso a este novo algoritmo. Desta forma, torna-se também mais fácil comparar a influência que o algoritmo A* tem na eficiência temporal do programa em relação ao algoritmo de Dijkstra.

Apresentação dos algoritmos implementados

Inicialmente, para o cálculo do melhor caminho entre dois pontos, foram implementados os algoritmos de Dijkstra e A*:

```
void Graph<T>::dijkstraShortestPath(const T &origin) {
    auto s = initSingleSource(origin);

    MutablePriorityQueue<Vertex<T>> q;
    q.insert(s);

    while (!q.empty()) {
        auto v = q.extractMin();
        for (auto e : v->outgoing) {
            auto oldDist = e->dest->dist;
            if (relax(v, e->dest, e->weight)) {
                if (oldDist == INF)
                    q.insert(e->dest);
                else
                    q.decreaseKey(e->dest);
            }
        }
    }
}
```

```
void Graph<T>::AStarShortestPath(const T &origin, const T &dest) {
    auto s = initSingleSourceAStar(origin, dest);
    MutablePriorityQueue<Vertex<T>> q;
    q.insert(s);
    while (!q.empty()) {
        auto v = q.extractMin();
        if(v->getInfo() == dest)
            return;
        for (auto e : v->outgoing) {
            auto oldDist = e->dest->dist;
            if (relaxAStar(v, e->dest, e->weight, dest)) {
                if (oldDist == INF)
                    q.insert(e->dest);
                else
                    q.decreaseKey(e->dest);
            }
        }
    }
}
```

Para o cálculo de rotas com vários pontos foram elaborados 2 algoritmos muito diferentes.

O primeiro algoritmo utiliza força bruta para calcular todas as combinações de pontos possíveis, selecionando no fim a opção com um menor peso total. Apesar de ser um algoritmo com grande complexidade temporal e espacial, é o úncio que nos garante a resposta perfeita para o problema.

```
getPaths(auxp, vectorAux);
for (size_t i=0; i<vectorAux.size(); i++)
{
    vectorAux[i].insert(vectorAux[i].begin(), point1);
    vectorAux[i].insert(vectorAux[i].end(), point2);
    vector<PoI*> p= calculateRouteWithOrderedPoints(vectorAux[i], aStar);
    if(p.size()>0)
        resps.push_back(p);
}
```

```
void Company::getPaths(const vector<PoI*>& v, vector<vector<PoI*> > &res)
{
    vector<PoI*> aux = v;
    sort(aux.begin(), aux.end());
    do
    {
        res.push_back(aux);
    }
    while (next_permutation(aux.begin(), aux.end()));
}
```

O segundo algoritmo utiliza programção dinâmica, mas é um algoritmo ganancioso. Este algoritmo só funciona com grafos fortemente conexos.

```
//calculating results for every pair of points
for (size_t i=0; i<points.size(); i++)
{
    for (size_t j=0; j<=i; j++)
    {

        vector<PoI*> poi =calculateRouteBetweenTwoPoints(points[i], points[j]);
        if(poi.size()!=0)
        {
            routes[i][j] = poi;
            weigths[i][j]=getWeight(routes[i][j]);
        }

        vector<PoI*> p2 = calculateRouteBetweenTwoPoints(points[j], points[i]);
        if(p2.size()!=0)
        {
            routes[j][i]=p2;
            weigths[j][i]=getWeight(routes[j][i]);
        }

    }
}
```

```
//calculating best route using a greedy strategy
for (size_t i=0; i<points.size()-2; i++)
{
    double min_weigth=INF;
    size_t min_choice=0;
    for (size_t k=0; k<points.size(); k++)
        weigths[k][starting_point]=0;
    for (size_t j=1; j<points.size()-1; j++)
    {
        double weigth=weigths[starting_point][j];
        if (weigth!=0 && weigth<min_weigth)
        {
            min_weigth=weigth;
            min_choice=j;
        }
    }
    if (min_choice == 0)
        return {};
    auxr = routes[starting_point][min_choice];
    auxr.erase(auxr.begin());
    resp.insert(resp.end(),auxr.begin(), auxr.end());
    starting_point=min_choice;
}
auxr = routes[starting_point][points.size()-1];
auxr.erase(auxr.begin());
resp.insert(resp.end(),auxr.begin(), auxr.end());
```

Para a criação do grupo dos turistas foi utilizado o algoritmo já descrito na secção Principais Algoritmos (fase 2 e 3). Este é também um algoritmo ganancioso.

```
while (it!=tous.end())
{
    vector<PoI*> auxp;
    vector<Tourist*> auxt;
    auxt.push_back(&(*find(tourists.begin(), tourists.end(), *it)));
    auxp = *(it->getPoIs());
    auto itt = it+1;
    while(itt != tous.end())
    {
        vector<PoI*> difference;
        set_difference((*(itt->getPoIs())).begin(), (*(itt->getPoIs())).end(),
                (*(it->getPoIs())).begin(),(*(it->getPoIs())).end(),
                inserter(difference, difference.begin()));
        if (difference.size() <= tolerance)
        {
            tolerance -= difference.size();
            auxt.push_back(&(*find(tourists.begin(), tourists.end(), *itt)));
            auxp.insert(auxp.end(), difference.begin(), difference.end());
            tous.erase(itt);
            continue;
        }
        itt++;
    }
    routes.push_back(auxp);
    touristGroups.push_back(auxt);
    it++;
}
```

```
 oldTouristGroups = createTouristGroups(initialTolerance, oldRoutes);
 bool breakCycle=false;
 do
 {
     initialTolerance--;
     touristGroups=createTouristGroups(initialTolerance, routes);
     if (routes.size()>buses.size())
         break;
     for(size_t i=0; i<touristGroups.size();i++)
         if (touristGroups[i].size()>busesCapacity)
             breakCycle=true;
 }while (touristGroups.size()>buses.size() && (!breakCycle));
breakCycle=false;
oldTouristGroups= touristGroups;
oldRoutes=routes;
routes.clear();
 do
 {
     initialTolerance++;
     touristGroups=createTouristGroups(initialTolerance, routes);
     if (routes.size()>buses.size())
         break;
     for(size_t i=0; i<touristGroups.size();i++)
         if (touristGroups[i].size()>busesCapacity || touristGroups[i].size()==tourists.size())
             breakCycle=true;
 }while ((touristGroups.size()<oldTouristGroups.size()) && (!breakCycle));
```

Análise de complexidade dos algoritmos

Algoritmo de Dijkstra e A*

Estes dois algoritmos realizam cálculos similares a uma pesquisa com recurso a uma fila de prioridades. Tendo em conta que a complexidade da pesquisa é O(n) e a da inserção de vértices na fila de prioridades é O(log(n)), então a complexidade dos algoritmos de Dijkstra e A* é O(n * log(n)).

Algoritmo de Combinações (opção 9 e 12)

Este algoritmo pode ser dividido em duas partes.

A primeira parte corresponde à função getPaths, que calcula todas as ordenações possíveis dos pontos de interesse, e por isso tem complexidade O(n!).

O restante processamento da função corresponde a um ciclo na função principal, que invoca a função calculateRouteWithOrderedPoints, que por sua vez tem também um ciclo e invoca os algoritmos Dijkstra/A*. Esta parte da função tem assim complexidade $O(n^3 * log(n))$.

Desta forma, o algoritmo completo tem complexidade $O(n^3 * log(n))$.

Algoritmo Ganancioso/ Programação Dinâmica (opção 11 e 13)

Este algoritmo consiste num melhoramento do algoritmo anterior (opção 9), sendo ganancioso e recorrendo a programação dinâmica. Compreende dois ciclos na função principal e invoca os algoritmos Dijkstra/A*, tendo por isso complexidade $O(n^3 * log(n))$.

Algoritmo de Turistas (opção 10 e 14 – apenas criação dos grupos)

Para a organização dos turistas em vários grupos, foi criado um algoritmo procura agrupar turistas com pontos de interesse comuns. Consiste num ciclo na função principal, que invoca um outro ciclo e também a função createTouristGroups (que por sua vez tem 2 ciclos encadeados). Desta forma, a complexidade do algoritmo é $O(n^3)$.

Análise Empírica

Em baixo encontram-se os valores médios de tempo medidos aquando de testes das diferentes opções, com um mapa de testes e o mapa de Aveiro:

Opção	Testes (8 Pois)	Aveiro (7 Pois)
9	17ms	49ms
10	9ms	49ms
11	1ms	12ms
12	18ms	47ms
13	19ms	57ms
14	2ms	15ms

Como se pode constatar, a opção 11, que utiliza programação dinâmica, revela-se bastante mais eficiente a nível temporal em relação à opção 9. Em relação ao teórico melhoramento do algoritmo A* em relação ao Dijkstra, não foi possível comprovar com estes testes, mas consideramos que o problema reside, uma vez mais, no facto de o grafo não for suficientemente conexo.

Conclusão

Consideramos que este trabalho foi efetuado com sucesso pelos membros do grupo. Apesar de toda a fase de testes ter sido dificultada pelo facto de o grafo fornecido ser pouco conexo, característica essencial para garantir que os algoritmos funcionam corretamente, julgamos que o problema foi ultrapassado através da criação de um pequeno grafo mais conexo para testes, ainda que saibamos que esta não seria a solução ideal.

Concluímos também que o planeamento realizado na 1ª entrega se revelou assertivo, uma vez que praticamente a totalidade das funções e algoritmos foi implementada, para além de pequenas correções que foram sendo necessárias ao longo do desenvolvimento do projeto.

Relativamente ao método de trabalho adotado, o trabalho foi sendo feito aos poucos por todos os elementos do grupo. Cada um ia construindo em cima da informação já existente e corrigindo sempre que necessário. Todos os elementos do grupo revelaram um esforço de tamanho igual no desenvolvimento do projeto.

Bibliografia

- PowerPoints das aulas teóricas da unidade curricular, R. Rossetti, L. Ferreira, L. Teófilo, J. Filgueiras, F. Andrade
- Problema do caixeiro viajante, https://pt.wikipedia.org/wiki/Problema_do_caixeiroviajante