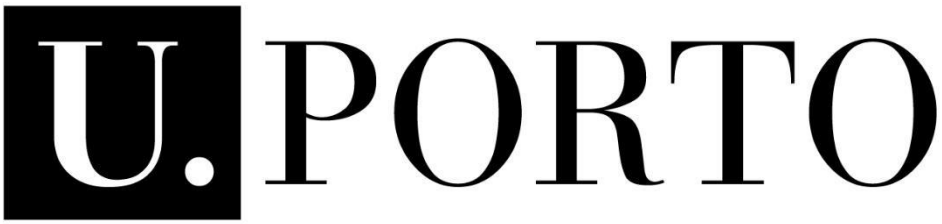


FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MEAT WAGONS
DISTRIBUIÇÃO DE PRISIONEIROS

Projeto CAL – MIEIC 2019/20

Turma 6 Grupo 6

Professor: Francisco Xavier Richardson Rebello de Andrade

Autores:

Bernardo Ramalho | 201704334

Francisco Gonçalves | 201704790

Martim Silva | 201705205

ÍNDICE

DESCRIÇÃO DO PROBLEMA

Os transportes de prisioneiros entre prisões, esquadras e tribunais são feitos usando carrinhas e camionetas adaptadas para o serviço. A empresa MeatWagons é uma empresa de segurança privada que realiza os transportes de prisioneiros entre quaisquer pontos de interesse definidos (prisões, esquadras e tribunais), disponibilizando um certo número de carrinhas para esse efeito.

Neste trabalho, pretende-se implementar um serviço, no qual sejam feitos um conjunto de pedidos de receção e entrega de prisioneiros a diversos pontos de interesse. Estes pedidos devem ser agrupados de forma a garantir otimização do percurso das carrinhas. O problema será decomposto em três iterações.

Tal como foi discutido com o professor, os prisioneiros estarão em locais distintos, mas assim que recolhidos irão todos para o mesmo local. Todas as carrinhas partem de uma central, fazem a receção de N prisioneiros e entrega dos diferentes prisioneiros no mesmo local voltando sempre à central no final da entrega.

O problema foi decomposto em **3 iterações principais**, de modo a testar diferentes problemas em cada uma das iterações.

1ª Iteração - Receber diversos pedidos utilizando 1 carrinha de capacidade = 1

Esta iteração servirá para testar os algoritmos do caminho mais curto entre 2 pontos. Nesta iteração iremos implementar todos os algoritmos (e suas melhorias) dos algoritmos do caminho mais curto. Como a carrinha é de capacidade 1, nesta fase ainda não haverá a preocupação de agrupar pedidos de diferentes prisioneiros. É importante referir que, cada pedido só será realizado, se for determinado que existe um caminho de ida e de volta, visto que a carrinha precisará de regressar à sua central. Além disso, poderá haver obras nas vias públicas e zonas inacessíveis, por isso, estas arestas também devem ser ignoradas no processamento. A ordem de entrega, nesta fase, será por ordem dos tempos de receção dos prisioneiros.

2ª Iteração - Receber diversos pedidos utilizando 1 carrinha de capacidade > 1

Nesta fase, devem ser agrupados os pedidos para garantir que a mínima distância possível seja percorrida e que o mínimo de carrinhas seja utilizado. O objetivo será então ter sempre que possível as carrinhas cheias, de modo a maximizar o potencial lucro da empresa. Assim, a carrinha deverá passar por uma série de diferentes pontos de recolha, entregar os prisioneiros no mesmo local e depois, regressar à central. Isto garante que seja alcançado o número mínimo de viagens possíveis. Nesta iteração a complexidade aumenta de forma exponencial, uma vez que terá de ser resolvido o

problema NP-hard do TSP. Neste problema teremos de considerar a melhor rota para ir buscar os diferentes pedidos em diferentes pontos.

3ª Iteração - Receber diversos pedidos utilizando várias carrinhas de diferentes capacidades

Na 3ª iteração, é usada uma frota de carrinhas de transporte de prisioneiros, com diversas capacidades. Isto implica diversas deslocações e diferentes combinações possíveis de agrupamentos de pedidos de entrega de prisioneiros, O sistema deve otimizar o agrupamento dos pedidos. Qualquer pedido, depois de concluído, deve voltar à central. Para o agrupamento dos pedidos deverão ser consideradas as zonas da localização de cada pedido, ou seja, conceber uma valor maxDist que se outro pedido estiver a uma distância maior que esse valor, não poderá ser considerado no agrupamento

FORMALIZAÇÃO DO PROBLEMA

Dados de entrada

- Gi = (V, E) → Grafo bidirecional(ou não dirigido) pesado constituído por:
  - V → Conjunto de vértices que representam pontos de recolha e entrega de prisioneiros. Cada vértice, por sua vez, é constituído por:
    - id → identificador do vértice
    - position → vector que contém as coordenadas geográficas do local;
    - tag → indica que tipo de local o vértice representa;
    - edges ⊆ E → subconjunto de E. Todos os caminhos que saem do vértice.
    - invEdges ⊆ E → subconjunto de E. Todos os caminhos que acabam no vértice.
    - outros dados, utilizados nos diferentes algoritmos (valor heurístico, índice da fila de prioridade mutável, path, path invertido, ...)
  - E → Conjunto de arestas, cada uma representando as estradas que ligam cada ponto de recolha e entrega. Cada aresta, por sua vez, é constituída por:
    - id → identificador da aresta
    - orig ∈ V → apontador para o vértice de origem;
    - dest ∈ V → apontador para o vértice de destino;
    - peso → peso da aresta
- R → Lista de pedidos que têm de ser entregues/recolhidos. Cada pedido R(i) tem a seguinte informação:
  - prisoner → Nome do prisioneiro;
  - des → identificador do vértice correspondente ao local de recolha;
  - priority → prioridade de recolha do prisioneiro;
  - arrival → Tempo para recolha do prisioneiro;
  - realArrival → Tempo real de recolha (dependendo do tempo de saída da carrinha);
  - realDeliver → Tempo real de entrega do prisioneiro;
- C → Lista de carrinhas que existem e podem ser usadas. Cada carrinha C(i) tem a seguinte informação:
  - ID → Identificador da carrinha;
  - capacity → Indica o número total de passageiros que a carrinha consegue levar;
  - nextAvailableTime → Tempo em que a carrinha está de volta à central;

- D → Representa a central da empresa MeatWagons. Esta central tem a seguinte informação:
  - position → vector que contém as coordenadas geográficas da central;
- M → Distância máxima entre dois pontos de interesse.
- I → Conjunto de vértices correspondentes aos pontos de interesse (tribunais, prisões, esquadras)

Dados de saída

- Cf → Lista de carrinhas que irão ser utilizadas para todos os trajetos. Cada carrinha Cf(i) tem a seguinte informação:
  - nextAvailableTime → Tempo em que a carrinha está de volta à central;
  - capacity → Indica o número total de passageiros que a carrinha consegue levar;
  - deliveries → Lista de entregas que a carrinha faz. Cada entrega tem a seguinte informação:
    - start → Tempo em que a carrinha começa a tratar da entrega (quando parte da central);
    - end → Tempo em que a carrinha acaba a entrega (quando chega à central);
    - requests $\subseteq R$→ Lista dos pedidos feitos na entrega.
    - forwardPath $\subseteq E$ → Caminho que a carrinha percorre até ao retorno da carrinha à central.
    - dropOff → Identificador do vértice que representa o ponto de entrega dos prisioneiros.
- Gf = (V, E) → O mesmo grafo bidirecional(ou não dirigido) pesado recebido no input

Restrições

Nos dados de entrada:

- $\forall$ V → Tag(V) == “CENTRAL” $\vee$ “INTEREST_POINT” $\vee$ “PICKUP” $\vee$ “DROPOFF” $\vee$ “DEFAULT”;
- $\forall$ E → dist(E) > 0 (isto é, não há distâncias negativas);
- $\forall$ C → Capacity(C) > 0 (Não existem capacidades negativas);
- M > 0

Nos dados de saída:

- Gf = Gi, isto é, o grafo deverá ser igual ao fornecido no input;
- Ci >= Cf, isto é, no máximo pode-se utilizar o número de carrinhas disponíveis;
- ∀ c ∈ Cf ∧ ∀ d ∈ deliveries(c):
    - Capacity(c) <= sizeOf(requests(d)), isto é, o número de requests de uma dada entrega não pode ser maior que a capacidade da carrinha;
    - start(d) < end(d), o tempo de saída tem de ser menor ao tempo de chegada.
- ∀ Cf → Capacity(Cf) > 0 (Não existem capacidades negativas);
- ∀ c ∈ Cf ∧ ∀ d ∈ deliveries(c): | requests(d) | <= | R | ;
- | Cf | <= | C | ;

Função objetivo

A nossa solução ótima é aquela que minimizar o número de lugares vazios por carrinha, minimizando assim o número de carrinhas usadas. Também se pretende minimizar a distância total percorrida. Isto é pretende-se minimizar a soma das funções f e g:

- f = | Ws |, isto é, minimizar os lugares vazios em cada uma das carinhas
- g = ∑c ∈ C (∑d ∈ deliveries(c) totalDist(d)), isto é, minimizar a distância total que as carrinhas viajam.

```
objectiveFunction():
    sum = 0
    for each wagon in wagons do
        for each delivery in deliveries(wagon) do
            sum += getTotalDist(delivery) + getSpaceLeft(wagon)

    return sum
```

ESTRUTURAS DE DADOS E CLASSES USADAS

Representação do Grafo

Grafo

A representação do grafo foi feita através da classe **Graph** declarada e definida em **src/Graph/Graph.h**. Nesta classe estão declaradas duas variáveis importantes: um **vetor de vértices** (std::vector<Vertex*> **vertexSet**) que nos permite operar sobre todo o conjunto de vértices de um grafo e um **hashmap** (std::unordered_map<int, Vertex*> **vertexIndexes**), cujo objetivo é associar o ID de um vértice ao próprio vértice, permitindo aceder aos vértices usando o ID em tempo constante (O(1)). Esta classe é composta também por mais métodos que permitem operações no grafo, assim como algoritmos de pesquisa.

Vértice

Os vértices são representados pela classe Vertex, definida em **src/Graph/Vertex.h**, que tem os campos **id** (int **id**), **posição** (Position **pos**), um **vetor de arestas** (**Edge**), que representa o conjunto de arestas que saem do vértice (std::vector<Edge> **adj**). Esta classe contém mais parâmetros entre eles a **tag** (enum Tag), que permite identificar o tipo de vértice (por exemplo se for o vértice central ou um ponto de interesse), o **path** (Vertex* **path**), que contém o último vértice do caminho mais curto até ao vértice a ser tratado e o "**caminho de arestas"** (Edge **edgePath**), que é necessário para assinalar **uma** aresta até ao vértice a ser tratado. Uma vez que implementamos o algoritmo de dijkstra bidirecional também precisamos de separar alguns conteúdos do vértice e por isso também temos os atributos invAdj, invDist, invPath, invPathEdge. Por último, ao estarmos a utilizar o algoritmo de Dijkstra single source multiple destinations para calcular os pontos mais curtos a partir da central, precisamos de os guardar, para não serem escritos por cima sempre que se faz um algoritmo do caminho mais curto entre 2 pontos. Para isso temos os atributos pathCentral e edgePathCentral.

Aresta

As arestas são representadas através da classe Edge definida em **src/Graph/Edge.h**. A estrutura guarda informação do **id** (int **id**) e peso da aresta (double **weight**). Como o grafo não é dirigido as arestas têm na sua estrutura informação sobre um vértice de origem (Vertex* **origin**) e destino (Vertex* **dest**).

Meat Wagons

Aplicação (Application.h)

Esta classe apresenta todos os menus do programa, gere todo o input do utilizador, e envia esse input, depois de processado, para o controlador.

Controlador (MeatWagons.h)

Definimos uma classe Meat Wagons, responsável por fazer de controlador geral do programa. Esta classe é responsável por gerir o grafo, de forma a separar da aplicação. Aqui são feitos todos os pedidos gerados na aplicação, desde ler, pré processar e calcular o caminho mais curto entre 2 pontos de 1 grafo já lido, até realizar as iterações propostas em cima e apresentá-las de uma maneira apelativa para o utilizador. Visto ser um programa dinâmico, poderá alterar se o número de carrinhas, as suas capacidades e realizar pedidos de prisioneiros de qualquer cidade de Portugal que contenha um mapa. Um objeto desta classe guarda o grafo, o visualizador do grafo, os pontos de interesse do grafo, o conjunto de carrinhas e de pedidos de entrega, a distância máxima considerada no agrupamento de pedidos e a velocidade média em metros por segundo.

Pedidos (Request.h)

Definimos uma classe Request, responsável por tratar da lógica dos pedidos. Esta classe trata de manipular a informação sobre nome de prisioneiro, ponto de destino, prioridade e tempo de partida e chegada. Relativamente aos tempos, decidimos aprofundar e ser minuciosos e, por isso, temos o tempo de partida desejado (escrito no pedido), e também o tempo real de partida do prisioneiro para o seu local de destino. Estes tempos podem diferir no caso de não haver carrinhas disponíveis, ou de outro pedido ter um tempo que não permita chegar a tempo do tempo desejado.

Carrinha (Wagon.h)

Definimos uma classe Wagon, responsável por tratar da lógica por trás das carrinhas, entidades que concretizam os pedidos. Esta classe tem informação sobre ID, capacidade, próxima hora em que está disponível, bem como um conjunto (vetor) do tipo Entrega (**Delivery**) com as entregas associadas à carrinha em questão. Esta classe é capaz de adicionar entregas associadas à carrinha e saber também a sua ocupação.

Entrega (Delivery.h)

Definimos uma classe Delivery, que permite facilitar as operações de entrega de prisioneiros. Esta classe contém informação sobre o tempo de partida e chegada, o conjunto (vetor) de pedidos, o caminho necessário a percorrer (vetor de arestas) e também o ID do vértice de entrega. Assim, uma entrega é composta por uma série de pedidos agrupados, o seu caminho percorrido, desde a central até à própria central, e o ponto associado ao destino dos prisioneiros coletados.

Outras classes

Leitor (Reader.h)

O nosso projeto tem uma classe Reader que se encarrega de fazer a leitura da informação necessária para projeto fazer sentido. Esta classe recebe um caminho para ler um grafo e todas as informações inerentes a este (nodes, edges, requests). É possível ler a informação dum grafo a partir do menu principal (com a opção 1). De facto, como cada mapa tem associado os seus pedidos, os pedidos são todos lidos depois do grafo ser lido.

Tempo (Time.h)

Definimos uma classe Time, responsável por tratar da lógica que envolve tempos, sendo um auxiliar relevante ao lidar com os tempos dos pedidos. Esta classe faz overload de uma série de operadores e facilitando o tratamento da lógica complexa de operações com tempos.

Posição (Position.h)

Definimos uma classe Position, responsável por guardar o x e um y associado a uma posição e fazer operações com esses atributos, tais como calcular a distância euclidiana com outro objeto Position.

Visualização dos mapas

Visualizador (GraphVisualizer.h)

Esta classe tem o propósito de gerir toda a parte externa da View (seguindo o padrão *MVC*) para este ser separado do controlador. é responsável por desenhar um grafo lido, um grafo pré processado, de desenhar a cores diferentes as arestas exploradas e processadas nos algoritmos do caminho mais curto e desenhar o percurso das carrinhas e da entrega dos prisioneiros. É também capaz de mudar a etiqueta associada a um vértice, a sua cor e tamanho. Todo o desenho é feito utilizando threads, mas este tópico é descrito, a promenor, no tópico dos **Casos de utilização**.

PERSPETIVA DE SOLUÇÃO

Pré-processamento

Pré Processamentos:

O grafo deve ser sujeito a um pré processamento ao nível do grafo, dos dados e dos percursos de duração mínima. Isto aumenta a eficiência temporal ao aplicar os algoritmos descritos abaixo e no decorrer do programa. É importante salientar que, apesar de fazer mais sentido considerar o grafo dirigido, face aos dados obtidos de cada um dos mapas, **foi necessário considerar o grafo não dirigido.**

Pré-processamento do grafo:

Devem ser eliminadas as arestas que representam estradas onde estejam a decorrer obras nas vias públicas (zonas inacessíveis). Além disso, é necessário avaliar a conectividade do grafo, a fim de identificar pontos de recolha e de entrega com pouca acessibilidade. Assim, todos os vértices, incluídos pontos de interesse, que não pertencem ao mesmo componente fortemente conexo da central, devem ser removidos. Este processamento que qualquer carrinha que saia da central consegue alcançar qualquer vértice do grafo processado.

Pré-processamento dos dados:

Devem ser retirados todos os pedidos em que os vértices de receção dos prisioneiros não estejam contidos no grafo pré processado. Os pedidos devem, também, ser ordenados pelo tempo definido para recolha dos prisioneiros, através da escolha de uma estrutura de dados pertinente.

Pré-processamento dos percursos de duração mínima:

Será necessário um algoritmo que permita saber a distância entre quaisquer pares de pontos de interesse, assim como o caminho entre eles. Tanto o algoritmo de Dijkstra como o de Floyd–Warshall podem ser utilizados nesta etapa. Os pontos de interesse considerados são os tribunais, esquadras e prisões. Foi estudado a opção de usar ambos os algoritmos. O algoritmo de Dijkstra e as melhorias que implementados foram cruciais para escolhermos o Dijkstra. Sendo assim, para qualquer uma das iterações, inicialmente é feito o algoritmo de Dijkstra original a partir da central designada para cada cidade e para todos os outros pontos do grafo. Deste modo, não volta a ser necessário fazer o algoritmo de caminho mais curto a partir da central, após a leitura do grafo.

Algoritmos

Dijkstra

Este algoritmo foi usado no cálculo do caminho mais curto entre todos os pares de vértices e entre dois vértices (single destination e all pairs destination). É um algoritmo ganancioso usado nos casos de grafos dirigidos com pesos, para encontrar o caminho mais curto.

Cada vértice terá de guardar a duração mínima até ao nó de origem assim como o caminho, isto é, o vértice anterior pertencente ao caminho mais curto.

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

Vamos tentar aplicar várias melhorias a este algoritmo, descritas abaixo.

Este algoritmo foi escolhido devido à facilidade da sua implementação, estudado nas aulas práticas e teóricas, e com algumas otimizações pode ser muito eficiente e prático.

Em termos de eficiência, a fase de preparação tem **O(|V|)** e a fase da pesquisa **O((|V|+|E|)*log(|V|)).**

Na parte de preparação, é atribuído um valor da distância a cada vértice de infinito e o caminho mínimo a ser percorrido (πf) como null. Coloca-se também que a distância do vértice inicial a 0 (vértice de começo).

Para além disto, coloca-se o vértice de origem na fila de mínima prioridade. Na parte de pesquisa, é feito um ciclo que acaba até a fila de prioridade estar vazia. Primeiro, extrai-se o vértice com menor prioridade (vértice com menor distância). Em seguida, para cada vértice adjacente ao vértice extraído, verifica-se a distância atual até ele (inicialmente como infinito) é maior do que a distância usando o caminho do vértice extraído. Se o for, atualiza esse vértice com os valores do caminho menor, do novo valor da distância e a posição na fila de prioridade. A complexidade espacial é de **O(|V|) .**

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)

RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u

INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Single source, single destination:

A primeira otimização a ser feita, será no cálculo do caminho mais curto entre 2 pontos. Como o algoritmo tradicional de Dijkstra calcula o caminho mais curto a partir de um vértice inicial até todos os outros vértices, a melhoria a ser feita neste caso será de parar o ciclo quando o vértice extraído, da fila de prioridade, for igual ao vértice de destino pretendido.

Fila de prioridade dinâmica:

Nesta melhoria, o próximo vértice a processar continua a ser o da distância mínima (greedy), no entanto, não é necessariamente o mais antigo. Isto obriga a usar a fila de prioridade em vez de uma fila simples. Além disso, será necessário alterar as prioridades estabelecidas e por isso terá também de ser dinâmica. A cada passo do método, pode ser necessário realizar operações de extração, inserção ou decrease-key na fila de prioridade.

Pesquisa orientada:

Juntando ao que foi acima descrito, nesta etapa acrescenta-se uma função de heurística (função de avaliação), normalmente a distância euclidiana entre o vértice atual e o vértice de destino. Para aplicar esta melhoria, é preciso estar consciente do que é a desigualdade triangular. Esta função de avaliação evita que o algoritmo seja completamente ganancioso e através da heurística definida consegue "indicar" para que direção o algoritmo de Dijkstra deve prosseguir. O ganho conseguido ao usar este algoritmo é enorme e muito benéfico, principalmente nas zonas mais densas. Apesar disto, não consegue garantir exatamente que o caminho encontrado seja sempre o mais curto (depende da heurística escolhida). A utilização, ou não, desta otimização será discutida na fase de implementação, com base a testes práticos. Assim, será mais rápido que o algoritmo de Djikstra normal, uma vez que tem de processar menos vértices. Tem uma complexidade de **O((|E| + |V|) * log(|V|)).** A complexidade espacial é de **O(|V|) .**

Pesquisa bidirecional:

A pesquisa bidirecional é considerado uma otimização (na maioria dos casos) dos algoritmos apresentados em cima (Dijkstra's e A*). Em vez de se fazer a pesquisa numa das direções, fazemos em ambas. Isto é, faz se uma pesquisa a começar do vértice inicial e outra, em simultâneo, a começar do vértice final.

A pesquisa acaba quando ambas as pesquisas tratam do mesmo vértice. Esta abordagem tem a vantagem de ser mais rápida visto que só faz metade do caminho. No entanto, esta abordagem tem um problema que algumas implementações não tratam. Este problema refere se ao caso em que o vértice em que ambas as pesquisas se encontram(interseção) não pertencer ao caminho mais curto. Apesar de parecer lógico que o vértice em que eles se encontram é o vértice mais próximo entre os dois, não há nenhuma forma de garantir empiricamente isso para todos os casos. Para combater isto, no final da pesquisa é preciso percorrer todos os vértices que se encontram na fila de prioridade para ver se há algum que tenha um caminho mais curto.

Apesar deste último passo a complexidade do algoritmo continua a ser mais baixa que a complexidade dos algoritmos normais. No dijkstra normal, assumindo um branching factor x e uma distância real desde o início até ao fim de y, temos que a complexidade é O(x^{y}). Como o algoritmo corre só até meio a complexidade fica 2 * O($x^{y/2}$) que é menor que a mostrada acima.

Algoritmo de Floyd-Warshall:

Este algoritmo será usado no cálculo do caminho mais curto entre todos os pares de vértices. Utiliza conceitos de programação dinâmica. Baseia-se em matrizes de adjacências, uma com as distâncias mínimas e a outra matriz de predecessor no caminho mais curto (path).

Tem uma complexidade melhor (O(|V|^3)) ao algoritmo de Dijkstra, tanto se o grafo for denso, como se o grafo for pouco denso. Como este algoritmo exige um trabalho muito grande de pré-processamento será avaliado mais tarde se o iremos utilizar.

Estratégias de aplicação dos algoritmos

Cada uma das estratégias abaixo definidas corresponde a cada uma das iterações já previamente descritas. Relativamente aos pré-processamentos, todas devem fazer o mesmo para otimizar o programa. Uma vez que todas as iterações (exceto a primeira), muito similares no seu contexto, tratam do problema Vehicle Routing Problem (VRP), uma generalização do problema Travelling Salesman Problem (TSP), partilham o mesmo objetivo de minimizar a distância total percorrida.

Em todos os casos, o pré-processamento do grafo, resulta num grafo apenas com todos os vértices correspondentes a pontos de interesse, e vértices que correspondem aos caminhos mais curtos entre todos os pares de vértices de pontos de interesse.

Receber diversos pedidos utilizando 1 carrinha de capacidade 1

Na primeira iteração, com uma única carrinha, o problema reduz-se a encontrar o caminho mais curto entre a central, local de receber o prisioneiro e o local de destino final do passageiro. Por último, regressar a central para ir buscar novo prisioneiro, pela ordem estabelecida de cada pedido.

Receber diversos pedidos utilizando 1 carrinha de capacidade maior que 1

Na segunda iteração, os pedidos passam a poder conter vários pontos para receber e apenas um para entregar prisioneiros, isto é, recolha de mais que um prisioneiro de diferentes locais e entrega no mesmo estabelecimento, usando o mesmo veículo. Nesta iteração estaremos a lidar com o problema do Caixeiro Viajante (Travelling Salesman Problem), uma vez que teremos de passar por todos os locais de interesse definidos com o menor custo total em termos de distância percorrida.

Nesta iteração a complexidade aumenta a um grau muito grande, por se tratar de um problema complexo (TSP). Temos soluções gananciosas como é o caso do *nearest neighbour*. Esta solução apesar de eficiente, não é exata. Baseia-se em escolher um vértice inicial aleatório e depois escolher o próximo vértice que for o mais perto do que escolheu inicialmente. Isto pode levar a problemas de sobreposição de caminhos o que faz não ser uma solução ótima (referência 2). Como alternativa sería utilizado o método de Held-Karp, *brute force* ou *spanning trees*. Estes algoritmos teriam um custo muito elevado e serão explorados apenas no final. A solução inicial escolhida será então do *nearest neighbour*.

Receber diversos pedidos utilizando várias carrinhas de diferentes capacidades

Na terceira iteração há a possibilidade de ter um conjunto de veículos, de diferentes capacidades, e combiná-los com um conjunto de diversos pedidos. O algoritmo deve ser capaz de alocar o número ideal de prisioneiros com base nas capacidades dos veículos, o número de carrinhas para cada pedido (pode ser necessário subdividir pedidos se a capacidade máxima do maior veículo não for necessária) tendo em conta uma função de avaliação. Nesta fase o problema torna-se idêntico ao problema do Vehicle Routing Problem (VRP), uma generalização do problema Travelling Salesman Problem (TSP).

Tendo carrinhas de diferentes capacidades, é necessário começar pelas carrinhas de maior capacidade para evitar desperdícios. Isto também mínima o número de veículos utilizados, reduzindo o custo da empresa. Para a escolha da carrinha a ser usada em seguinte foi necessário desenvolver um algoritmo ganancioso:

```
chooseWagon(n, W) // n = number of prisioners, W = available wagons
    sort(W) // sort by capacity (first the largest)
    w ← W.begin()
    while n > 0 & W.size > 0 do
        // until there is no space left in a wagon
        if n > w.capacity then
            n - w.capacity
            w -> U
            W.erase(w)
            w ← W.begin()
        // if there is a space left -> start by the smallest capacity wagon
        else
            w ← next(W)
            if w == W.end() || n > w.capacity then
                w ← retrieve(W)
                n - w.capacity
                w -> U
                W.erase(w)
```

De seguida, será aplicado o algoritmo para agrupar os melhores pedidos para cada capacidade de cada veículo e será aplicado o algoritmo do TSP para entregar todos os prisioneiros recolhidos no mesmo local.

Estratégias de agrupamentos de pedidos

A questão de agrupar os pedidos conforme as localizações é fulcral para a questão de otimizar e obter a máxima eficiência para os percursos. Estes agrupamentos devem ter como base o maior lucro da empresa, ou seja, maximizar a capacidade de cada carrinha para diminuir o número de viagens e diminuir a distância total percorrida.

Neste caso, tratando-se de prisioneiros, os tempos de espera não serão o mais importante para a função objetivo, mas sim maximizar as capacidades das carrinhas em movimento.

As carrinhas partem da central completamente vazias. Sendo o objetivo preencher totalmente a capacidade de cada carrinha, serão procurados |N| (capacidade da carrinha) pedidos de receção de prisioneiros para preencher a carrinha com base nas distâncias entre cada pedido.

Começamos por adicionar o primeiro pedido que existe na lista que está ordenado por tempo de espera (primeiro os que têm o menor tempo de espera). Depois corremos a lista de todos os pedidos à procura de pedidos que tenham a menor distância ao ponto inicial.

Se houver menos que N pedidos, eles são todos adicionados à mesma carrinha. Se não, adicionamos os primeiros pedidos que estejam a menos da distância máxima aceitável à carrinha.

Quando a carrinha estiver cheia, continuamos a comparar os pedidos que ainda não estão atribuídos com aquele que têm a maior distância ao ponto inicial. Se encontrarmos algum que esteja a menos distância então substituímos aquele que têm maior distância com este. Fazemos isto até iterarmos sobre todos os pedidos não atribuídos.

Desta forma, conseguimos ter a certeza que a carrinha não se afasta muito do ponto inicial e diminuímos assim o percurso que esta faz. O algoritmo que implementámos percorre todos os pedidos (O(|R|)) e, no pior caso, em cada requests terá de percorrer o vetor onde guardamos N (capacidade da carrinha) pedidos. O que faz com que a complexidade seja **O(|R| * |N|).**

Análise da conectividade

Como foi mencionado na secção de perspetiva de solução e pré processamento dos grafos, deve ser feita uma remoção de arestas indisponíveis e vértices inalcançáveis, bem como uma análise de conectividade dos grafos. Nesta análise pretendemos distinguir o componente fortemente conexo

(**CFN**) ao qual pertence a **central**, removendo vértices que não pertençam a este componente. Abaixo mencionamos 3 possibilidades de análise de conectividade.

Algoritmo Kosaraju

Para distinguir todos os **CFN** do grafo vamos utilizar o algoritmo de Kosaraju, que se traduz nos seguintes passos:

1. Fazer uma pesquisa em profundidade no grafo, numerando os vértices encontrados em pós ordem
2. Transpor o grafo (inverter o sentido de todas as arestas)
3. Realizar uma nova pesquisa em profundidade, dando prioridade aos vértices com índice maior, que estão ainda por visitar.

Após aplicar este algoritmo, todas as árvores resultantes representam **CFN**s. No entanto, para o nosso caso específico apenas é relevante o componente que contém a central, pelo que quando o **CFN** que contém este ponto for encontrado, podemos parar a execução do algoritmo, fazendo uma otimização relevante. Este método reduz-se a duas pesquisas em profundidade e também à transposição (inversão das arestas) do grafo, pelo que todo este algoritmo tem uma complexidade linear, já que cada uma destas operações tem uma complexidade O(V+E).

Algoritmo de Tarjan

Uma outra opção para determinar os **CFN**s de um grafo é o Algoritmo de Tarjan, que apesar de também ter uma complexidade linear, é mais eficiente, já que usa apenas uma pesquisa em profundidade.

Depth first search (DFS)

Perante a possibilidade de serem fornecidos mapas com grafos não-dirigidos vamos optar por implementar **DFS**, fazendo a pesquisa a partir da **central** e marcando os vértices como visitados ao longo do percurso. Resumidamente, o algoritmo:

1. Marca todos os vértices do grafo como não visitados;
2. Marca o vértice presente como visitado;
3. Caso o vértice de destino ainda não tenha sido visitado, visitá-lo recursivamente, fazendo uso do passo 2 e 3 na recursividade, para todas as arestas com origem no vértice presente.

O facto deste algoritmo ser recursivo garante que os vértices serão percorridos em profundidade ao invés de em largura, explorando cada ramo o mais fundo possível antes de realizar o backtracking. Como o vértice inicial será a central, apenas encontraremos vértices pertencentes ao mesmo **CFN**. A complexidade média deste algoritmo deve ser **O(|V| + |E|)**, já que para marcar todos os vértices como não visitados o algoritmo tem de percorrer todos os vértices - |V| - e, já na fase de pesquisa, percorre todas as arestas de cada vértice **encontrado** - |E|.

CONECTIVIDADE DOS GRAFOS UTILIZADOS

Relativamente aos grafos utilizados, como já referimos, utilizamos os grafos bidirecionais devido ao mau nível de utilização dos mapas dados inicialmente. Apostamos na diversidade e escalabilidade, e por isso, o nosso programa funciona com qualquer 1 dos mapas dados inicialmente.

Os mapas dados inicialmente estavam em mau estado, uma vez que depois de pré processar o grafo, o número de vértices que pertencem ao mesmo componente fortemente conexo a que a central pertencia eram muito reduzidos. Assim optámos por considerar o grafo direcional, para podermos executar as iterações com sucesso.

Recebemos, no entanto, o mapa do Porto corrigido, no qual é possível analisar a correta implementação do algoritmo do DFS. Este mapa depois não é utilizada nas entregas, uma vez que não possui as tags e o programa já funcionava completamente com o mapa do Porto antigo, antes deste mapa do Porto corrigido ter sido dado. Abaixo é possível analisar o nosso pré processamento do novo mapa do Porto e comparar com o que era esperado

Em cima, podemos ver, à esquerda, o que é esperado obter do novo mapa do Porto depois de pré processado, e à direita, o mapa novo do Porto antes do pré processamento. Em baixo, podemos ver os nossos resultados com o mesmo mapa.

IMPLEMENTAÇÃO E ANÁLISE

Após planear a implementação passámos à parte mais prática do projeto, seguindo as ideias que tínhamos delineado até aqui. Começámos por adaptar as estruturas de dados (mais detalhes em **Estruturas de dados e classes utilizadas** – página 8) fornecidas nas aulas práticas e criar uma interface robusta que controla o funcionamento do programa. Implementámos o algoritmo clássico de Dijkstra, o Dijkstra orientado (**A***), bem como o Dijkstra bidirecional e adaptámos as soluções às necessidades do problema proposto, deixando de parte alguns algoritmos que mencionámos na parte de planeamento.

Serão agora abordados todos os algoritmos desenvolvidos e implementados por nós. Sobre os algoritmos do caminho mais curto não será abordado a sua complexidade, uma vez que já foi discutida em tópicos anteriores, no entanto será analisado a diferença entre os algoritmos abordados.

Função Objetivo

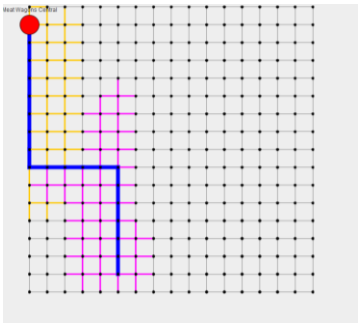
Este algoritmo exige que sejam percorridos todos os veículos e, para cada veículo, todas as entregas feitas pelo mesmo. Assim, como cada entrega está associada a apenas uma carrinha, a complexidade deste algoritmo é **O(|E|)**, onde |E| é o número de entregas total .

Pseudocódigo:

```
objectiveFunction():
    sum = 0
    for each wagon in wagons do
        for each delivery in deliveries(wagon) do
            sum += getTotalDist(delivery) - getSpaceLeft(wagon)

    return sum
```

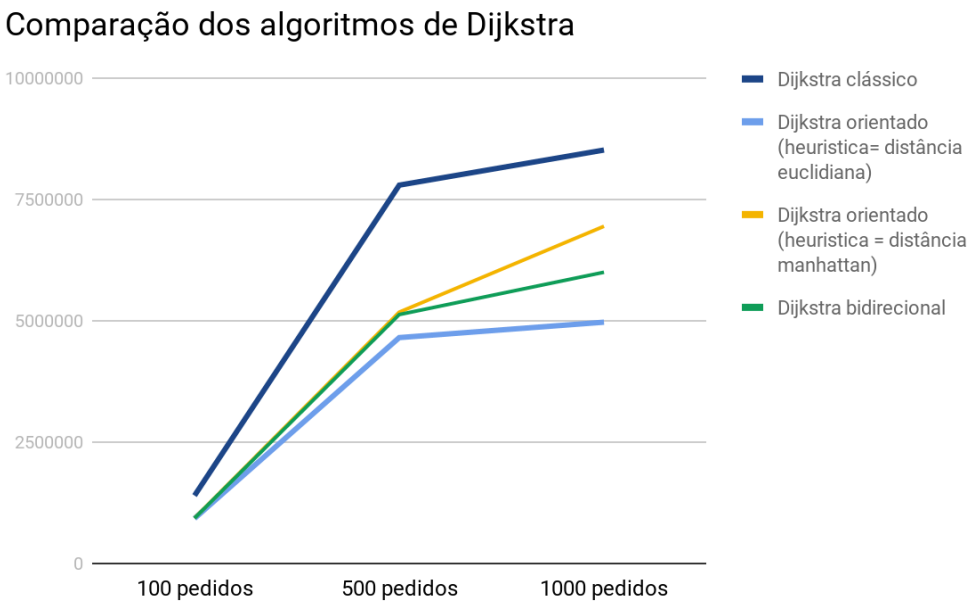
Comparação dos algoritmos de caminho mais curto entre 2 vértices

Fase de implementação e teste



Para testar efetivamente todos os algoritmos do caminho mais curto começamos por analisar os algoritmos nos grafos "*Grid*" , uma vez que é bastante mais fácil dar debug. Funcionando a próxima fase era testar nos mapas dados e analisar os diferentes algoritmos.

Comparação dos algoritmos de Dijkstra: Função Objetivo



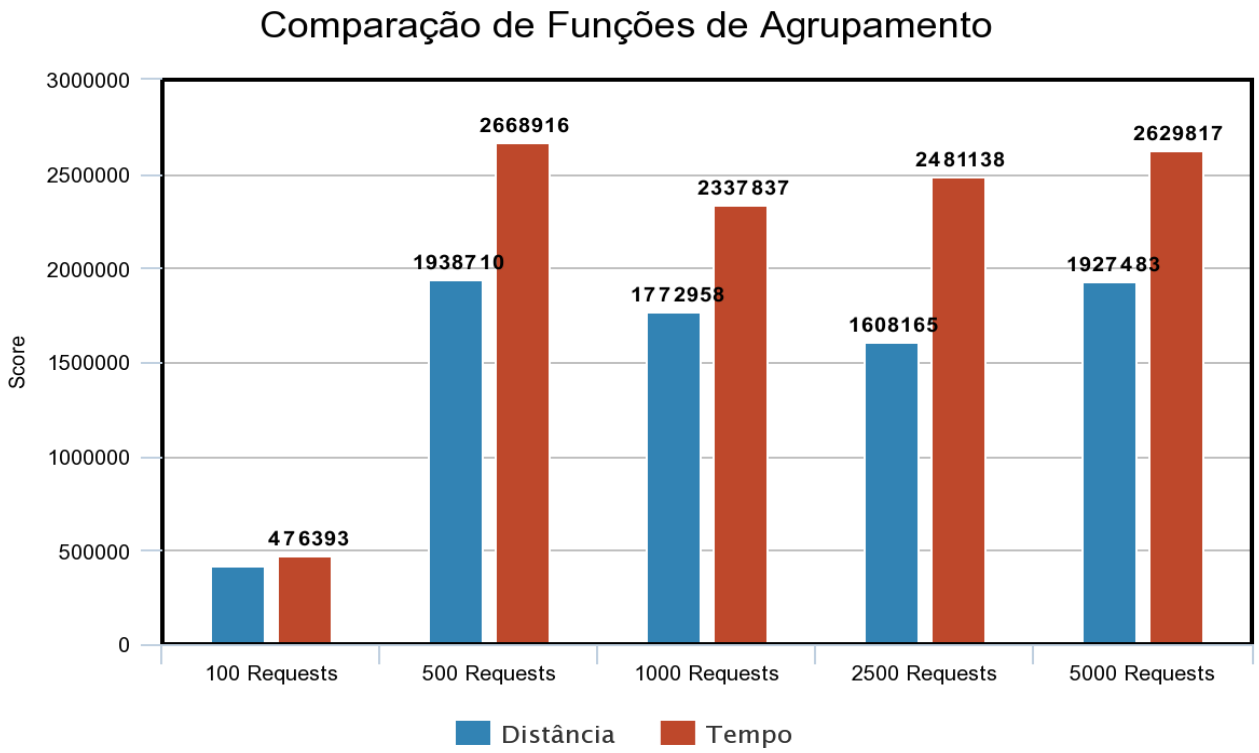
Comparando utilizando a iteração 1 e a função objetivo claramente é visível que o A* com a heurística da distância euclidiana é bastante melhor que o dijkstra original e ligeiramente melhor que o A* com a heurística da distância de manhattan. Relativamente à complexidade, já discutido na fase de planeamento, concluímos que o A* é bastante mais rápido que o dijkstra original. Também foi possível testar, em termos práticos, que o tempo de execução do A* com a heurística da distância euclidiana é inferior que qualquer outro algoritmo discutido aqui. Por isso a decisão será entre o A* e o dijkstra Bidirectional. Olhando para os dados o A* tem claramente vantagem, na análise da função objetivo, e no tempo de execução médio. Em média odijkstra Bidirectional chega a demorar 2 vezes mais que o A*, devido às confirmações que tem de fazer, descritas no tópico dos Algoritmos. Por isso, decidimos escolher o A* para executar o algoritmo do caminho mais curto entre 2 pontos. Foi tentado executar o dijkstra bidirecional com threads, mas não se obteve sucesso.

Comparação de diferentes estratégias de agrupamentos

Para este trabalho desenvolvemos duas formas de agrupamento de pedidos. Uma que agrupava os pedidos tendo em conta a distância entre si e outra que agrupava os pedidos que tinham o tempo de espera mais curto.

Para comparar usamos a nossa função objetivo, explicada no capítulo da **Formalização do Problema**, que nos dá um valor empírico para podermos comparar de forma justa ambos os algoritmos. O objetivo é sempre minimizar a função objetivo.

Em termos de testes, fizemos os testes sempre com 4 carrinhas (capacidades 1, 5, 10 e 12) a correr a iteração 3. Fizemos testes com 100, 500, 1000, 2500 e 5000 pedidos. Os resultados podem ser visto no gráfico seguinte:



As barras a azul são relacionadas com o algoritmo que agrupa pela distância e as vermelhas pelo tempo de espera. Ao analisar este gráfico conseguimos ver que o algoritmo que agrupa pela distância tem uma performance muito melhor (relativa à nossa função objetivo) que o outro. Por estas razões é que escolhemos utilizar o algoritmo das distância no nosso trabalho.

De notar que não se deve comparar a performance com quantidades de requests diferentes. Isto é, ao olhar para o gráfico pode se estranhar que os 1000 Requests tenham um score mais baixo que os 500, no entanto, os requests são gerados aleatoriamente por um script de python. Isto faz com que o grupo de 500 Requests gerados simplesmente estejam mais longe uns dos outros do que os do grupo de 1000 requests.

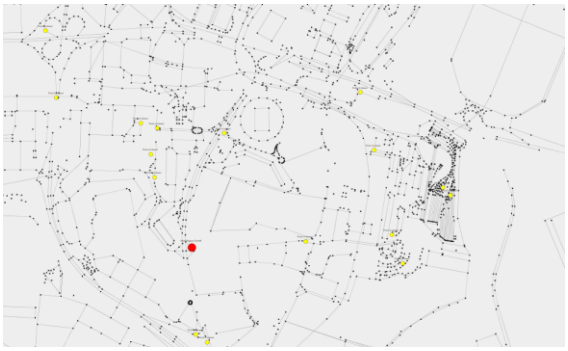
Conectividade e pré-processamento do grafo

```
preProcess(node):
    for each vertex in vertexSet(graph) do
        visited(vertex) = false

    dfsVisit(node)

    for each vertex in vertexSet(graph) do
        if(!visited(vertex)) do
            removed.insert(vertex)
            erase vertex

    for each vertex in vertexSet(graph) do
        for each edge in adj(graph) do
            if(removed.find(dest(dest))) do
                erase edge
```

```
dfsVisit(graph, node):
    visited(node) = true
    for each edge in adj(v) do
        if !visited(dest(graph,edge)) then
            dfsVisit(dest(graph, edge))

    for each edge in invAdj(v) do
        if !visited(dest(graph,edge)) then
            dfsVisit(dest(graph, edge))
```

Como foi dito acima que iremos considerar, para a análise da conectividade, o algoritmo do DFS, dado que os mapas fornecidos são representados por grafos não-dirigidos.

Os vértices não visitados pelo DFS são removidos do conjunto de vértices do grafo. A função que trata desta análise chama-se preProcess (definida na classe **Graph**) e recebe o índice de um vértice origem a partir do qual a pesquisa é iniciada. Ao fazer este processamento através do **menu** para o grafo de Lisboa obtivemos os seguintes resultados:



Grafo sem pré processamento: 74622 vértices

Grafo pré processado: 25658 vértices

Como podemos observar este método consegue remover uma quantidade muito significativa(depende do mapa) de vértices que não pertencem ao componente fortemente conexo associado à central definida, tendo neste caso, para o grafo de Lisboa a começar num nó de conectividade alta ao qual chamámos de **central**, removido cerca de ⅔ dos vértices e concluindo que o grafo não é fortemente conexo.

Em termos de complexidade, a função dfsVisit tem complexidade O(|E|), pois percorre, no máximo, todas as arestas do grafo. A função preProcess percorre todos os vértices do grafo (|V|) para os marcar com uma flag de não visitados. De seguida, é chamada a função dfsVisit, que como já foi descrita, percorre todos as arestas e marca os nós que pertencem ao mesmo componente fortemente conexo a que a central pertence, tendo complexidade O(|E|) . Até este ponto, complexidade do algoritmo dfsVisit é então **O(|V| + |E|)**.

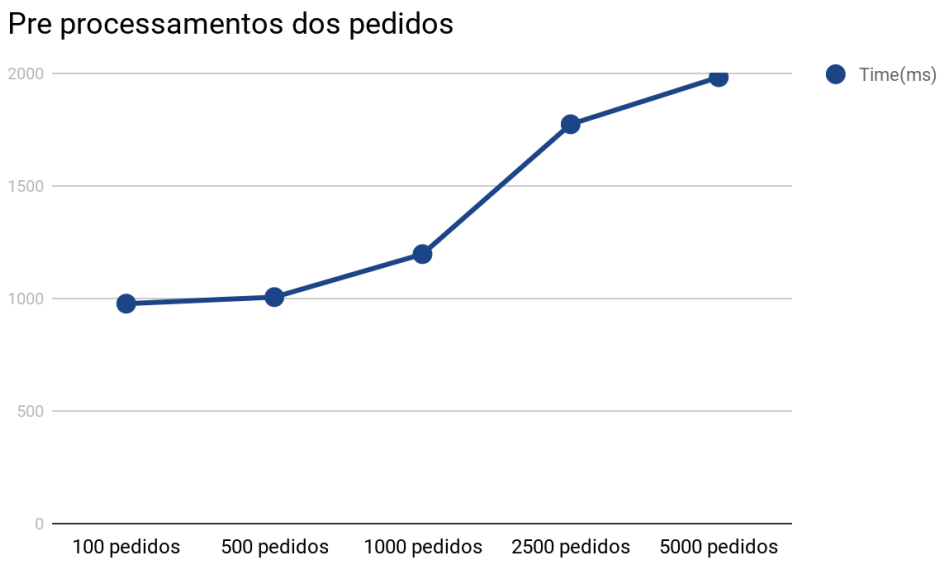
Por último, o método preProcess, remove todos os vértices que não foram visitados pelo dfsVisit, tendo por isso, uma complexidade de **O(|V|^2)**, visto que percorre toda a lista de vértices (|V|) e a operação de eliminação também tem complexidade O(|V|). Coloca também os vértices removidos numa estrutura de dados (hashmap) para ser usada ao analisar as arestas, com complexidade constante. Isto serve para eliminar as arestas que tenham como destino vértices eliminados. Esta parte tem uma complexidade **O(|E|)**, já que a procura num hashmap é constante.

Assim, no global, a complexidade da função preProcess é de **O(|V| + |E| + |V|^2 + |E|)**

Pré-processamento das reserva

```
proProcessGlobal(node):
    preProcess(node)
    for each request in requests(graph) do
        if(findVertex(dest(request)) == null) do
            erase(request)
```

Depois de pré processado o grafo, discutido em cima, serão apagados todos os pedidos em que o local de recolha do prisioneira já não esteja contido no grafo pré processado. A complexidade global deste método, tendo em conta a complexidade descrita no algoritmo usado, de cima, é de **O(O(**preProcess**) + |R|).**



Solução Travelling Salesman Problem (tspPath)

Para calcular o caminho mais curto entre a central e o ponto de entrega, passando por todos os pontos de recolha, seguindo a estratégia do *nearest neighbour*, nós fazemos os seguintes passos:

1) Começamos escolher o ponto de recolha com o menor tempo de espera (como usamos um vetor ordenado, este será o ponto no inicio do vetor);
2) Procuramos o pedido relativo a esse ponto (O(|R|), em que R é o **número de requests agrupados**) e atribuímos o tempo a que a carrinha chega a esse sítio;
3) Iteramos sobre os restantes pontos de recolha (O(|R-1|)) e em cada iteração fazemos:
    a) Procuramos o ponto mais perto do ponto de recolha anterior (O(|R|), existem tanto pontos como requests);
    b) Calculamos a distância do ponto mais perto até ao ponto de recolha anterior, utilizando o *djikstraBidirectional* ;
    c) Calculamos esse caminho usando *getPathTo*(O(|Vn1|) em que Vn1 representa o número de vértices pertencentes ao caminho mais curto)
    d) Procuramos o pedido relativo a esse ponto (O(|R|), em que R é o número de requests agrupados) e atribuímos o tempo a que a carrinha chega a esse sítio;
4) Calculamos a distância e o caminho do último ponto ao ponto de recolha utilizando as funções *djikstraBidirectional* e *getPathTo*(O(|Vn2|) em que Vn2 representa o número de vértices pertencentes ao caminho mais curto entre o último ponto e o ponto de recolha));
5) Como já temos a distância total do percurso, guardamos o tempo de entrega em todos os pedidos (O(|R|)).

A complexidade global desta interação é O(|R| + |R - 1|*(|R| + O(*dijkstraBidirectional*) + O(|Vn1|)) + O(*dijkstraBidirectional*)) + |Vn2| + |R|)

Nota: A complexidade do *dijkstraBidirectional* é referida em cima, no tópico **Algoritmos.**

```
tspPath(tspNodes, requests, path, dropOff, start):
    totalDist = 0

    closest = nearestToCentral(tspNodes)
    totalDist += getPathFromCentral(closest, path)
    erase(closest in tspNodes)

    Request r = findRequest(closest, requests)
    if(r is not null) do
        startTime = startTime - Time(totalDist / velocity)
        realArrival(r) = startTime + Time(totalDist / velocity)

    while(tspNodes is not empty) do
        next = nearestNeighbour(closest, tspNodes)
        djikstraBidirectional(closest, next)
        totalDist += getPathAfterDjikstra(central, path)

        Request r = findRequest(next, requests)
        if(r is not null) do
            realArrival(r) = startTime + Time(totalDist / velocity)

        erase(next in tspNodes)
        closest = next

    djikstraBidirectional(closest, dropOff)
    totalDist += getPathAfterDjikstra(dropOff, path)

    for each request in requests do
        realDeliver(r) = startTime + Time(totalDist / velocity)
        erase(request)

    return totalDist
```

1ª iteração - 1 carrinha de capacidade 1 (|W| = 1)

Numa primeira fase e como descrito atrás nesta primeira iteração, só utilizamos 1 carrinha (wagon) de capacidade 1, entregando cada pedido (request) no seu local de destino, voltando sempre à central de onde parte a carrinha.

A implementação desta iteração passa por inicializar a carrinha , seguido dum ciclo em que processamos os pedidos, um de cada vez(O(|R|)). A solução em código está apresentada de forma genérica, no entanto, trata-se de remover a única carrinha disponível da estrutura **wagons**, o que significa no contexto do nosso programa que ela vai satisfazer um pedido, saindo da central.

Ao sair da central é necessário calcular:

1. O caminho da central até ao destino associado ao request(localização do prisioneiro). Isto é feito recorrendo à função pertencente à classe *getPathFromCentralTo*(O(|Vn1|) em que Vn1 representa o número de vértices pertencentes ao caminho mais curto)
2. Ponto de entrega do prisioneiro através da função *chooseDropOff*(), que retorna um ID de um ponto de destino (id de um vértice aleatório que não seja o vértice de destino do pedido) (O(|1|))
3. A hora exata a que a carrinha sai da central, para isso, comparamos o tempo de espera do pedido que a carrinha tem com a próxima hora que a carrinha vais estar disponível. Se a carrinha tiver disponível antes do tempo de espera, então a carrinha espera para sair da central de forma a chegar às horas exatas ao ponto de recolha. Se não, a carrinha sai imediatamente depois de chegar à central.
4. O caminho do local de partida do prisioneiro até ao ponto de destino escolhido (dropOffNode). Novamente, é feito recorrendo à função pertencente à classe *getPathTo*(O(|Vn2|)). É também chamada a função do *djikstraBidirectional*
5. Por último, são usadas novamente as funções *getPathTo*(O(|Vn3|)) e *djikstraBidirectional* para calcular o caminho de regresso à central. Desta forma passamos a ter conhecimento da distância total do caminho a percorrer e podemos associar um pedido a uma entrega (Delivery).

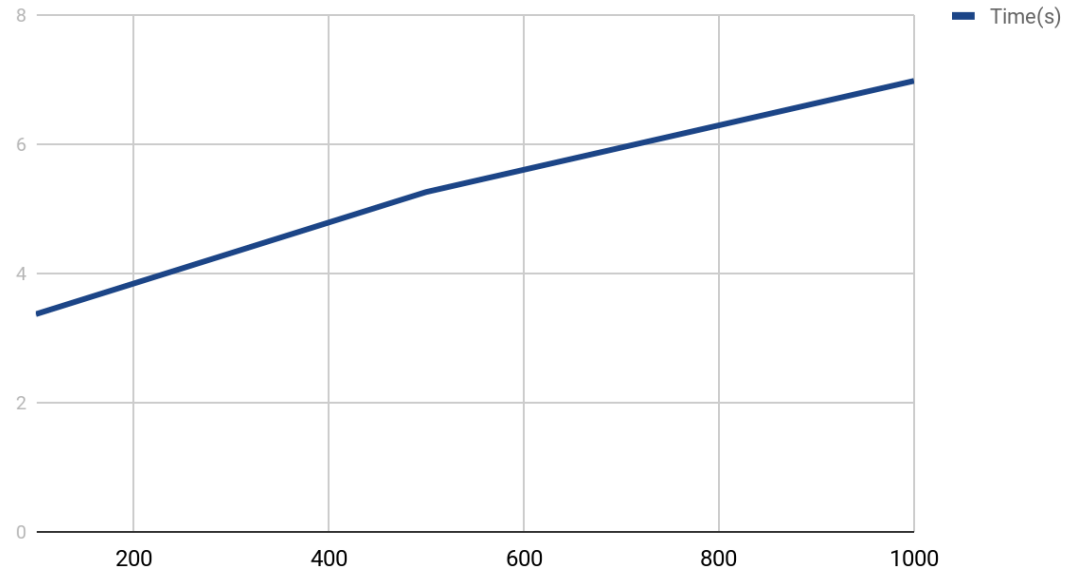
A complexidade global desta interação é O(|R|*(|Vn1| + |Vn2| + |Vn3| + 2*O(*djikstraBidirectional*))

Nota: A complexidade do *djikstraBidirectional* é referida em cima, no tópico **Algoritmos.**

Pseudocódigo:

```
firstIteration():
    initialize(wagon)
    djikstra(central)
    while still requests to process do
        request = earliest(requests)
        distPriosoner = getPathFromCentral(dest(request), path)

        dropOff = chooseDropOff()
        djikstraBidirectional(dest(request), dropOff)

        distDropOff = getPathAfterDjikstra(dropOff, path)
        djikstraBidirectional(dropOff, central)

        totalDist = distPriosoner + distDropOff
        totalDist += getPathAfterDjikstra(central, path)

        Time start = size(deliveries(wagon)) > 0 ?
            end(lastDelivery(wagon)) :
            arrrival(request) - Time(distPriosoner / velocity)

        realArrival(request) = start + Time(distPriosoner / velocity)
        setRealDeliver(getRealArrival(request) + Time(0, 0, distDropOff / velocity))

        delivery = new Delivery(...)
        wagon.insert(delivery)
```

Iteração 1 com diferente número de pedidos



2ª iteração - 1 carrinha de capacidade > 1 (|W| = 1)

Nesta iteração, decidimos melhorar a eficiência das entregas fazendo com que a carrinha (wagon) possa levar mais que um prisioneiro (fazendo vários requests de cada vez).

A implementação desta iteração também passa por inicializar a carrinha , seguido dum ciclo em que processamos os pedidos, e os agrupamos em conjuntos de N (capacidade da carrinha)(O(|R| / |N|)). Antes de sair da central é necessário calcular:

1. Agrupamentos dos pedidos, tendo em conta o tempo de espera e a distância dos pedidos entre si (O(*groupedRequests*)). A função *groupedRequests* é descrita no tópico **Estratégias de agrupamento de pedidos.**
2. Ponto de entrega do prisioneiro através da função *chooseDropOff*(), que retorna um ID de um ponto de destino (id de um vértice aleatório que não seja um vértice de destino dos pedidos agrupados) (O(|1|)). Depois disso adicionamos a um vetor os vértices dos destinos de cada 1 dos pedidos que fazem parte de 1 agrupamento (O(|N|)).

3. A hora exata a que a carrinha sai da central, para isso, comparamos o tempo de espera do primeiro pedido do grupo que a carrinha tem com a próxima hora que a carrinha vais estar disponível. Se a carrinha tiver disponível antes do tempo de espera, então a carrinha espera para sair da central de forma a chegar às horas exatas ao ponto de recolha. Se não, a carrinha sai imediatamente depois de chegar à central.

4. O percurso mais curto que passe por todos os pontos de destino dos pedidos agrupados. Este percurso começa na central e acaba no local de entrega dos prisioneiros(dropOff). Para isso implementamos uma função *tspPath()* que é descrita acima.

5. Por último, são usadas novamente as funções *getPathTo*(O(|Vf|), em que Vf representa o número de vértices pertencentes ao caminho mais curto) e *djikstraBidirectional* para calcular o caminho de regresso à central. Desta forma passamos a ter conhecimento da distância total do caminho a percorrer e podemos associar um pedido a uma entrega (Delivery).

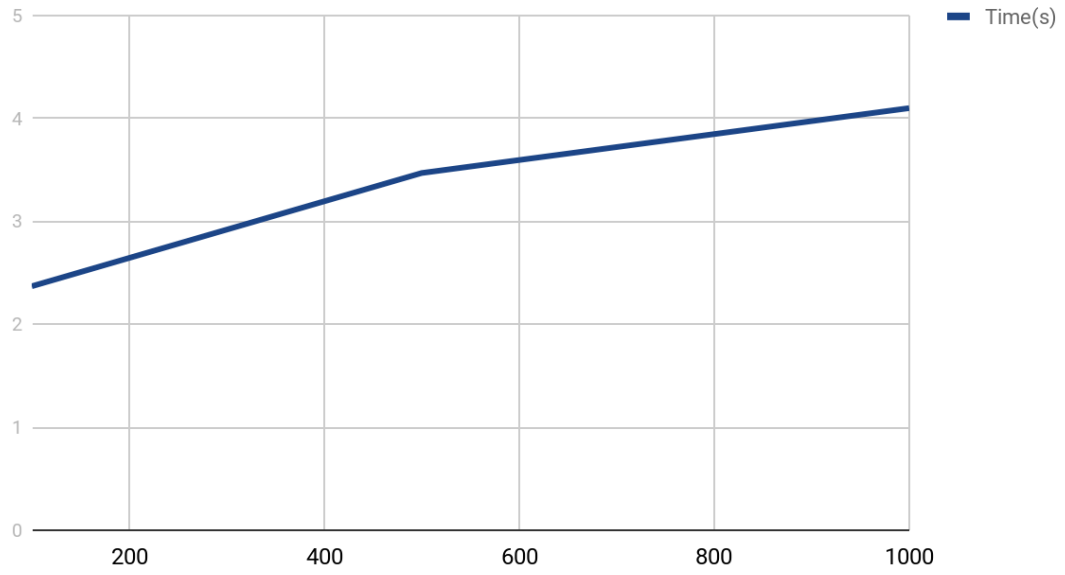
A complexidade global desta interação é O((|R| / |N|)*(O(*groupedRequests*) + |N| + O(*tspPath*) + |Vf| + O(*djikstraBidirectional*))

Nota: A complexidade do *djikstraBidirectional* é referida em cima, no tópico **Algoritmos.** A complexidade do algoritmo de *groupedRequests* é referida em cima, no tópico **Estratégias de agrupamentos de pedidos.**

Pseudocódigo:

```
secondIteration():
    initialize(wagon)
    djikstra(central)
    while requests is not empty do
        grouped = groupRequests(capacity(wagon))
        for each r in grouped do
            add(dest(r)) to tspNodes

        Time start = size(deliveries(wagon)) > 0 ?
            end(last(delivery(wagon))) :
            arrrival(first(grouped)) - Time(distPriosoner / velocity)

        dropOff = chooseDropOff(tspNodes)
        totalDist = tspPath(tspNodes, groupRequests, path, dropOff, start)

        djikstraBidirectional(dropOff, central)
        totalDist += getPathAfterDjikstra(central, path)
        wagon.setNextAvailabel(start + Time(totalDist / velocity))

        delivery = new Delivery(...)
        wagon.insert(delivery)
```

Iteração 2 com diferente número de pedidos



Escolha de Carrinha (getWagon)

Este algoritmo serve para encontrar a carrinha a ser usada entre a listas de todas as carrinhas. Para isso percorremos todas as wagons (O(|W|)) e verificamos qual delas é que está disponível mais cedo. Como os Wagons estão ordenados por capacidade, temos a certeza que, no início, escolhemos a que têm maior capacidade primeiro. A partir do momento em que todas as wagons já foram escolhidas, escolhemos a que está livre mais cedo.

A complexidade global deste método é **O(|W|)**, no pior dos casos.

Pseudocódigo:

```
getIdealWagon():
    minTime = availableTime(last(wagons))

    for each wagon in wagons do
        if(availableTime(wagon) is 0) return wagon
        if(availableTime(wagon) < than min_time) do
            it = wagon
            minTime = (availableTime(wagon)

    if(availableTime(first(wagons)) <= 0) return first(wagons)
    if(availableTime(first(wagons)) < minTime) return first(wagons)
```

3ª iteração - pelo menos 2 carrinha (|W| > 1)

Nesta iteração, decidimos melhorar ainda mais a eficiência das entregas utilizando mais do que uma carrinha (wagon) que possa levar mais que um prisioneiro (fazendo vários requests de cada vez).

A implementação desta iteração também passa por inicializar a carrinha , seguido dum ciclo em que processamos os pedidos, e os agrupamos em conjuntos de N (capacidade da carrinha)(O(|R| / |N|)). Antes de sair da central é necessário calcular:

1. Percorrer a lista de carrinhas que temos e escolhemos a melhor consoante a disponibilidade no momento ou a que tiver disponibilidade mais cedo.

2. Agrupamentos dos pedidos, escolhendo o pedido com menor tempo de espera e depois os requests mais pertos desse pedido (O(*groupedRequests*)).

3. Com tudo isso feito, calculamos a hora exata a que a carrinha sai da central comparando o tempo de espera do primeiro request atribuído à carrinha com a próxima hora que a carrinha vais estar disponível. Se a carrinha tiver disponível antes do tempo de espera, então a carrinha espera para sair da central de forma a chegar às horas exatas ao ponto de recolha. Se não, a carrinha sai imediatamente depois de chegar à central.

4. O percurso mais curto que passe por todos os pontos de destino dos pedidos agrupados. Este percurso começa na central e acaba no local de entrega dos prisioneiros(dropOff). Para isso implementamos uma função *tspPath()* que é descrita no tópico **?**.

5. De seguida, são usadas novamente as funções *getPathTo*(O(|Vf|)) e *djikstraBidirectional* para calcular o caminho de regresso à central. Desta forma passamos a ter conhecimento da distância total do caminho a percorrer e podemos associar um pedido a uma entrega (Delivery).

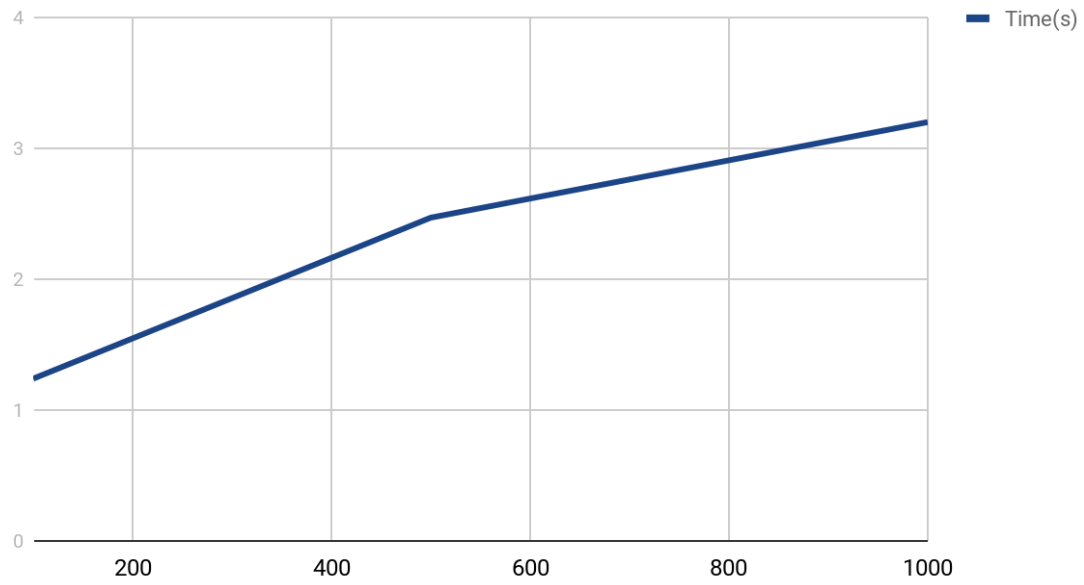
A complexidade global desta interação é O((|R| / |N|)*(O(*groupedRequests*) + O(*getIdealWagon*) + |N| + O(*tspPath*) + |Vf| + O(*djikstraBidirectional*))

Nota: A complexidade do *djikstraBidirectional* é referida em cima, no tópico **Algoritmos.** A complexidade do algoritmo de *groupedRequests* é referida em cima, no tópico **Estratégias de agrupamentos de pedidos.** A complexidade de *getIdealWagon* é referida acima.

Pseudocódigo:

```
thirdIteration():
    for each wagon in wagons do
        initialize(wagon)

    djikstra(central)
    while requests is not empty do
        wagon = getIdealWagon()
        grouped = groupRequests(capacity(wagon))
        for each r in grouped do
            add(dest(r)) to tspNodes

        Time start = size(deliveries(wagon)) > 0 ?
            end(last(delivery(wagon))) :
            arrrival(first(grouped)) - Time(distPriosoner / velocity)

        dropOff = chooseDropOff(tspNodes)
        totalDist = tspPath(tspNodes, groupRequests, path, dropOff, start)

        djikstraBidirectional(dropOff, central)
        totalDist += getPathAfterDjikstra(central, path)
        nextTimeAvailable(wagon) = start + Time(totalDist / velocity)

        delivery = new Delivery(...)
        wagon.insert(delivery)
```

Iteração 3 com diferente número de pedidos



CASOS DE UTILIZAÇÃO

A comunicação entre a aplicação e o utilizador será feita através de uma interface simples de texto. Esta interface será composta por um sistema de menus com várias opções descritas abaixo.

- Importar um grafo através dum ficheiro de texto;
- Calcular o caminho mais curto entre quaisquer dois pontos num grafo;
- Calcular a rota ótima entre vários pontos;
- Mudar a posição da central;
- Listar pedidos;
- Listar veículos disponíveis na central e as respetivas capacidades;
- Adicionar e remover um veículo à central (para testar diferentes iterações sem terminar a execução do prog);
- Verificar se o grafo é fortemente conexo (pré processamento de grafo);

Menu

O aspeto do menu pode ser visto em baixo. Tendo várias opções com diferentes complexidades apresentamos em baixo um pequeno tutorial sobre o mesmo:

1. Esta opção (única visível ao iniciar o programa) permite ler um grafo associado a uma cidade qualquer, desde que esteja na pasta maps/PortugalMaps.
2. Permite remover partes do grafo (vértices e arestas) que não façam parte do mesmo componente fortemente conexo a que a central pertence.
3. Permite calcular o caminho mais curto entre 2 pontos do grafo lido, usando um dos 3 algoritmos disponíveis e fornecendo os IDs dos vértices.
4. Permite testar as 3 iterações feitas, ou seja, testa as entregas de diferentes pedidos dos prisioneiros na cidade em questão.
5. Encarrega-se de alterar o vértice marcado como **central** para o grafo lido.
6. Leva ao sub-menu onde é possível realizar operações sobre carrinhas (adicionar, remover e listar carrinhas).
7. Permite listar na consola os pedidos associados ao mapa da cidade em questão.
8. Permite ligar ou desligar a visualização do grafo em casos não fundamentais de visualização de solução, i.e., ler o grafo ou pré processá-lo.

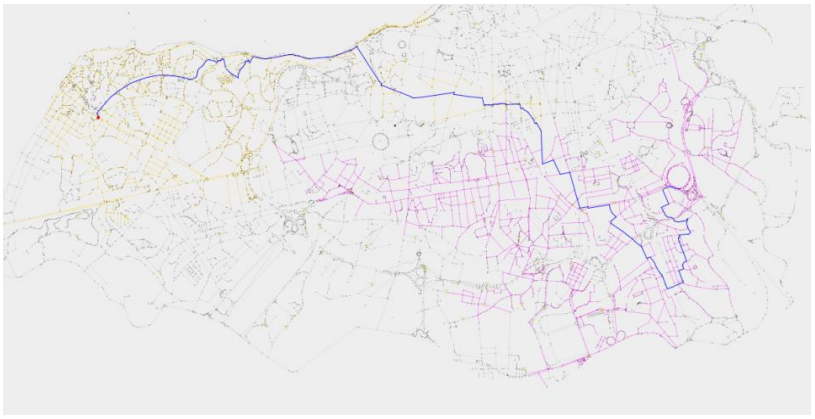
A opção 0 leva ao término do programa com código de saída 0. Qualquer outro *input* do utilizador será ignorado, sendo que o programa espera até ler um input válido. Em qualquer sub-menu do programa é possível viajar para trás(menu anterior) escrevendo como input a string '**back**'.

```
----------------------------------------------------------------------
Menu Options: [Type 'back' to go back in any menu]
1 - Read Graph
2 - Pre Process
3 - Shortest Path
4 - Deliver
5 - Set Central
6 - Wagon Operation
7 - List Requests
0 - Exit

Graph read for: 'Porto'
Central node ID: 90379359
----------------------------------------------------------------------
Input: >
```

```
Input: >3

--- Shortest Path ---
1 - Classic Dijkstra
2 - Oriented Dijkstra (A*)
3 - Bidirectional Dijkstra
Input: >1


Provide <origin node> <destination node> [Example: 90379359 411018963]
Input: >
```

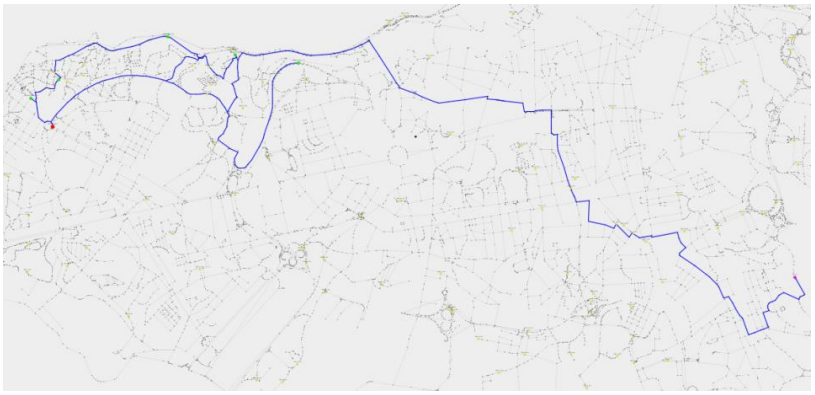
Visualizador

Tentámos ser o mais expressivos possíveis na demonstração de resultados através do visualizador. Relativamente ao teste de algoritmos de caminho mais curto, em todos os dijkstra, temos a cor amarela para demonstrar as arestas exploradas e o caminho azul revela o caminho efetivamente mais curto. Como no caso do dijkstra Bidirectional exploramos dois caminhos através de 2 filas de prioridade mínima, achamos interessante demonstrar a diferença entre estes 2 caminhos (o caminho a magenta demonstra o caminho percorrido desde o ponto final até ao ponto de interseção).

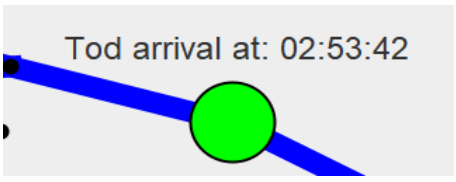


Visualização Djikstra Bidirecional

Relativamente à demonstração das entregas, o caminho azul revela o caminho percorrida, os nós a verde os nós onde os prisioneiros são recolhidos e o nó magenta o local onde os prisioneiros são deixados, depois de recolhidos. Cada nó tem também uma label que indica a que hora o prisioneiro foi recolhido, sendo que esta informação está também disponível na consola, para melhor análise.



Visualização de Entregas

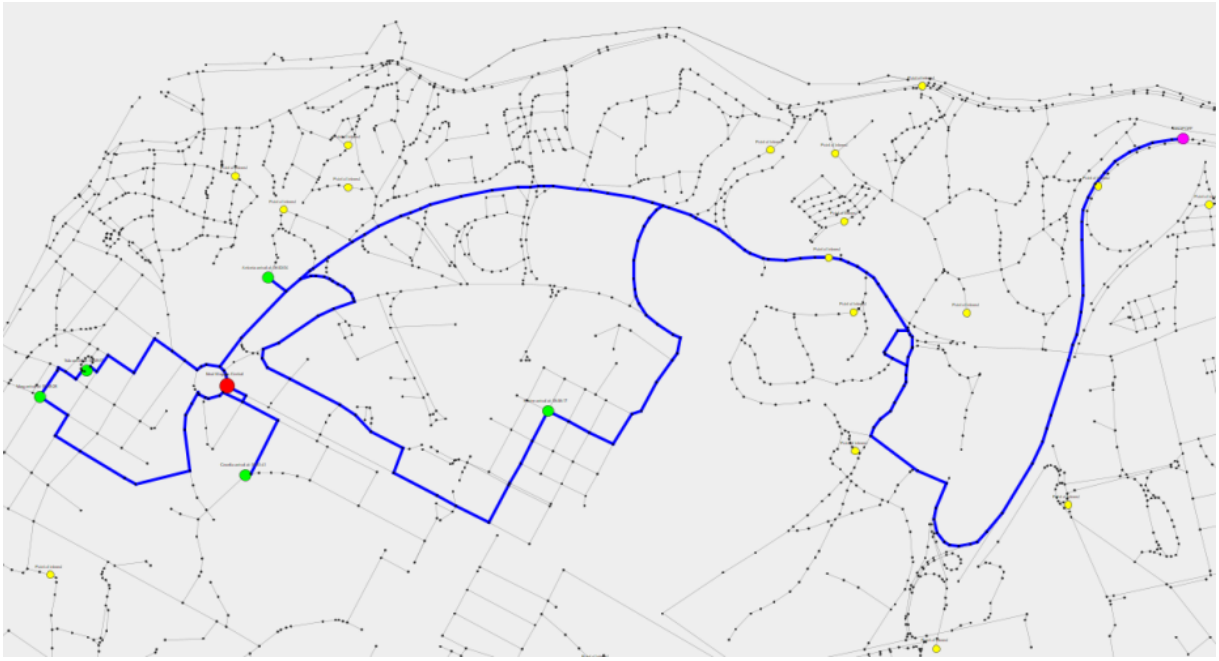


Threads

Uma vez que o programa de visualização do grafo é um pouco lento, decidimos implementar threads para o desenho de qualquer grafo no visualizador. Isto faz com que o menu não fique muito tempo parado, sendo possível, enquanto que o grafo é desenhado, executar outras operações.

```
Delivery* MeatWagons::drawDeliveriesFromThread(int wagonIndex, int deliveryIndex) {
    thread threadProcess(&MeatWagons::drawDeliveries, this, wagonIndex, deliveryIndex);
    threadProcess.detach();
    return  next( _First: this->wagons.begin(), wagonIndex)->getDeliveries().at(deliveryIndex);
}
```

Entrega de prisioneiros



A central é representada através da cor vermelha. É daqui que a central parte para ir recolher os prisioneiros.

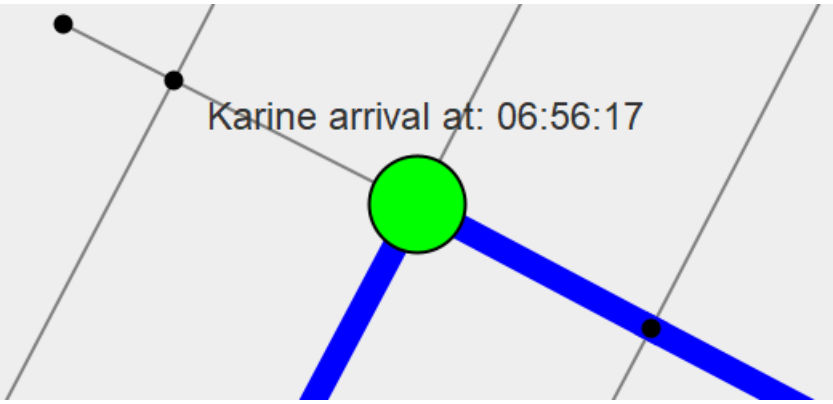
Os pontos verdes representam os locais onde os prisioneiros são coletados. é possível através da label ver o nome do prisioneiro e a que horas este é coletado. O ponto rosa representa o ponto onde os prisioneiros, depois de agrupados, são entregues.

A carrinha depois faz o percurso desde desse ponto até à central, para continuar o serviço.

```
Wagon[0] leaves central at: 06:48:07
Wagon[0] returns to central at: 07:11:36

Requests done:
| Prisoner: Tula        | Priority: 2  | Arrives at: 06:49:07 | Delivered at: 07:03:37 |
| Prisoner: Antonia     | Priority: 2  | Arrives at: 06:53:04 | Delivered at: 07:03:37 |
| Prisoner: Cecelia     | Priority: 1  | Arrives at: 06:51:41 | Delivered at: 07:03:37 |
| Prisoner: Karine      | Priority: 1  | Arrives at: 06:56:17 | Delivered at: 07:03:37 |
| Prisoner: Marg        | Priority: 2  | Arrives at: 06:49:28 | Delivered at: 07:03:37 |
```

Através da consola é possível observar-se melhor os tempos e como são feitas às entregas. Neste caso, a carrinha sai da central às 06:48:07 e vai buscar depois o prisioneiro Tula às 06:49:07. Os outros prisioneiros são coletados até á última prisioneira, a Karine. Esta é coletada às 06:56:17. Depois os presos são deixados no ponto magenta às 07:03:37. A carrinha depois volta à central e só chega às 07:11:36.

É possível também através das labels ver a que horas o prisioneiros são apanhados.



Através do menu é possível escolher qualquer uma das carrinhas usadas na iteração selecionada, e para cada uma, escolher uma entrega, sendo demonstrada claramente os valores de limite que podem ser escolhidos no input

```
--- Choose Wagon --- [0, 1]
Wagon Index >0

--- Choose a Delivery done by Wagon #0 --- [0, 6]
Delivery Index >0
```

CONCLUSÃO

Ao analisar o trabalho realizado, é possível ver que conseguimos, com sucesso, concluir todos os objetivos que tínhamos planeado para o projeto. Foi realizado através da divisão em 3 iterações diferentes na sua dificuldade e na sua abordagem, mas essa estratégia facilitou a incorporar diferentes funcionalidades no programa.

Conseguimos implementar algumas resoluções para o *Travelling Salesman Problem* utilizando otimização do algoritmo de Dijkstra, incluindo A* e Dijkstra Bidirecional. Todos os algoritmos são de nossa autoria, sendo o bidirectional implementado com verificação (que não se encontra em muitos algoritmos) do caminho mais curto, pois nem sempre o ponto em que ambas as pesquisas se encontram pode não pertencer ao caminho mais curto. Isto é referido na **referência 1.**

Foram identificados os vários problemas inerentes ao tema, tendo sido expostos com detalhe e apresentadas diferentes propostas de solução para os resolver. Estas foram baseadas através do nosso conhecimento adquirido através das aulas teóricas de CAL, assim como uma profunda pesquisa alargada, fora do âmbito da unidade curricular.

Consideramos, para todas as iterações, a complexidade teórica e empírica dos algoritmos desenvolvidos. Para isto foram feitos diferentes testes, efetuando pequenas variações nos algoritmos, de modo a obtermos a melhor combinação de resultados, nomeadamente, no agrupamento de pedidos, escolha do algoritmo mais curto entre 2 pontos, etc.

Podemos destacar alguns problemas com que nos deparámos, tais como, caminho mais curto entre 2 pontos, caminhos mais curto entre todos os pares de vértices, caminho mais curto passando por uma sequência de vértices, análise de conectividade, utilização de diferentes estratégias de agrupamento de pedidos, utilização do graphViewer, utilização de threads, codificação de um menu com verificação de input, problemas de escalabilidade e generalização.

Houveram algoritmos que foram estudados, e projetados implementar, no entanto, devido aos mapas recebidos inicialmente, conclui-se que estes não seriam úteis devido à utilização de um grafo bidirecional. Destes conceitos estudados, mas não implementados, destacam-se os algoritmos de Kosaraju e Tarjan, para a parte de análise de conectividade e algoritmos da resolver o problema do TSP tais como brute force e Held-Karp.

Assim, com este trabalho, conseguimos conciliar a matéria dada nas aulas teóricas de uma forma muito eficaz. Sentimos que este trabalho foi muito proveitoso nesse sentido e que nos ensinou coisas fundamentais para o nosso futuro, especialmente técnicas de cálculo de caminhos mais curto e de análise de grafos. Desta matéria destacam-se os conceitos utilizados de bruteforce, recursividade, programação dinâmica, algoritmos gananciosos, divisão e conquista, retrocesso.

Também conseguimos aplicar e melhorar os nossos conhecimentos de análise de algoritmo e a sua posterior otimização algo que, neste momento, é algo muito importante para a conceção de aplicações.

Contribuição

Cada membro do grupo dedicou um esforço mais ou menos similar a esta primeira entrega do projeto, tendo a divisão das tarefas sido feita da seguinte forma (⅓ para cada):

Martim Pinto da Silva

- Descrição do problema (relatório), Pré-processamentos (relatório e implementação), Algoritmo de Dijkstra (e variantes) (relatório), Algoritmo de Dijkstra original e A* (implementação), Algoritmo de Floyd-Warshall (relatório), Estratégias de aplicação dos algoritmos (relatório), Casos de utilização (relatório), Iteração 1(implementação), Iteração 2(implementação), Iteração 3(implementação), Conclusão(relatório), Bibliografia(relatório), Interface gráfica do visualizador de grafos (implementação), Conectividade dos grafos utilizados(relatório), Comparação dos algoritmos de caminho mais curto entre 2 vértices(relatório), Conectividade e pré-processamento do grafo(relatório), Pré-processamento das reservas(relatório), tspPath(implementação e relatório), Pseudocódigo e análises temporais.

Bernardo

- Formalização do problema (relatório), Implementação de Dijkstra Bidirectional, Implementação dos tempos certos nas entregas, Iteração 1(implementação), Iteração 2(implementação), Iteração 3(implementação), Estratégias de agrupamentos de pedidos (relatório), Função objetivo, Comparação de diferentes estratégias de agrupamentos(relatório), getWagon(relatório e implementação)

Francisco

- Capa(relatório), Casos de utilização (relatório), Estruturas de dados e classes usadas(relatório), Análise da conectividade (relatório), Menu (implementação), Iteração 1(implementação)

BIBLIOGRAFIA

Dijkstra:

Speeding up Dijkstra (YouTube) - **referência 1**

Bidirectional Search (GeeksForGeeks)

Bidirectional Dijkstra - Advanced Shortest Paths

Bidirectional Dijkstra vs Dijkstra (Stack Exchange Forum)

The Shortest Path Problem - Bidirectional Dijkstra's / Alt / Reach (YouTube)

Dijkstra vs Bi-directional Dijkstra Algorithm on US Road Network (YouTube)

Dijkstra's Algorithm - Computerphile (YouTube)

Dijkstra Explained (YouTube)

Dijkstra's algorithm in 3 minutes — Review and example (YouTube)

TSP:

Traveling Salesman Problem Visualization (YouTube) - **referência 2**

Traveling Salesperson Problem (Wikipedia)

Travelling Salesman Problem | Dynamic Programming | Graph Theory (YouTube)

Nearest Neighbour Algorithm (Wikipedia)

Conectividade:

Strongly Connected Components (YouTube)

Tarjan's Strongly Connected Components algorithm | Graph Theory (YouTube)

Tarjan's strongly connected components algorithm (Wikipedia)

Strongly connected component (Wikipedia)