FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEIC 2º Ano

CAL

Maio 2020

Fix It: Reparações ao Domicílio

Turma 6 - Grupo 2

Pedro Miguel Novais do Vale - up201806083

Rui Filipe Mendes Pinto - up201806441

Tiago Gonçalves Gomes – up201806658

Índice

Parte I

Descrição do Tema

Uma empresa de reparações recebe muitos pedidos de clientes residenciais para diversos tipos de reparação, incluindo torneiras com folga, esquentadores avariados falhas elétricas, etc. De acordo com cada uma dessas ocorrências, é marcado um horário sobre o qual a reparação poderá ser efetuada. Essa mesma é efetuada por diversos piquetes, tendo cada um um conjunto de competências distintas. Neste projeto, pretende-se conjugar essas atividades por todos os piquetes, considerando diversas variáveis que combinadas tentarão maximizar ao máximo o lucro da empresa.

Iterações do Problema

A resolução deste problema pode ser decomposta nas seguintes iterações.

Iteração 0: pré-processamento

1. Cálculo dos valores da tabela auxiliar, que será utilizada para acelerar o algoritmo em si (só necessário da primeira vez que o algoritmo é executado);
2. Remoção das arestas indesejáveis para a execução do algoritmo:
    - **1ª iteração:** remoção das que representam vias inutilizáveis pelos veículos (estas serão indicadas no grafo com peso = 0).
    - **2º iteração:** No caso de haver mais que uma aresta com a mesma origem e destino, remoção das que não têm peso mínimo (caso improvável, mas possível).

Iteração 1: apenas um piquete

Esta iteração pode ser dividida em duas partes:

Iteração 1.1: Um piquete, uma função, melhor trajeto entre vários pontos POIs (Points Of Interest)

Numa primeira fase, considera-se que a empresa é composta apenas por um piquete, que tem apenas uma função, e que não existem horários para cada trabalho. Assim, o objetivo é apenas de determinar qual o caminho mais curto que começa na empresa, que passa por todos os pontos de interesse e que, por fim, regresse ao ponto de partida. Desta forma tentaremos minimizar o combustível e o tempo gasto pelo piquete, de forma a maximizar os lucros da empresa.

É importante notar que uma viagem entre dois pontos de interesse só pode ser efetuada se existirem caminhos que liguem esses pontos, em ambos os sentidos. Portanto, todos os pontos de interesse devem fazer parte do mesmo componente conexo do grafo. Esta necessidade advém do facto que, da empresa, o piquete deve conseguir alcançar todos os pontos de interesse e, depois, deve poder regressar à empresa. Assim, torna-se fundamental efetuar uma análise da conectividade do grafo.

Certas vias de comunicação podem não poder ser usadas pelo piquete, devido à existência de obras nas vias públicas, pelo que será necessário desprezar certas arestas durante o processamento do grafo.

Iteração 1.2: Adição de Escalonamento de Atividades e várias funções para o piquete

Numa segunda fase, considera-se apenas um piquete, que pode ter mais do que uma função, bem como o escalonamento de atividades, de modo a respeitar o horário requerido por cada cliente. Caso não seja possível incorporar o pedido de reparação no horário de um piquete, procede-se à contratação dos serviços de um novo piquete (abordado na iteração seguinte). Deste modo, tentaremos efetuar o máximo de atividades possíveis para o menor número de piquetes.

Com esta estratégia, tentaremos maximizar os lucros da empresa, uma vez que se o escalonamento das tarefas for realizado corretamente, isso implicará que menos trabalhadores serão contratados e o tempo será utilizado de uma forma mais rentável.

Iteração 2: Vários piquetes onde é combinado o escalonamento de atividades

Mais do que um piquete, cada um com várias funções. Aqui o objetivo passa por combinar a atividade dos vários piquetes, de modo a permitir o máximo possível de clientes atendidos, como referido na iteração anterior.

Com isto deverá ser conjugada a funcionalidade desenvolvida na iteração 1.1 que passa por determinar o melhor trajeto possível para vários POIs. A par disso pretende-se que a escolha de um piquete para fazer um determinado trabalho seja feita de forma otimizada em conjugação com as tarefas dos restantes piquetes.

Dados de Entrada

FW - tabela com valores das distâncias/tempos mínimos entre cada par ordenado de vértices, e possivelmente o caminho que leva a essas distâncias/tempos;

Pi - Sequência de piquetes da empresa, sendo Pi(i) o seu i-ésimo elemento. Cada um é caracterizado por:

- Rj - Conjunto de funções do piquete, sendo Rj(j) o seu j-ésimo elemento.
- Experiência (número de serviços prestados à empresa)

Ci - Conjunto de pedidos feitos pelos clientes, caracterizados por:

- ID - Identificador do pedido (não pode ser gpsCoords, já que um cliente pode fazer mais do que um pedido).
- gpsCoords - coordenadas GPS relativas ao ponto do mapa onde o pedido será atendido
- Tipo (semelhante ao *role*)
- Horário (intervalo de tempo em que o cliente se encontra disponível para receber o piquete: ~3h)

Gi = (Vi, Ei) - Grafo dirigido pesado, que representa o mapa da cidade. Este grafo é composto pelo conjunto de vértices V e arestas E.

Vi - vértices (que representam pontos da cidade) com:

- gpsCoords - coordenadas GPS do ponto do mapa
- Adj $\subseteq$ E - conjunto de arestas que partem do vértice

Ei - aresta (representa ruas, pontes e outras vias) é caracterizada por:

- w —> peso da aresta, que representa o tempo/distância do percurso entre os dois vértices que esta liga;
- ID —> identificador único de uma aresta;
- dest $\in$ V - vértice de destino da aresta;

S $\in$ **Vi** —> vértice inicial, que representa a central de onde os veículos saem.

T $\subseteq$ **V** - vértices finais (representam as moradas dos clientes)

Dados de Saída

Gf = (Vf, Ef) - grafo dirigido pesado, sendo Vf e Ef os mesmos atributos que V e E.

Pf - Sequência de piquetes da empresa. Cada um é caracterizado por:

- Atividades atribuídas (conjunto de tarefas / ID do pedido)
- Experiência (número de serviços prestados à empresa, sendo o mesmo incrementado do número de atividades efetuadas)
- Path = {e ∈ Ei | 1 ≤ j ≤ |P|} - sequência ordenada de arestas a visitar, sendo e_j o seu j-ésimo elemento. Representa o percurso efetuado relativo às tarefas atribuídas.

Restrições

Os **dados de entrada**, apresentam as seguintes restrições:

∀ i ∈ [1; Pi.size()] :

- Rj(Pi[i]) = { Rj ⊆ Roles | Roles = {“Eletricista”, “Picheleiro”, “Canalizador”, “Trolha”, Técnico Informático”}} - O tipo de função do piquete deverá ser eletricista, picheleiro, canalizador, trolha e/ou técnico informático.
- Experiência(Pi[i]) >= 0) - O número de serviços prestados à empresa não pode ser negativo.

∀ i ∈ [1; Ci.size()] :

- Tipo(Ci[i]) = { Tipo ∈ Roles | Roles = {“Eletricista”, “Picheleiro”, “Canalizador”, “Trolha”, Técnico Informático”}} - O tipo de um pedido deverá ser eletricista, picheleiro, canalizador, trolha ou técnico informático.

∀ e ∈ Ei, w > 0, dado que o peso da aresta representa um tempo/distância entre dois pontos do mapa.

Seja o conjunto L = {vert ∈ V | vert = S ⋁ vert ∈ T}. É preciso que todos os elementos de L façam parte do mesmo componente fortemente conexo de um grafo, de modo a que não exista qualquer ponto de articulação. Isto implica que ao retirarmos um qualquer vértice do grafo nessa região (situação de interrupção das vias de circulação), os restantes vértices são na mesma alcançáveis.

Os **dados de saída**, apresentam as seguintes restrições:

Grafo Gf:

- ∀ vf ∈ Vf, ∃ vi ∈ Vi tal que vi e vf têm os mesmos valores para todos os atributos.
- ∀ ef ∈ Ef, ∃ ei ∈ Ei tal que ei e ef têm os mesmos valores para todos os atributos.

|Pf| <= |Pi| - uma vez que o número de piquetes, cujos serviços foram contratados, poderá ser igual ou inferior ao número de piquetes declarados inicialmente.

Relativamente ao Path de Pf:

- Se e_1 é o 1º elemento do Path de Pf, então e_1 ∈ Adj(S), pois o piquete parte da sede da empresa.
- Se e_2 é o último elemento do Path de Pf, então dest(e_2) = S, pois o piquete termina o seu serviço também na sede da empresa.

Função Objetivo

A solução ótima do problema passa por minimizar o número de piquetes usados e a distância total percorrida por estes, de forma a que todos os clientes sejam atendidos nos horários marcados e a maximizar os lucros da empresa. Portanto, a solução ótima passa pela minimização das funções:

f = |Pf|

$$g = \sum_{p \in Pf} \sum_{\text{e} \in \text{Path}} w(e)$$

Tal como referido na descrição do problema, irá ser privilegiada a minimização da função f sobre a da função g.

Perspetiva de Solução

Com vista em resolver o problema enunciado, temos por base a realização dos seguintes procedimentos:

Pré-processamento

1. **Cálculo dos valores do tempo/distância** e armazenamento numa estrutura de dados auxiliar, possivelmente uma matriz. É apenas necessário correr este algoritmo no início do programa, de modo a utilizar os valores nas operações seguintes.
2. **Remoção das arestas indesejáveis**. Pretende-se remover todas as arestas de peso igual a zero, pois as mesmas representam vias inutilizáveis pelos piquetes. Por fim, no caso de haver arestas com a mesma origem e destino, deverá proceder-se à eliminação do "excedente".

Relativamente ao primeiro tópico teremos duas opções para o cálculo destes valores auxiliares. A utilização repetida do algoritmo de **Dijkstra** para cada vértice do grafo, algo que é feito quando o grafo é esparso (**$|E| = O(V)$**). A complexidade temporal deste algoritmo é **$O(|V| * |E| * \log(V)$**. A segunda opção passa pela execução do algoritmo de **Floyd-Warshall**, algo que é feito quando o grafo é mais denso **($|E| = O(V^2)$**). A complexidade temporal deste algoritmo é **$O(V^3)$**.

Componente Fortemente Conexa (CFC)

Neste tópico pretende-se analisar todos os POIs (moradas dos clientes e da sede da empresa). Assim, verifica-se se o conjunto dos POIs selecionados pertencem todos à mesma componente fortemente conexa do grafo. Uma vez que teremos um grafo dirigido em mãos, utilizaremos o método lecionado nas aulas teóricas.

"Pesquisa em profundidade no grafo G determina floresta de expansão, numerando vértices em pós-ordem (ordem inversa de numeração em pré-ordem). Inverter todas as arestas de G (grafo resultante é Gr). Segunda pesquisa em profundidade, em Gr, começando sempre pelo vértice de numeração mais alta ainda não visitado. Cada árvore obtida é um componente fortemente conexo, i.e., a partir de um qualquer dos nós pode chegar-se a todos os outros." R. Rossetti, L. Ferreira, H. L. Cardoso, F. Andrade.

Após obter todas as CFCs, verificamos se o conjunto dos POIs pertence todo à mesma CFC. Caso isso não aconteça poderemos estar perante um trajeto impossível. Obviamente que o trajeto em si também dependerá da rota que cada piquete terá de percorrer. Tal como foi referido, convém que todos os POIs do seu percurso respeitem a regra enunciada acima.

Verificação de um trajeto entre POIs

Inicialmente deverá ser analisada a possibilidade de a partir de um vértice de origem chegar a um POI. Para isso, existem dois algoritmos principais, denominados **Pesquisa em Profundidade** (*depth-first search*) e **Pesquisa em Largura** (*breadth-first search*).

No algoritmo de **Pesquisa em Profundidade**, as arestas são exploradas a partir do vértice mais recentemente descoberto que ainda tenha arestas a sair dele. É uma pesquisa inerentemente recursiva e que pode ser implementada recorrendo a uma estrutura de dados auxiliar, de modo a armazenar os vértices já visitados, por exemplo, uma pilha. Uma vez que numa tabela de dispersão a procura de um elemento é feita em tempo constante (**O(1)**), os vértices já visitados são colocados numa tabela de dispersão. Caso o vértice de destino da nossa pesquisa estiver na tabela no final do processamento, então isso implica que há uma conectividade e que é possível navegar desde a origem até ao destino.

```
1   DFS(G, v):                  // G - Grafo, v - Origem
2       visitedNodes <- {}      // Hash Table com vértices visitados
3       DFS-VISIT(G, v)
4       return visitedNodes
5
6
7   DFS-VISIT(G, v):
8       if notVisitedNode(v) {
9           insert(VisitedNodes, v)
10          for each w in Adj(v)
11              DFS-VISIT(G, w)
12      }
```

A **complexidade temporal** deste algoritmo é **O(|V| + |E|)**. Cada vértice é visitado, no máximo, uma vez e a pesquisa é posteriormente realizada para todos os vértices de destino das suas arestas. Relativamente ao espaço utilizado, no pior caso, irão ser armazenados |V| vértices na tabela de dispersão. A **complexidade espacial** deste algoritmo é **O(|V|).**

Por fim, relativamente ao algoritmo de **Pesquisa em Largura**, dado um vértice de origem *s*, explora-se sistematicamente o grafo descobrindo todos os vértices a que se pode chegar a partir de *s* (vértices adjacentes). Só de seguida é que se processa outro vértice. Esta pesquisa é feita recorrendo a uma fila.

```
1   BFS(G, s):                  // G - Grafo, s - Origem
2       visitedNodes <- {}      // Hash Table com vértices visitados
3       Q <- {}                 // Queue dos vértices
4
5       insert(visitedNodes, s)
6       push(Q, s)
7
8       while !empty(visitedNodes) do
9           v <- EXTRACT-MIN(Q)
10
11          for each w in Adj(v) do
12              if not discovered(w) then
13                  push(Q, w)
14                  insert(visitedNodes, w)
15
16      return visitedNodes
17
```

A **complexidade temporal** deste algoritmo é, tal como no algoritmo de Pesquisa em Profundidade, **O(|V| + |E|)**. Cada vértice é, também, visitado no máximo apenas uma vez. Relativamente ao espaço utilizado no pior caso, a fila que armazena os vértices ao longo de execução do algoritmo terá |V| elementos. A complexidade espacial deste algoritmo é **O(|V|)**.

Vimos que a complexidade de ambos os algoritmos é a mesma. Porém, qual é a melhor a utilizar? Isso depende bastante da estrutura do nosso grafo. Se sabemos que a nossa solução não está longe da raiz, então a **Pesquisa em Largura** poderá ser a melhor opção a utilizar. Se o grafo for bastante profundo e as soluções forem escassas, então a **Pesquisa em Largura** deverá ser utilizada. Se a árvore formada pelo grafo for bastante larga, a **Pesquisa em Largura** necessitará de bastante memória para operar, resultando em algo impraticável. Por fim, caso as soluções sejam frequentes mas estejam bastante no fundo da árvore formada pelo grafo, então a **Pesquisa em Profundidade** será aconselhável, uma vez que a **Pesquisa em Largura** será também impraticável.

Melhor trajeto entre POIs

O melhor trajeto entre dois locais, no que diz respeito a uma rede de grafos, é já um problema ancestral. Existem diversos algoritmos já estudados que resolvem esta questão: algoritmo de **Dijkstra** e um "*improvement*" do mesmo, o algoritmo **A* (A-Star)**. No caso, o algoritmo A* utiliza uma heurística de otimização capaz de reduzir o tempo de execução do algoritmo. Passaremos de seguida para a explicação de cada um deles e das suas variantes.

Algoritmo de Dijkstra

O algoritmo de Dijkstra, concebido pelo holandês Edsger Dijkstra em 1956 e publicado em 1959, tem o objetivo de calcular o melhor caminho entre quaisquer dois vértices de um grafo dirigido ou não dirigido, com arestas de peso não negativo. Contudo, tendo em conta o contexto do nosso problema, iremos fixar o algoritmo a encontrar apenas o melhor caminho desde um vértice de origem até a um vértice de destino.

Este método é executado por etapas, sendo que se obtém uma melhor solução a cada etapa, sem considerar consequências futuras. No final do algoritmo, obtemos a melhor solução possível.

Pseudocódigo deste algoritmo:

```
1   Dijkstra(G, s):                 // s - Vértice de origem
2       for each vertex in V do
3           dist(vertex) <- INF
4           path(vertex) <- nil
5
6       dist(s) <- 0
7       Q <- ∅ // min-priority queue
8       INSERT(Q, (s, 0)) // inserts s with key 0
9
10      while Q ≠ ∅ do
11          vertex <- EXTRACT-MIN(Q) // greedy
12
13          for each edge in Adj(vertex) do
14              if dist(edge) > dist(vertex) + weight(vertex, edge) then
15                  dist(edge) <- dist(vertex) + weight(vertex, edge)
16                  path(edge) <- vertex
17
18                  if edge in Q then // old dist(edge) was INF
19                      INSERT(Q, (edge, dist(edge)))
20                  else
21                      DECREASE-KEY(Q, (edge, dist(edge)))
22
```

O algoritmo pode ser dividido em duas etapas principais:

1. **Inicialização** - cada vértice começa com uma distância infinita, e sem nenhum *path* associado. Ao primeiro vértice é atribuído uma distância de zero e adiciona-se o mesmo a uma estrutura de dados do tipo *min-priority queue*. Esta estrutura vai conter todos os vértices que já foram encontrados e manter o de menor distância preparado para ser processado a seguir.

2. **Processamento** - enquanto não encontramos o vértice de destino ou não tenhamos percorrido todos os vértices do grafo, o algoritmo verifica para cada vértice "vizinho" (*N*) do vértice atual (*V*) se a sua distância melhora/diminui a ir de *V* para *N*; se isso acontecer, a distância e o *path* de *N* são atualizados, e a *priority queue* é rearranjada; caso contrário, não são efetuadas nenhumas mudanças.

O algoritmo de **Dijkstra** irá percorrer uma enorme quantidade de vértices, já que há medida que nos vamos afastando do início, as distâncias aumentam e o algoritmo tem constantemente que retroceder para um vértice anterior para verificar se existe um melhor caminho. Portanto, este algoritmo dá-nos sempre a melhor solução possível. No entanto, esta busca pela perfeição tem um enorme custo de *performance*, principalmente se o grafo for muito denso, pelo que é preferível usar outros algoritmos, como o A*, apresentado a seguir. A complexidade temporal do algoritmo de Dijkstra é **O((|V| + |E|) * log|V|)**, e a espacial é **O(|V|)**.

Algoritmo A* (A-star)

Como referido anteriormente, este algoritmo, considerado uma extensão do algoritmo de Dijkstra, foi inicialmente introduzido por Peter Hart, Nils Nilsson e Bertram Raphael, no ano 1968. O algoritmo obtém um melhor desempenho devido a uma heurística que o permite alcançar o vértice de destino, avançando no sentido de diminuir a distância do vértice atual ao de destino.

Pseudocódigo deste algoritmo (próxima página):

```
A_Star(G, Vi, Vf):              // G - Grafo, V - Vértices, Vi - Vértice inicial,
                                // Vf - Vértice final, Heuristic - função de cálculo de distâncias
    for each v in V:
        distance(v) <- INF
        path(v) <- INF
        gScore(v) <- INF

    dist(Vi) <- 0
    gScore(Vi) <- Heuristic(Vi, Vf)
    Q <- ∅                          // min-priority queue
    INSERT(Q, (Vi, dist(Vi)))

    while Q ≠ ∅
        v <- EXTRACT-MIN(Q) // Minimum is the one with lowest gScore

        if equals(v, Vf):
            return buildPath(G, Vi, Vf)

        for each w in Adj(v):
            if dist(w) > dist(v) + weight(v, w) then
                dist(w) <- dist(v) + weight(v, w)
                gScore(w) <- dist(w) + Heuristic(v, Vf)
                path(w) <- v

                if w not in Q then
                    INSERT(Q, (w, dist(w)))
                else
                    DECREASE-KEY(Q, (w, dist(w)))
```

```
buildPath(G, Vi, Vf):
    path <- {}          // Vetor retornado com o caminho encontrado
    w <- Vf

    while !equals(w, Vi) do
        pushFront(path, w)
        w <- path

    pushFront(path, Vi)

    return path
```

O algoritmo é em tudo semelhante ao algoritmo de **Dijkstra**, com exceção da forma como os vértices são ordenados na *priority queue*. Desta forma, os vértices mais perto do vértice de destino têm maior prioridade face aos vértices que estão mais distantes. Contrariamente ao algoritmo de **Dijkstra**, o algoritmo **A***, por ser um algoritmo heurístico, não garante sempre uma solução ótima, visto que o maior propósito deste algoritmo é acelerar o processo de cálculo, produzindo um resultado o mais otimizadamente possível.

Como mencionado anteriormente, pelo facto de este algoritmo ser em tudo idêntico ao algoritmo de Dijkstra, a sua complexidade será também ela igual. Portanto, a **complexidade temporal** é **O((|V|+|E|) * log |V|)** e a sua complexidade espacial é **O(|V|)**. No entanto, apesar de os algoritmos serem iguais em termos de complexidade, devido à heurística de aproximação do algoritmo **A***, espera-se que este alcance um melhor desempenho, no caso médio, do que o algoritmo **Dijkstra**.

Pesquisa Bidirecional com algoritmo de Dijkstra

A **pesquisa Bidirecional**, destina-se à resolução do problema de encontrar o caminho mais curto entre dois vértices num grafo. Trata-se de um processamento paralelo (*multithreading*) onde é feita uma pesquisa desde o vértice de origem até ao vértice de destino e vice-versa, tendo o grafo de estar invertido na segunda parte. A pesquisa fica completa quando ambas as pesquisas convergem. Este algoritmo requer mais memória, porém isso compensa pelo facto de termos uma pesquisa mais rápida. No entanto, o tempo de pré-processamento dedicado à inversão do grafo pode "deitar por terra" essa vantagem. De qualquer forma, o ganho de velocidade em um processamento *multithreading* comparativamente com um *single-threaded* é substancial. Por esta razão, presume-se que a *performance* deste algoritmo ultrapasse a do algoritmo de Dijkstra e A*.

Pseudocódigo deste algoritmo:

```
1  bidirectionalSearch(s, t): // s - Origin vertex | t - Destination vertex
2
3      Vertex intersectNode <- NULL
4
5      // Initialization
6      for each v in V
7          visited(v) <- False
8          path <- NULL
9
10
11     ForwardQueue <- ∅ // Forward queue that will hold vertices
12     INSERT(ForwardQueue, s)
13     visited(s) <- True
14
15     BackwardQueue <- ∅ // Backward queue that will hold vertices
16     INSERT(BackwardQueue, t)
17     visited(t) <- True
18
19     while ForwardQueue != ∅ AND BackWardQueue != ∅
20
21         // Search to Use: Dijkstra, A*, etc...
22
23         forwardSearch(ForwardQueue)
24         BackwardSearch(BackWardQueue)
25
26         // Evaluate if both searches have converged
27         intersectNode <- forwardIntersectingBackward()
28
29         if(intersectNode != NULL)
30             return SUCESS
31
32     return ERROR
```

Pesquisa Bidirecional com algoritmo A*

Uma vez que o Algoritmo A* é uma variação do Algoritmo de Dijkstra, o A* Bidirecional segue a mesma lógica do Dijkstra Bidirecional. Faz-se uma pesquisa desde a origem até ao destino (*forward*) e do destino até à origem (*backward*). Ambas as pesquisas tende a chegar a um ponto de convergência. Em seguida, o trajeto processado é o relativo ao *forward* + *backward*.

O objetivo na utilização deste algoritmo é trazer a melhor performance do A* para este método, embora a possibilidade de a solução encontrada não ser a melhor possível, aumente. No entanto, a relação entre a velocidade e a exatidão deste algoritmo faz com que seja um dos melhores no que diz respeito à procura da solução ótima.

Num artigo publicado na organização ACM SIGSPATIAL, respeitante à procura de caminhos em áreas restritas, alguns dos algoritmos mencionados foram também abordados. Os testes realizados consistiram na computação de 100 caminhos mais curtos, avaliando dois casos diferentes. O primeiro consiste em utilizar o valor de uma aresta, como a distância (caminho mais curto) e o segundo consiste em utilizar o tempo que leva a percorrer o trajeto (caminho mais rápido).

Os resultados foram os seguintes:

Method	execution time [μs]	maximum error [%]	average error [%]
Dijkstra	8747	0.0	0.0
Bi-Dijkstra	7327	0.0	0.0
A*	3478	25.3	9.4
Bi-A*-Dij	3462	24.3	8.3
Bi-A*	2797	28.3	10.1

Tabela 1: Resultado com pesos das arestas iguais às distâncias

Method	execution time [μs]	maximum error [%]	average error [%]
Dijkstra	8961	0.0	0.0
Bi-Dijkstra	6800	0.0	0.0
A*	2995	40.0	19.8
Bi-A*-Dij	3017	39.4	18.5
Bi-A*	2386	40.0	22.2

Tabela 2: Resultado com pesos das arestas iguais ao tempo

Em ambos os testes, o tempo de execução encontra-se em microssegundos. A conclusão retirada desta comparação vai de encontro ao que foi dito anteriormente, ou seja: a **pesquisa bidirecional com o algoritmo A*** revelou-se a mais rápida, no entanto a que é suscetível a mais erros, no que diz respeito à procura da solução ótima. Na outra face da moeda, temos o **algoritmo unidirecional de Dijkstra** que é o mais lento, mas em contrapartida tem uma taxa de erro igual a zero, o que implica que nos garante sempre uma solução ótima.

Casos de Utilização

Os casos de utilização iniciam pela importação do grafo a partir do ficheiro de texto com os nós e as arestas respetivas. Posterior visualização, através do *GraphViewer* das rotas adotadas pelos piquetes (entre os diferentes POIs). Importação de cada piquete, considerando as funções dos mesmos, algo que será também feito a partir de um ficheiro de texto. No menu do programa, aparecerá uma opção que servirá para registar um pedido de reparação numa determinada morada. Sobre o mesmo também será requerido qual o tipo de reparação, uma vez que é a partir disso que temos conhecimento da *skill* do piquete que atender a *issue*. Através do registo de cada atividade, será criada a rede de POIs e posteriormente, após a realização de um *clustering* por cada zona (agrupamento de moradas). No fim, serão atribuídas um conjunto de tarefas ao menor número de piquetes possível, utilizando como base o problema do casamento estável possivelmente aliada a uma estratégia *greedy*. Vale a pena sublinhar que ao realizar o referido *clustering* estamos já a dar um grande passo na minimização do caminho total percorrido pelos piquetes. Posteriormente sobre as atividades que cada um terá de fazer, iremos calcular a rota ótima e, com auxílio do *GraphViewer,* mostraremos as mesmas, tal como foi referido no início deste tópico (tendo em atenção que os trajetos deverão tomar cores diferentes por forma a distinguir o trajeto efetuado por cada piquete).

Conclusão Preliminar

O objetivo principal desta formalização é de definir as ferramentas que teremos que usar para resolver o nosso problema. Começamos por definir uma técnica de pré-processamento do grafo inicial, de forma a aumentar a *performance* dos nossos algoritmos. Depois, decidimos dividir o problema em iterações sucessivas, aumentando o nível de dificuldade a cada passo, o que facilitará a sua implementação. Falamos também dos algoritmos que serão utilizados, com ênfase nos mais importantes no contexto do nosso problema. Esta análise fez-nos chegar à conclusão que, embora os algoritmos heurísticos (por exemplo, o A*) não tenham uma solução ótima, são bastante mais eficientes do que os não heurísticos, pelo que decidimos fazer uso destes no nosso projeto. Além disso, em relação aos algoritmos A* e Dijkstra, depreendemos que se utilizarmos uma perspetiva bidirecional, ao invés de unidirecional, conseguimos aumentar a *performance* ainda mais, pelo que implementaremos soluções bidirecionais na resolução do nosso problema.

Portanto, toda a pesquisa necessária para a elaboração desta formalização contribuiu para uma melhor compreensão do problema.

Esforço dedicado por cada elemento – Parte I

Pedro Vale - up201806083@fe.up.pt
Tarefas:

- ---

Esforço dedicado: 0%

Rui Pinto - up201806441@fe.up.pt
Tarefas:

- Descrição do tema
- Iterações do problema
- Dados de entrada e saída
- Restrições
- Função objetivo
- Perspetiva de solução
- Algoritmos: pesquisa em profundidade, pesquisa em largura, A*, Pesquisa Bidirecional com algoritmo de Dijkstra
- Casos de utilização

Esforço dedicado: 50%

Tiago Gomes - up201806658@fe.up.pt
Tarefas:

- Descrição do tema
- Iterações do problema
- Dados de entrada e saída
- Restrições
- Função objetivo
- Perspetiva de solução
- Algoritmos: Dijkstra, Pesquisa Bidirecional com algoritmo A*
- Conclusão

Esforço dedicado: 50%

Bibliografia (parte I)

- Slides das aulas teóricas
- Depth-first-search e Breadth-first-search, https://stackoverflow.com/questions/3332947/when-is-it-practical-to-use-depth-first-search-dfs-vs-breadth-first-search-bf
- Algoritmo de Dijkstra, https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra
- Algoritmo A*, https://en.wikipedia.org/wiki/A*_search_algorithm
- Champeaux, Dennis and Sint, Lenie. "An improved bidirectional heuristic search algorithm". Journal of the ACM 24, no. 2(1977), https://dl.acm.org/doi/10.1145/322003.322004
- Champeaux, Dennis. "Bidirectional heuristic search again". Journal of the ACM 30, no. 1(1983), https://dl.acm.org/doi/10.1145/322358.322360

Parte II

Introdução Parte II

No decorrer do desenvolvimento da parte II do projeto, percebemos que teríamos de ultrapassar certas dificuldades, nomeadamente a implementação de novos algoritmos que fossem úteis ao desenvolvimento do projeto e sobretudo a utilização de mapas onde os mesmos pudessem correr "livremente", sem qualquer tipo de limitação. No caso dos mapas baseados em *Grids*, tivemos que implementar arestas bidirecionais de modo a que através de um qualquer ponto X se pudesse chegar a todos os outros.

De uma maneira generalizada, o nosso projeto foi diagnosticado com as seguintes funcionalidades:

- Leitura do grafo a partir de um ficheiro de texto; primeiramente os nós e depois as arestas. Na leitura das arestas, já é computado heuristicamente a distância entre dois nós.
- Leitura das diferentes *tasks* e *pickets*, que serão utilizados no contexto do problema e definição do vértice onde se encontrará a sede da *Fix It.*
- Pré-processamento do grafo, onde é executado um algoritmo (SCC - *Strongly Connected Components*) que permite avaliar as diferentes componentes fortemente conexas que um grafo tem. Caso o mesmo não seja fortemente conexo, dá-se a sua modificação utilizando a maior árvore fortemente conexa presente.
- Divisão do mapa em *clusters* que permitem um acesso mais eficaz aos diferentes POIs (Pontos de Interesse), minimizando as distâncias percorridas pelos piquetes.
- Representação gráfica do mapa, do mapa com *clusters* e dos trajetos dos piquetes, com recurso ao *GraphViewer*.
- Desenvolvimento de um algoritmo *greedy*, que realiza a atribuição das diferentes *tasks* para cada piquete, tido tendo em consideração os horários de trabalho definidos pela empresa.
- Cálculo de rotas ótimas para cada piquete. Não se tem em consideração a ordem pela qual cada task deve ser efetuada. Essa ordem é determinada por nós, da maneira que permita minimizar ao máximo a distância percorrida por cada piquete, ou seja, somos nós que atribuímos os horários às tarefas.

Estruturas de dados Utilizadas

Ao longo do projeto, foram utilizadas diversas classes que recorriam a várias estruturas de dados. Na nossa primeira tentativa de *Clustering* recorremos aos *maps,* no entanto tivemos de abandonar essa ideia por razões que iremos explicar adiante. No cálculo das Componentes Fortemente Conexas, recorremos ao uso de *stacks* de modo a fazer a numeração dos vértices em pós-ordem. Porém na esmagadora maioria do projeto a estrutura de dados utilizada foi o vetor, uma vez que é bastante flexível e adequava-se às nossas necessidades.

Graph, Vertex, Edge

As classes *Graph*, *Vertex* e *Edge*, são baseadas na das aulas práticas e foram adaptadas consoante os algoritmos utilizados. Grande parte do seu funcionamento permaneceu inalterado.

Vertex

```
enum MAP_ZONE {ZONE0 = 0, ZONE1 = 1, ZONE2 = 2, ZONE3 = 3, ZONE4 = 4};


/*
 * ============================================================================================
 * Class Vertex
 * ============================================================================================
 */

template <class T>
class Vertex {

public:

    T info;                     // Vertex Content

    vector<Edge<T> *> outgoing; // Outgoing Edges

    vector<Edge<T> *> incoming; // Incoming Edges

    double weight;              // Weight used in Path Finding Algorithms

    bool visited;               // Checked if Vertex was visited (Path Finding Algorithms)

    bool invVisited;            // Checked if Vertex was visited (Path Finding Bidirectional) in reverse order.

    Vertex<T> *path;            // Previous Vertex for Path Finding.

    int queueIndex = 0;         // Required by MutablePriorityQueue

    long double x;              // X Coordinate

    long double y;              // Y Coordinate

    MAP_ZONE vZone;             // Zone where the Vertex belongs

    double dist = 0;            // For Path Finding Algorithms
```

A classe Vertex armazena informação sobre um nó do grafo. Possui um atributo *template* info, que especifica o seu ID. Possui também coordenadas X e Y que serão posteriormente utilizadas na execução dos *Path Finding Algorithms* e na visualização do grafo utilizando o *GraphViewer*.

Por fim, como foi referido em cima, cada vértice está associado a uma zona. Os vértices pertencentes à *ZONE0* são vértices comuns e os vértices pertencentes às restantes zonas já são efetivamente *tasks* a realizar que, por sua vez foram divididos em zonas.

Picket

```
class Picket {

    static int current_id; // Last inserted ID

    int id;                // Picket ID

    string name;           // Picket Name

    vector<string> roles;  // Skills

    vector<Task*> tasks;   // Assigned Tasks

    int zone;              // Zone where it belongs

    int numTasksDone;      // Number of previous tasks done (experience)

    vector<long> path;     // Path to wander

    Time currentTime;       // Last task Time
```

A classe *picket*, representa um trabalhador da *Fix IT.* Cada trabalhador é identificado unicamente através de um ID, nome, conjunto de *skills*, representado através de um vetor de *strings*, as tarefas que que lhe foram atribuídas, representado através de um vetor de *task**, a zona a que ele pertence, o número de *tasks* previamente efetuadas, que será um critério que atribui uma certa prioridade a um piquete quando está a ser selecionada para a execução das *tasks*. Temos também o vetor *path* que detém o ID dos vértices que o *picket* terá de percorrer no seu "percurso diário" e, por fim o *currentTime* que representa o horário da última tarefa realizada pelo *picket*.

Task

```
class Task {
    static int current_id;      // ID da última task inserida

    int id;                     // Task ID

    string function;            // Função requerida para realizar a task

    Picket* responsiblePicket;  // Picket responsável por fazer a task

    long int nodeId;            // ID do nó correspondente

    int durationMinutes;        // Duração em minutos

    Time beginTime;             // Tempo de início

    MAP_ZONE zone;              // Zona do mapa a que a task pertence
```

A classe *task*, representa uma tarefa que um *picket* da *FIX IT* terá de fazer. É novamente identificada pelo seu ID, pela função requerida, pelo *picket* que a realizou (*Picket*)*, pelo ID do nó com o qual se relaciona, a sua duração em minutos, pelo instante inicial em que foi executada e pela zona em que se encontra.

Company

```
class Company {
    string name;                            // Nome da empresa

    vector<Picket*> pickets;                // Lista Pickets

    vector<Task*> tasks;                    // Lista de tasks

    Graph<long int> cityGraph;              // Grafo utilizado

    Time beginTime;                         // Horário de trabalho - Início

    Time endTime;                           // Horário de trabalho - Fim

    long startVertexId;                     // Nó que representa a empresa

    SEARCH_ALGORITHM searchAlgorithm;       // Algoritmo de procura a ser utilizado

    int maxNumTasks = INT32_MAX;            // Máximo número de tasks

    int maxNumPickets = INT32_MAX;          // Máximo número de Pickets
```

A classe *Company* é o cerne do nosso projeto. É ela que detém o grafo a utilizar, os *pickets* e as *tasks* da *FIX IT*, etc. Possui também o horário de trabalho a ser respeitado pelos *pickets,* o vértice representativo da sede da empresa (pode ser alterado em *runtime*), o máximo de *tasks* a atender, bem como o máximo de *pickets* a trabalhar.

Conectividade do Grafo

A conectividade do grafo é avaliada no início da leitura do mapa. Primeiramente lêem-se os ficheiros relativos aos *nodes* e *edges* e em seguida, procede-se à execução do algoritmo SCC (*Strongly Connected Components)*. O mesmo é dado pelos seguintes passos:

- Pesquisa em profundidade no grafo que determina uma floresta de expansão, numerando os vértices em pós-ordem.
- Inverter todas as arestas do grafo; algo que não foi necessário fazer porque o grafo em si já tem uma lista de *incoming edges* para cada vértice, o que já substitui esse *reverse*.
- Segunda pesquisa em profundidade, começando pelo vértice de numeração mais alta ainda não visitado.
- Cada árvore obtida é um componente fortemente conexo, i.e., a partir de um qualquer nó pode-se chegar a todos os outros.

Após a execução desse mesmo algoritmo é selecionada a maior árvore obtida e procede-se à atualização do grafo. No caso dos mapas fornecidos na Unidade Curricular (Porto, Penafiel e Espinho) a transformação dos mapas não fortemente conexos origina os mapas fortemente conexos para cada cidade, respetivamente. Já nos mapas fortemente conexos, a árvore origina tem o mesmo tamanho do grafo inicial, razão pela qual não é feita nenhuma modificação adicional. Esta análise é feita, porque os algoritmos de *Path Finding* necessitam de atuar sobre regiões fortemente conexas, caso contrário correríamos o risco de referenciar *tasks* em regiões inalcançáveis pelos piquetes.

Numa primeira fase, fazemos uma pesquisa em profundidade normal entre todos os vértices. Em seguida, no que diz respeito à parte recursiva, armazenamos os vértices numa *stack*, de acordo com a já referida pós-ordem. Posteriormente, fazemos uma visita inversa, que é exatamente igual à outra *DFS* referida, com a pequena diferença de que recorremos às *incoming edges* e que retornará todos os vértices alcançáveis a partir do vértice em questão. O resultado virá em *subTree* que terá cada uma das árvores do grafo. No caso de ele ser fortemente conexo será apenas uma com todos os vértices. Por fim, todas essas árvores serão armazenadas no vetor *resultingTree* sendo um vetor bidirecional irá deter todas as componentes fortemente conexas do grafo. Segue o pseudocódigo que retrata o referido algoritmo:

```
calculateSCC(G, V) // G -> Grafo | V -> Vértices

    postOrder <- ∅ // Stack que terá os vértices ordenados em pós-ordem
    for each vertex in V
        visited(vertex) = false;

    for each vertex in V
        if(!visited(vertex))
            dfsWithPostOrder(vertex, postOrder) // Fulfill postOrder

    resultingTree <- ∅
    subTree <- ∅
    // postOrder holds the vertexes in post-order

    for each vertex in V
        visited(vertex) = false;

    while Q ≠ ∅ do
        toVisit = TOP(postOrder)
        POP(postOrder)

        if(!visited(toVisit))
            ReversedfsWithPostOrderVisit(toVisit, subTree) // DFS Search using reversed graph (incoming edges)

        INSERT(resultingTree, subTree)
```

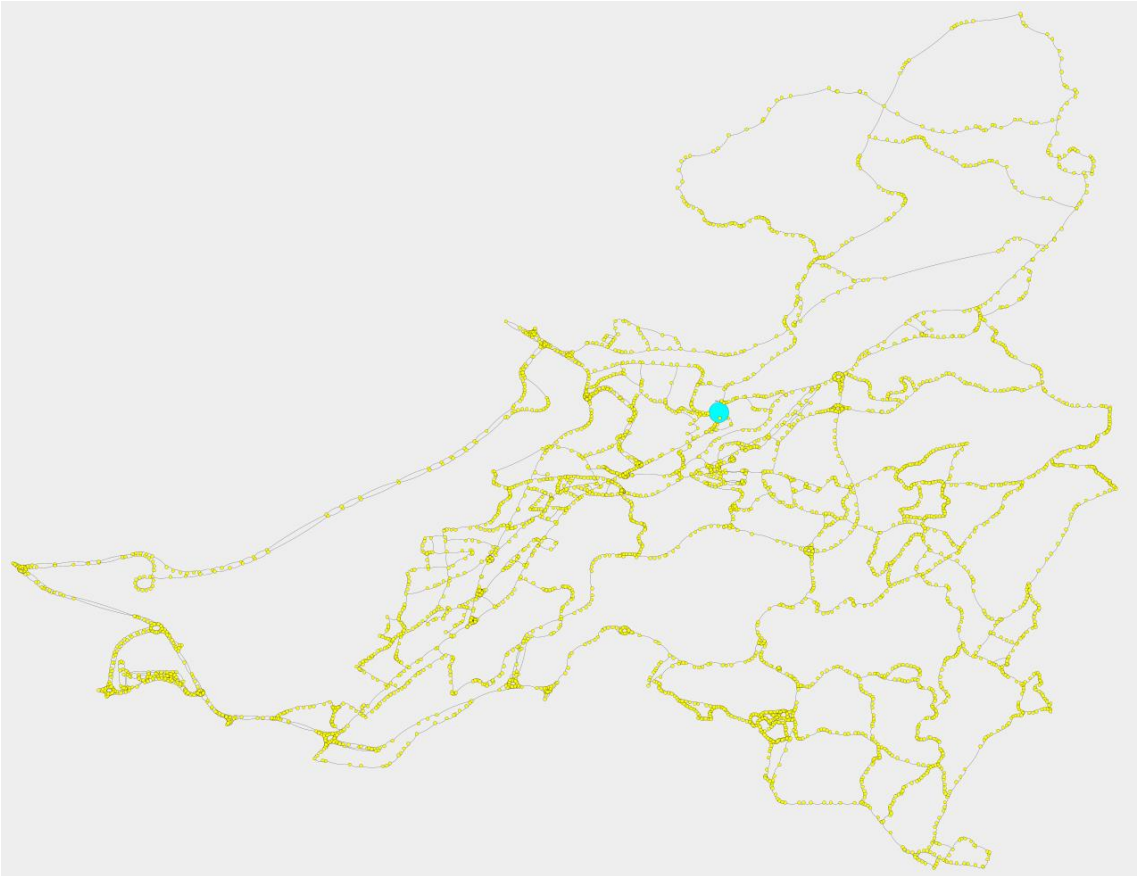
STRONGLY CONECTED COMPONENTS - PSEUDOCÓDIGO

A sua complexidade é a mesma da DFS, ou seja: O(|V| + |E|) (temporal) e O(|V|) (espacial).

Em jeito de demonstração, segue um exemplo com o mapa de Penafiel. Nesta imagem temos o mapa inteiro fornecido:

PENAFIEL (MAPA INTEIRO)

E após aplicar o *SCC*, o resultado produzido para o mesmo mapa, que coincide com o mapa fortemente conexo fornecido, como referido anteriormente.



PENAFIEL (MAPA FORTEMENTE CONEXO)

Algoritmos efetivamente implementados

Dijkstra, A* (A-Star), Depth-First-Search, Breadth-First-Search

Tanto o algoritmo de *Dijkstra*, como o algoritmo A*, como a *DFS* e *BFS,* foram utilizados no projeto. No que diz respeito ao mesmo, na execução do *TSP Nearest Neighbour*, fazemos uso do *Dijkstra* e do A* sendo dada ao utilizador opção para escolher entre os dois. Relativamente ao pseudocódigo de cada um destes algoritmos, o mesmo consta na primeira parte do relatório, nos respetivos tópicos.

Travelling Salesman Problem (TSP)

A classe TSP é responsável por calcular o melhor caminho que começa num certo vértice e termina noutro, passando por um conjunto de pontos de interesse. Assim, depois de efetuado o algoritmo, um objeto da classe TSP é composto por um vetor com os IDs dos vértices que terão de ser percorridos sequencialmente.

A implementação usada utiliza uma heurística de cálculo denominada vizinho mais próximo. Nesta *approach*, são escolhidos sucessivamente os pontos de interesse mais próximos do ponto atual, até que todos foram visitados, começando no vértice inicial. Embora este algoritmo seja bastante eficiente nem sempre consegue retornar o melhor caminho possível; todavia, consegue fazer uma boa aproximação do mesmo. A sua complexidade temporal é, no pior caso, $O((|V|+|E|)*\log|V|^2)$, e a espacial é $O(|V|)$.

O pseudocódigo deste algoritmo é o seguinte (próxima página):

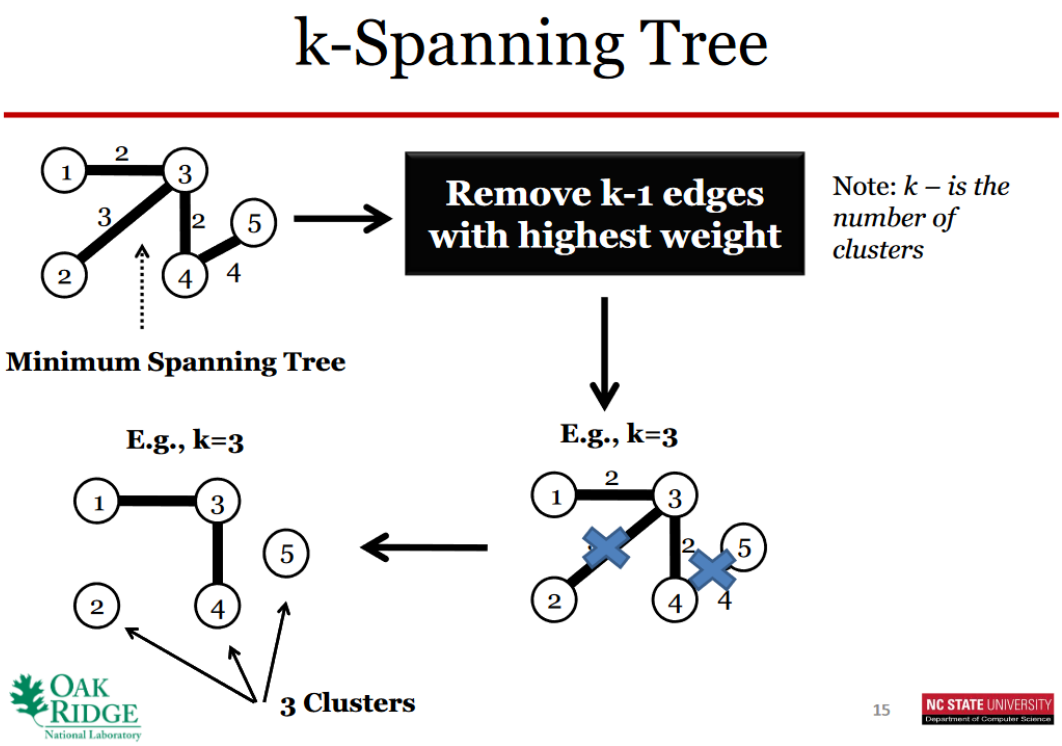
```
// G - Grafo
// startVertex - vértice de origem
// endVertex - vértice de destino
// POIs - conjunto de vértices que temos que percorrer
TSP(G, startVertex, endVertex, POIs):

    // Calcular a sequência de visita dos vértices
    visitOrder <-- {}
    findBestVisitOrder(G, startVertex, POIs, visitOrder)

    // adicionar o vértice de destino à sequência de visita
    append(visitOrder, endVertex)

    // se a sequência de visita contem todos os pontos de interesse, construir caminho
    if contem(visitOrder, POIs):
        buildSolution(G, visitOrder)

    // senão não há solução
    else
        return nil


findBestVisitOrder(G, startVertex, POIs, visitOrder):

    // começamos com o vértice inicial
    insert(visitOrder, startVertex)

    // se não é possível construir caminho a partir do vértice, fazer back-tracking
    if (!canAccess(startVertex, POIs)):
        remove(visitOrder, startVertex)
        return

    // encontrar o ponto de interesse mais próximo do ponto atual
    nextVisit <-- getClosestVertex(G, startVertex, POIs)

    // remover o próximo ponto a visitar dos pontos de interesse
    remove(POIs, nextVisit)

    // chamar a função recursivamente
    findBestVisitOrder(G, nextVisit, POIs, visitOrder)

    // verificar se a recursão teve sucesso
    if isComplete(visitOrder):
        return

    else
        // remover o vértice e fazer back-tracking
        remove(visitOrder, startVertex)
        return

// construir caminho
buildSolution(G, visitOrder):
    solution <-- {}

    // encontrar sequencialmente o melhor caminho entre cada ponto da sequência de visita
    for each vertex in visitOrder do:
        // também poderia ser o A-Star
        append(solution, Dijkstra(vertex, nextVertex)

    return solution
```

No contexto do nosso problema, este algoritmo é usado para calcular o melhor trajeto de cada piquete, que começa e termina na sede da empresa, e que passa por todas as tarefas que lhe foram atribuídas.

```
TSP<long> tsp(&cityGraph, searchAlgorithm);
vector<long> path = tsp.calculatePath(picketTasksIds, startVertexId, startVertexId);
picket->setPath(path);
```

Clustering

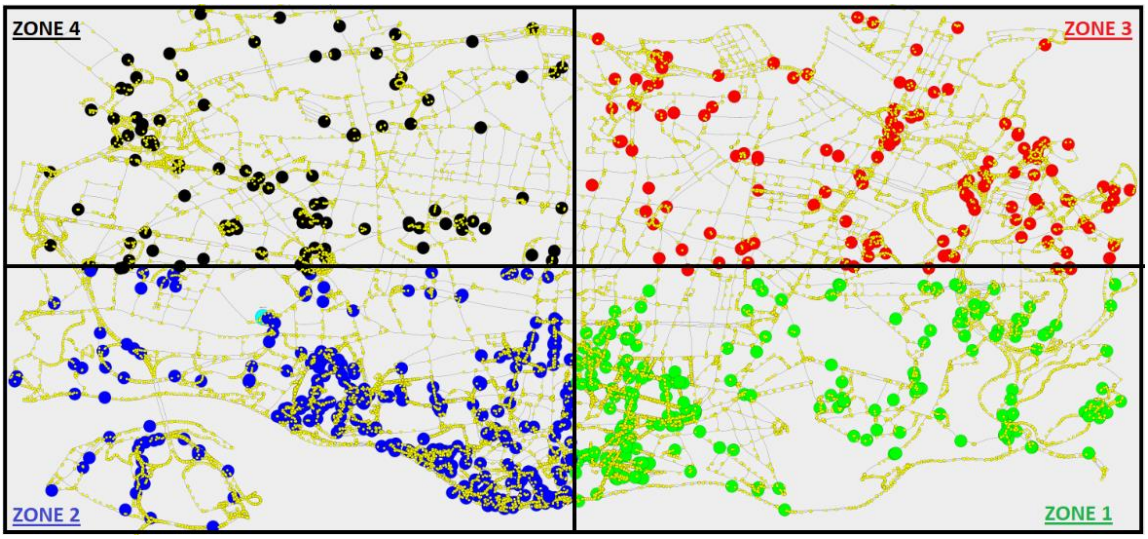
Numa primeira fase, a nossa intenção foi construir um algoritmo de *clustering* baseado em algum já existente. Após alguma pesquisa, deparamo-nos com um algoritmo (referenciado na bibliografia) que fazia uso da *k-spanning Tree* e consistia em: através do algoritmo de *Prim*, construir uma árvore de expansão mínima e caso quiséssemos obter X *clusters*, eliminaríamos X – 1 *edges* começando por aquelas com maior peso. Segue um slide de uma apresentação da *NC State University – Department of Computer Science* que retrata o algoritmo:



Este raciocínio foi adotado numa primeira fase, mas na reta final foi abandonado uma vez que a produção da *minimum spanning tree*, teria de ser feita para todos os POIs, processo esse que se revelou um pouco confuso. A solução adotada tentava para cada POI, encontrar outra POI o mais perto possível, de uma forma heurística. Acontece que esta solução nem sempre se revelou eficaz.

Square Clustering

Uma vez que a tentativa de implementação do algoritmo de *clustering* anterior falhou, a solução adotada é relativamente mais simples mas produz resultados interessantes que tentam minimizar a distância percorrida pelos piquetes. Desta forma é mais lucrativo para a empresa enviar X piquetes para uma determinada zona do mapa, ao invés de ter de enviar o piquete percorrer moradas espalhadas pelo mapa inteiro. Segue então uma imagem que retrata o algoritmo, uma vez que uma imagem explica melhor do que mil palavras.



EXEMPLO DE SQUARE CLUSTERING

Como a imagem indica, é feito um quadrado à volta da região das *tasks*, assinaladas com as cores verde, azul, vermelho e preto e interiormente volta-se a dividir o quadrado em quatro partes. Cada uma delas corresponderá a uma zona diferente, como indicado na figura.

Pairing

A classe *Pairing* é composta por um algoritmo *greedy*, que serve para atribuímos as tarefas aos piquetes de acordo com as suas funções, zonas e disponibilidades. Começamos por filtrar as tarefas de acordo com a sua zona, e por distribuir os piquetes pelas diversas zonas disponíveis de uma forma aleatória. Depois, começamos a atribuir as tarefas aos piquetes, de uma forma *greedy*. Iteramos por cada zona, e para cada tarefa dessa zona, percorremos o vetor de piquetes disponíveis e verificamos se o piquete tem a zona, a função e a disponibilidade compatíveis com tarefa (de notar que o vetor de piquetes está ordenado por prioridades dos mesmos: tem prioridade o piquete que tiver um maior número de *skills*, e entre os que tiverem a mesma quantidade, têm prioridade aqueles que tiverem mais experiência na empresa, ou seja, aqueles que tiverem um maior número de tarefas realizadas). Se o piquete for compatível com a *task*, então esta é adicionada a um vetor de tarefas da classe *Picket*, e o atributo *responsiblePicket* da classe *Task* é preenchido com um apontador para o objeto *Picket* em questão. No caso de no fim de todas as iterações pelas diversas zonas ainda hajam tarefas por fazer e piquetes sem tarefas atribuídas, efetuamos uma nova iteração, desta vez permitindo ao piquete que mude a zona de origem para a zona da *task* não emparelhada. Desta forma, conseguimos ultrapassar os inconvenientes da utilização *clusters* e da atribuição de zonas aleatórias aos piquetes, já que poderiam haver tarefas por fazer e piquetes sem fazer nada só porque estes não tinham a mesma zona. Vale também a pena referir que é verificado qual o número máximo de tarefas que cada piquete pode fazer, de acordo com a hora de início e fim de serviços da empresa.

Todavia, a ordem pela qual o piquete terá que completar as *tasks* não é determinada neste passo do algoritmo, mas sim depois de todos os piquetes já terem definidas quais as suas tarefas. Nesse passo, implementado na função *setBestPathToPickets*, da classe *Company*, tiramos partido do algoritmo TSP para calcular qual é o melhor percurso do piquete de forma a começar e acabar na empresa, e efetuar todas as suas tarefas. Tendo a ordem pela qual as vai efetuar, vamos atribuindo os horários às mesmas, de uma forma sequencial, e deixando um certo tempo entre a realização de cada uma para a sua deslocação do piquete. Devido ao facto de no passo anterior já termos verificado se o piquete conseguia fazer todas as tarefas que lhe foram atribuídas no seu horário de trabalho (incluindo o tempo de deslocação entre as mesmas), neste passo não teremos problemas com tarefas cuja hora de início ou fim ultrapassa o horário de fim de atividades da empresa.

A complexidade temporal deste algoritmo é, no pior caso: T*P + Pt*(|V|+|E|)*log(V2), sendo T o número de tarefas, P o número de piquetes e Pt o número de piquetes com tarefas. A parte Pt*(|V|+|E|)*log(V2) corresponde à utilização do algoritmo TSP para cada piquete com tarefas atribuídas.

De seguida, é mostrado um exemplo de execução do algoritmo, para 10 piquetes e 50 tarefas, em que o horário de trabalho é das 8h:30m até às 17h:30m.

```
Number of tasks attributed to a picket: 41
Number of tasks not attributed to a picket: 9
Number of pickets that needed to work: 9
Number of pickets that did not work: 1
```

Número de tarefas e piquetes

```
================================TASKS WITH PICKETS====

Task with ID = 62 and zone = 2
Function: Canalizador
Begin Time: 11:03
End Time: 12:03
Picket chosen: Mickael Ribas Carregueiro (ID= 2)
----------------------------------------
Task with ID = 19018 and zone = 4
Function: Eletricista
Begin Time: 11:31
End Time: 14:31
Picket chosen: Zezinho (ID= 3)
----------------------------------------
Task with ID = 33250 and zone = 1
Function: Picheleiro
Begin Time: 16:37
End Time: 17:07
Picket chosen: Mickael Ribas Carregueiro (ID= 2)
----------------------------------------
```

Algumas tarefas que foram atribuídas a piquetes

```
=======================TASKS WITHOUT PICKETS==========

Task with ID = 30076 and zone = 1
Function: Eletricista
Duration: 180
----------------------------------------
Task with ID = 12268 and zone = 1
Function: Eletricista
Duration: 180
----------------------------------------
Task with ID = 41837 and zone = 1
Function: Eletricista
Duration: 180
----------------------------------------
Task with ID = 12721 and zone = 2
Function: Eletricista
Duration: 180
----------------------------------------
```

Algumas tarefas que ficaram sem piquetes atribuídos, devido à falta de disponibilidade

```
==========================PICKETS WITH TASKS=======

Picket with id 0 and zone 1
His tasks are:

Function: Canalizador
NodeId: 44128
Duration: 60
Begin Time: 09:32
End Time: 10:32
Responsible Picket: 0


Function: Trolha
NodeId: 25193
Duration: 30
Begin Time: 08:30
End Time: 09:00
Responsible Picket: 0


Function: Trolha
NodeId: 5578
Duration: 30
Begin Time: 09:01
End Time: 09:31
Responsible Picket: 0
```

```
Function: Trolha
NodeId: 47262
Duration: 30
Begin Time: 10:33
End Time: 11:03
Responsible Picket: 0

--------------------------------------
Picket with id 1 and zone 2
His tasks are:

Function: Trolha
NodeId: 10140
Duration: 30
Begin Time: 10:12
End Time: 10:42
Responsible Picket: 1


Function: Trolha
NodeId: 33863
Duration: 30
Begin Time: 09:41
End Time: 10:11
```

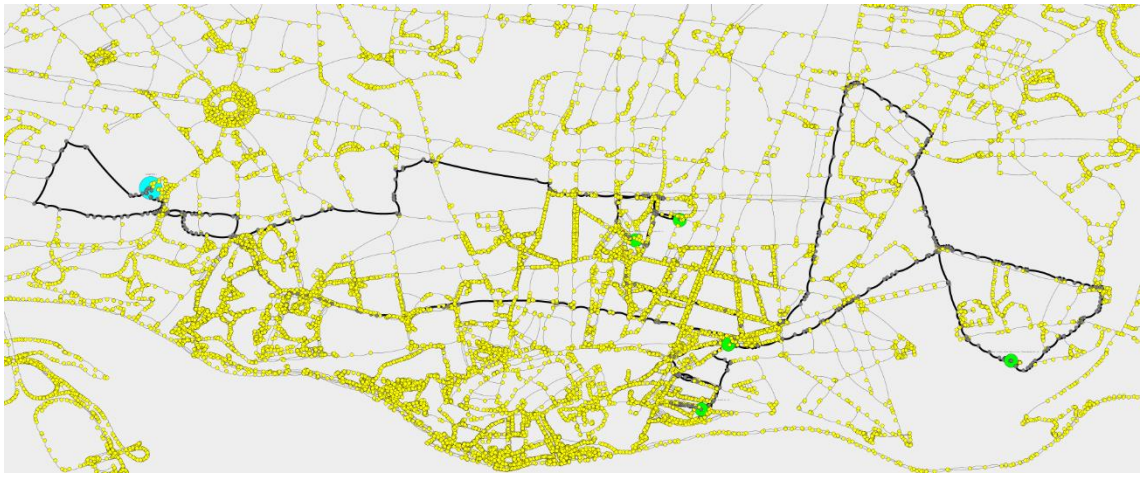
Alguns piquetes com tarefas atribuídas

```
------------------------------------
Picket with id 2 and zone 1
His tasks are:

Function: Canalizador
NodeId: 62
Duration: 60
Begin Time: 11:03
End Time: 12:03
Responsible Picket: 2


Function: Picheleiro
NodeId: 38621
Duration: 120
Begin Time: 14:05
End Time: 16:05
Responsible Picket: 2


Function: Canalizador
NodeId: 909
Duration: 60
Begin Time: 09:01
End Time: 10:01
Responsible Picket: 2
```

```
Function: Picheleiro
NodeId: 911
Duration: 30
Begin Time: 16:06
End Time: 16:36
Responsible Picket: 2


Function: Picheleiro
NodeId: 3070
Duration: 120
Begin Time: 12:04
End Time: 14:04
Responsible Picket: 2


Function: Picheleiro
NodeId: 3964
Duration: 30
Begin Time: 08:30
End Time: 09:00
Responsible Picket: 2
```

```
Function: Canalizador
NodeId: 163
Duration: 60
Begin Time: 10:02
End Time: 11:02
Responsible Picket: 2


Function: Picheleiro
NodeId: 33250
Duration: 30
Begin Time: 16:37
End Time: 17:07
Responsible Picket: 2

------------------------------------
Picket with id 4 and zone 3
His tasks are:

Function: Canalizador
```

Piquete com duas *skills*, a trabalhar praticamente desde a hora de início até à hora de fim



Exemplo de percurso de piquete com 5 tarefas, representadas a verde. A sede da empresa está representada a ciano

Primeiras duas tarefas do piquete representado na imagem anterior

Tarefas da imagem anterior, com mais detalhe

Exemplo de piquete com 17 tarefas

Exemplo de piquete com 9 tarefas, numa diferente zona
das imagens anteriores

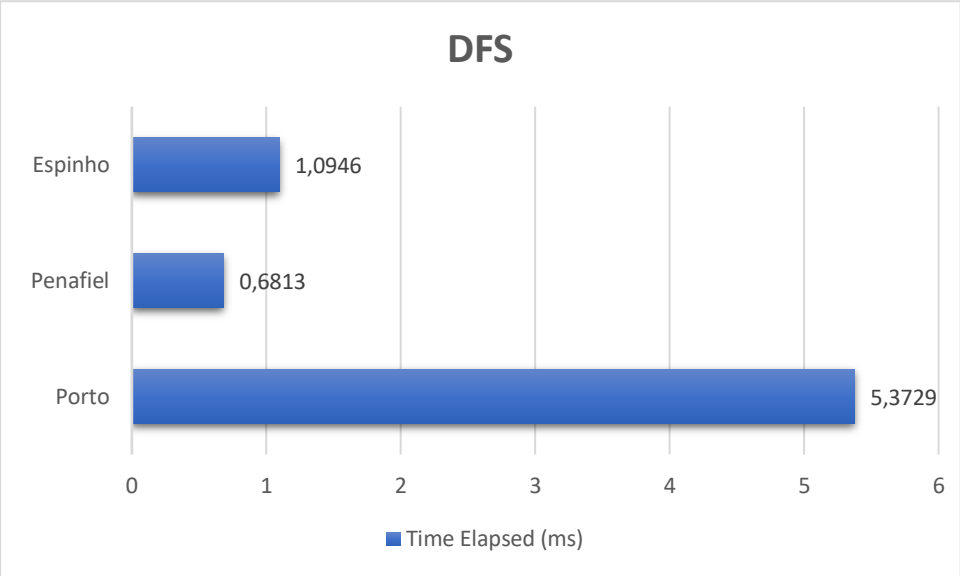
Análise temporal empírica dos Algoritmos

Para os vários algoritmos implementados, foi realizado uma medição do tempo que os mesmo levavam a executar. Para esta avaliação, foram utilizados novamente os três mapas referidos nas aulas teóricas (Porto, Penafiel e Espinho). As medições destes intervalos foram feitas com o auxílio da biblioteca *chrono* de *C++*.

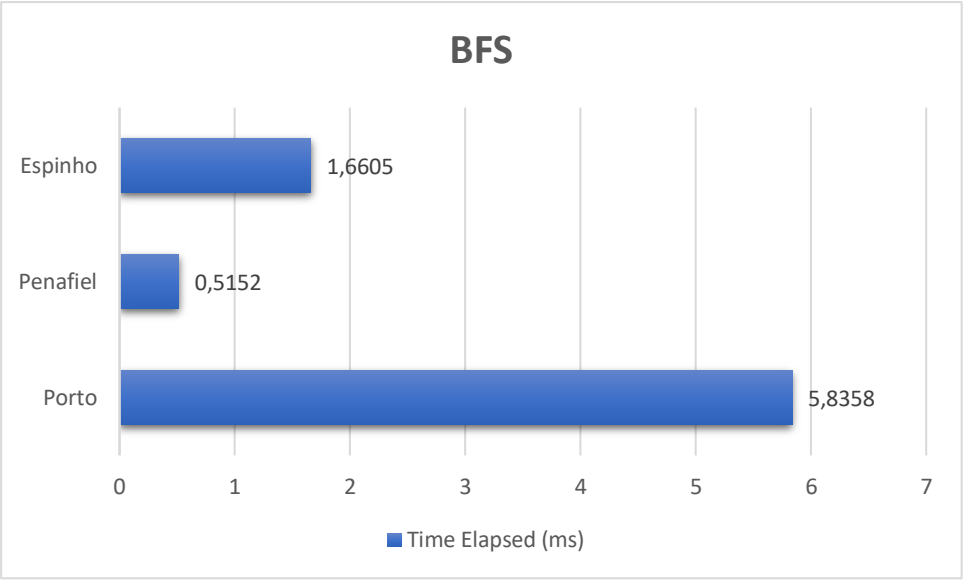
Cidade	**V**	**E**
Porto	26098	29488
Penafiel	3964	4237
Espinho	7108	7938

CARACTERÍSTICAS DOS MAPAS UTILIZADOS

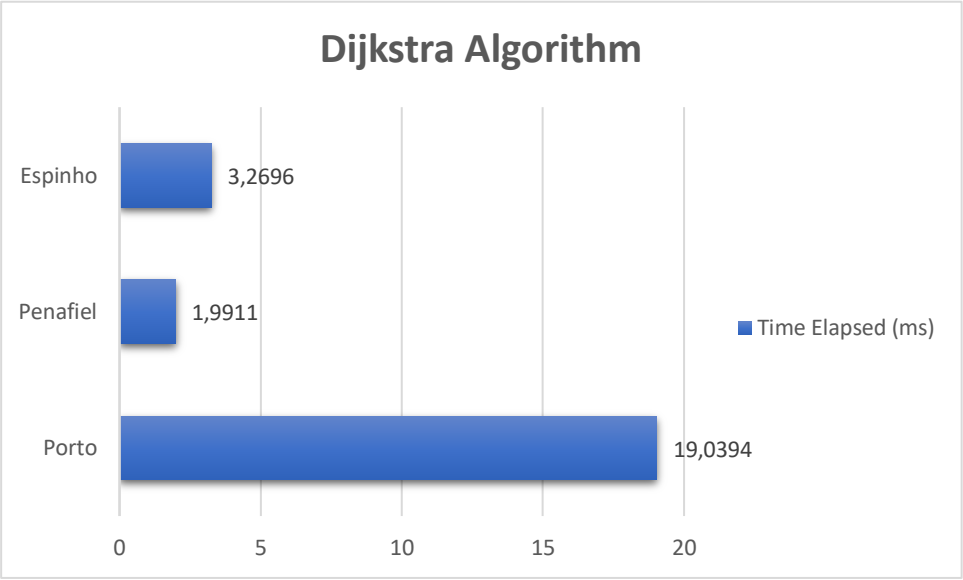
Pesquisa em Profundidade (DFS)



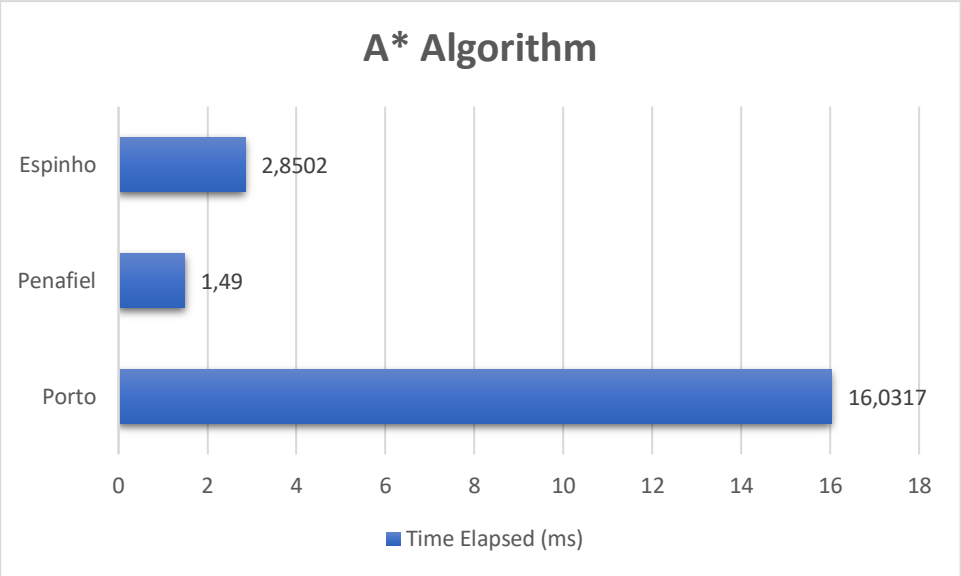
Pesquisa em Largura (BFS)



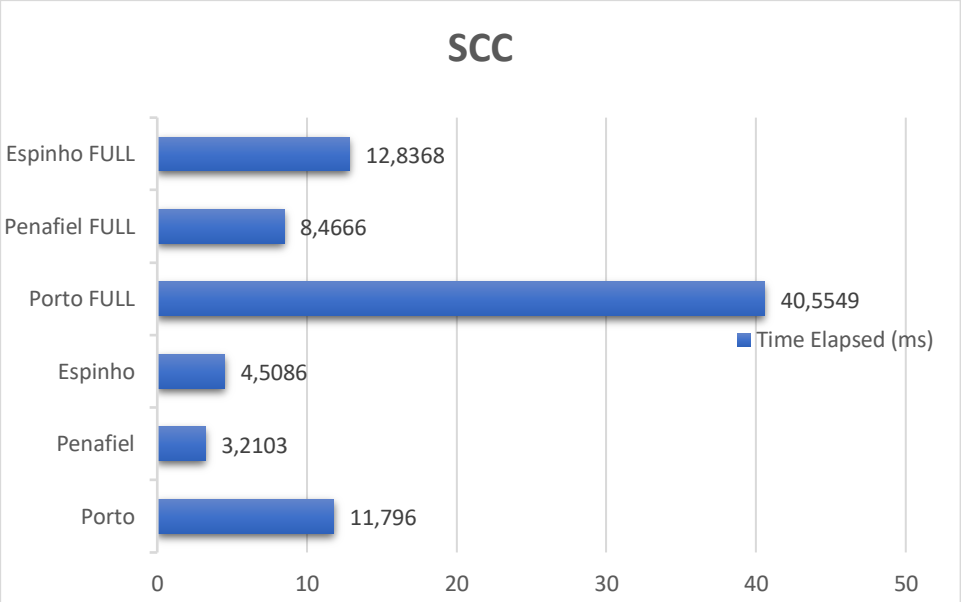
Algoritmo de Dijkstra



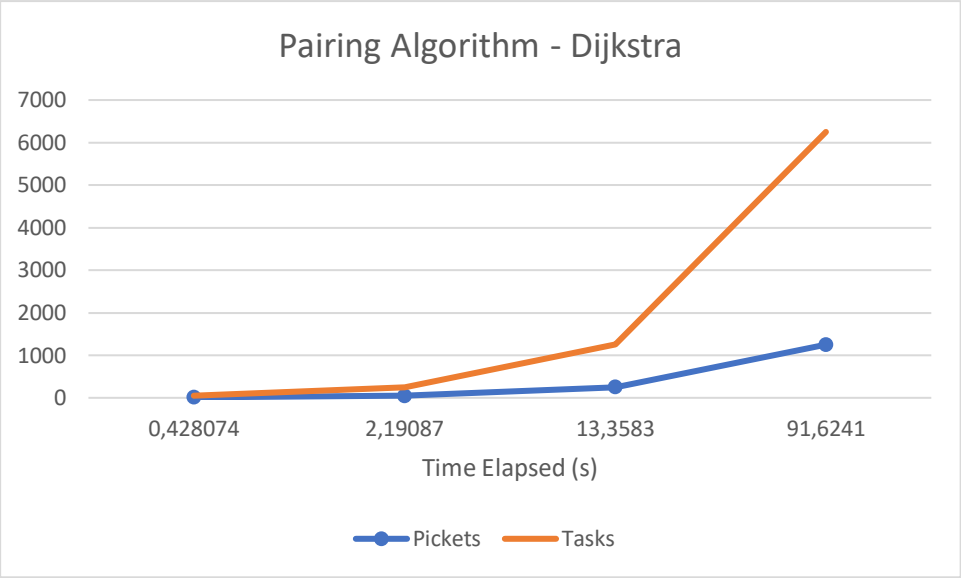
Algoritmo AStar (A*)



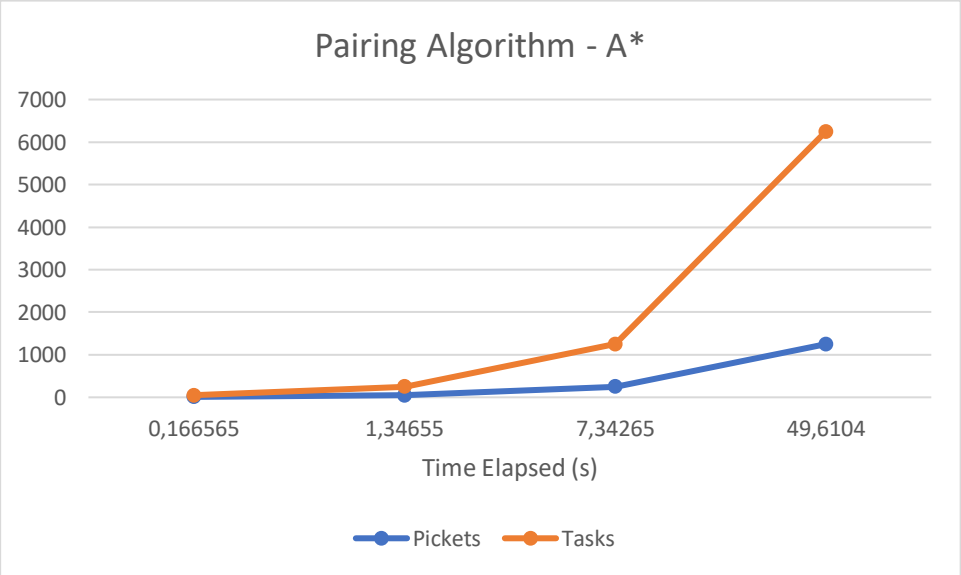
Strongly Connected Components



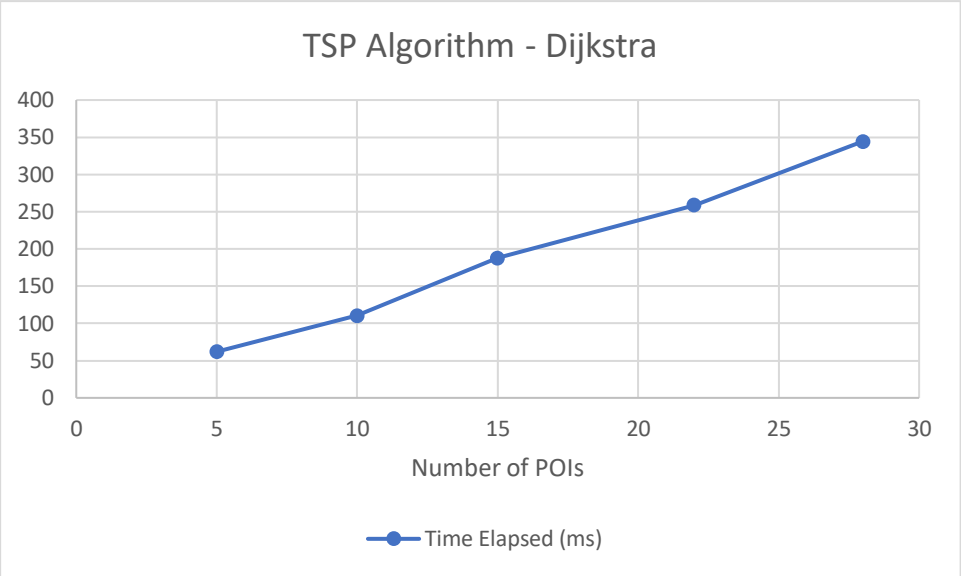
Pairing Algorithm – Dijkstra



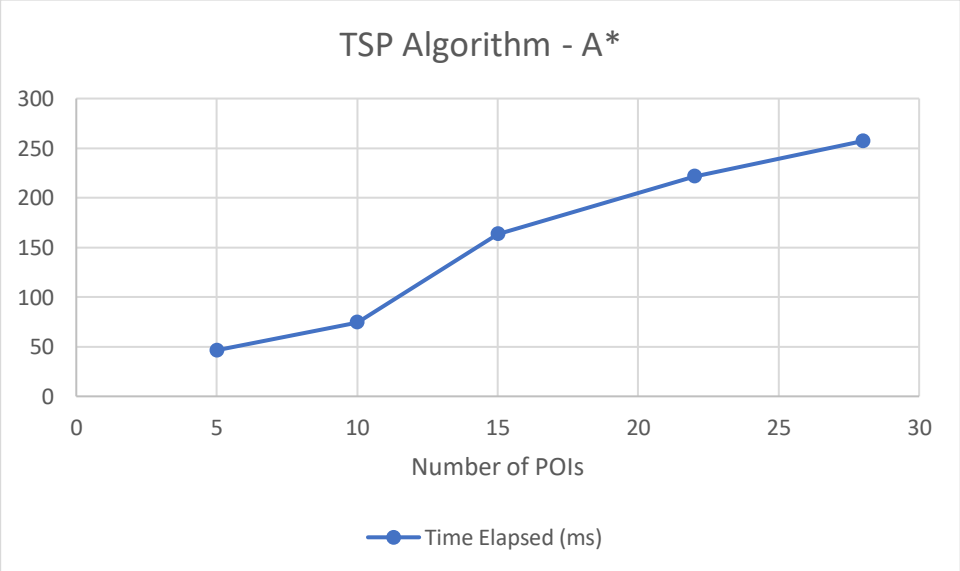
Pairing Algorithm – AStar (A)*



TSP Algorithm - Dijkstra

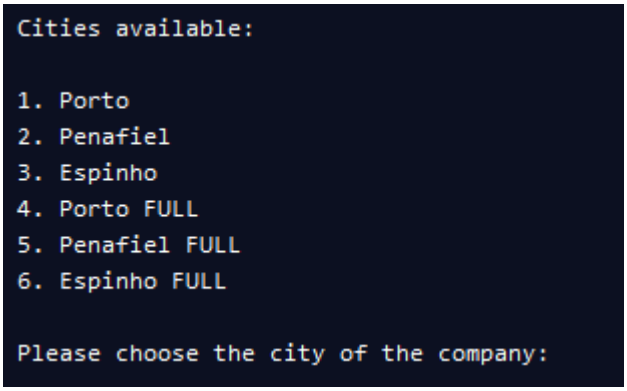


TSP Algorithm – AStar (A)*



Principais Casos de Uso Implementados

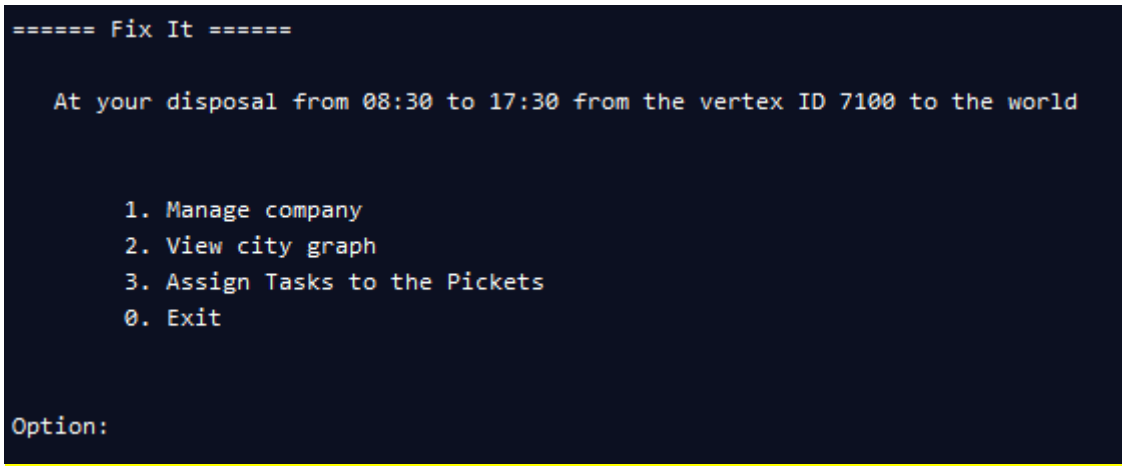
Passamos agora então às funcionalidades e casos de utilização do programa. Na imagem seguinte, podemos observar o "pré-menu" do programa.



PRÉ-MENU

É dada ao utilizador a opção de carregar cada um destes diferentes mapas. Os primeiros três indicam os mapas fortemente conexos, enquanto que os últimos 3 indicam a versão desconexa dos mesmos.

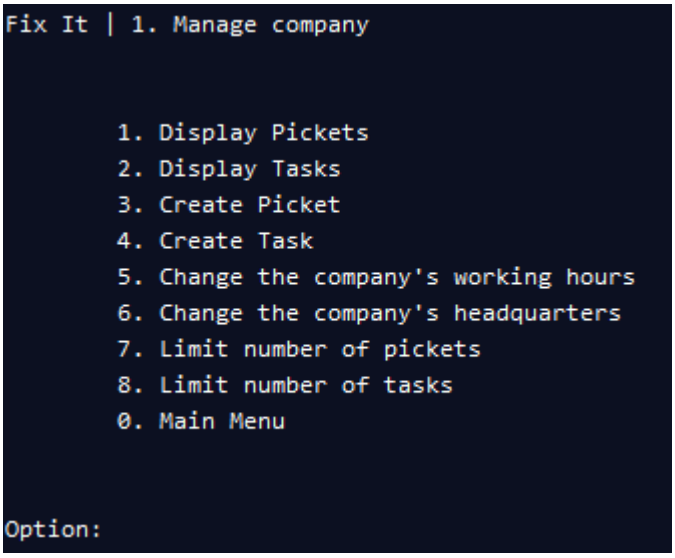
Seguidamente, após a leitura do mapa selecionado e posterior transformação de modo a torná-lo numa versão fortemente conexa, avançamos para o "menu principal". Aqui encontramos diferentes opções, retratadas na seguinte imagem.



MENU PRINCIPAL

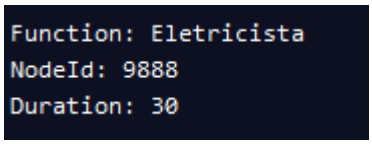
No mesmo consta informação sobre o nome da empresa, o horário sobre o qual os trabalhadores podem efetuar as diferentes tarefas e o ID do vértice que detém a sede da empresa *FIX IT*.

Na **primeira opção do Menu** ("*Manage Company*"), podemos executar ações sobre a *FIX IT*, como retrata a imagem a seguir.

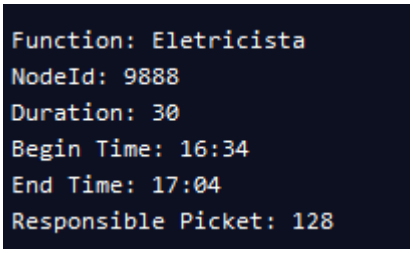


MANAGE COMPANY

Neste submenu temos a opção de listar os diferentes piquetes e *tasks*. Numa fase inicial em que nenhum piquete está atribuído a nenhuma *task*, a informação das mesmas não revela horas de início e fim de execução. Uma vez executado o algoritmo de *pairing* mencionado, no display das *tasks* passará a constar informação relativa à hora de início, fim e o *picket* responsável pela mesma. A título de exemplo, seguem duas imagens referentes ao antes e depois da execução do algoritmo de *pairing*.



ANTES DA EXECUÇÃO DO ALGORITMO DE PAIRING



APÓS A EXECUÇÃO DO ALGORITMO DE PAIRING

Temos também opções que criam tanto novas *tasks* como novos piquetes, alteração do horário de trabalho dos piquetes, alteração da localização da sede da empresa *FIX IT,* bem como limitação do número de piquetes e de *tasks*.

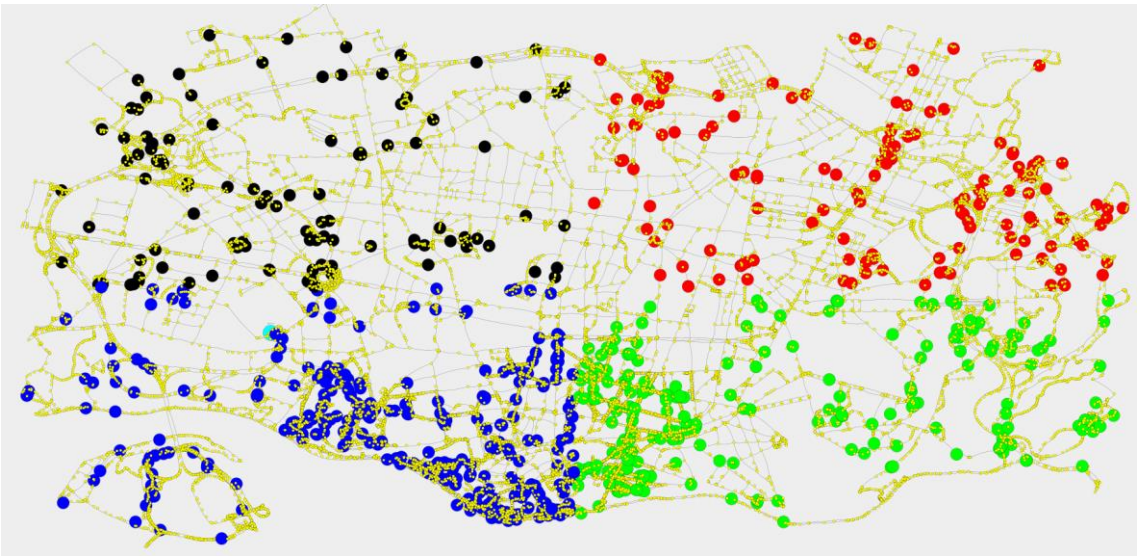
Na **segunda opção do Menu,** procede-se à visualização do grafo com recurso ao *GraphViewer*. São dadas as seguintes opções:

```
Fix It | 2. View city graph


        1. Display Graph
        2. Display & Generate Clusters' Tasks
        0. Main Menu


Option:
```

A primeira opção é responsável pela visualização total do grafo, sem fazer qualquer referência às *tasks* ou às zonas; apenas à sede da *FIX IT*. No caso do mapa do Porto, o resultado obtido é o seguinte:

MAPA DA CIDADE DO PORTO

A segunda opção é responsável pela visualização do grafo considerando a divisão das *tasks* por zonas, como mencionado no *Square Clustering*.



MAPA DA CIDADE DO PORTO ORGANIZADO POR CLUSTERS

Vale a pena sublinhar que não é feita qualquer compressão das coordenadas do grafo, caso contrário seria incrivelmente difícil de visualizar os trajetos percorridos pelos piquetes. No entanto as funções que seriam responsáveis por essa modificação encontram-se implementadas.

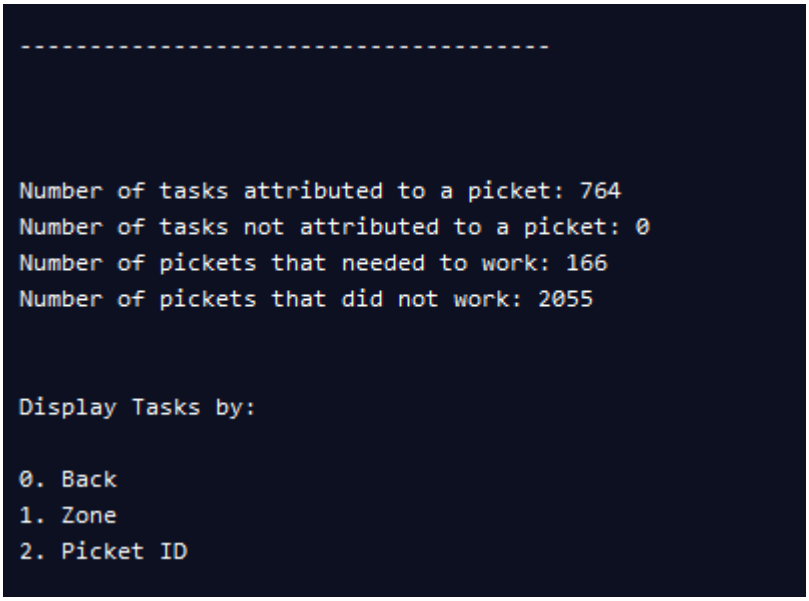
Por fim, na **terceira opção do menu** executa-se o algoritmo de *pairing* previamente enunciado. Aqui é dada a opção de execução do algoritmo recorrendo ao algoritmo de *Dijkstra* ou ao A*, como demonstrado na imagem seguinte:

```
Fix It | 3. Assign Tasks to the Pickets


        1. Assign Tasks to the Pickets
        0. Main Menu


Option:1
 Please choose a search algorithm to use in TSP algorithm:

0. Dijkstra
1. AStar

Option (0 / 1):
```

ESCOLHA DO ALGORITMO A EXECUTAR

Seguidamente é dado o display das *tasks* que às quais foi atribuído um piquete, bem como as *tasks* de cada piquete. Algo já demonstrado no tópico de *pairing*.

Posteriormente é apresentada informação relativamente à execução do *pairing* e, por fim é dada ao utilizador a opção de dar *display* dos caminhos de um piquete em específico ou de *pickets* relacionados a uma determinada zona e com mais de X *tasks* para atender.



OPÇÕES DE VISUALIZAÇÃO DOS CAMINHOS DOS PIQUETES

A visualização dos trajetos dos piquetes foi já abordada no tópico do *Pairing*, com imagens demonstrativas dos mesmos.

Conclusão

Ao longo do desenvolvimento do projeto, deparamo-nos com várias dificuldades. Contactamos por diversas vezes os monitores presentes e conseguimos chegar a soluções que ultrapassaram esses problemas. Aplicamos os diversos algoritmos lecionados nas aulas teóricas, alguns deles não referidos nas práticas (*SCC*, por exemplo), realizamos pesquisas de modo a encontrar algoritmos melhores e mais eficazes (*TSP Nearest Neighbour*) para resolver o problema das rotas ótimas dos piquetes e utilizamos soluções heurísticas de modo a reduzir o processamento utilizado.

Na altura da primeira entrega, estávamos ainda muito "verdinhos", como se costuma dizer. Ao longo do desenvolvimento do projeto fomos percebendo a complexidade do problema, algo que testou a nossa capacidade de adaptação às dificuldades que enfrentávamos.

Em jeito de conclusão, estamos de acordo em dizer que os objetivos propostos pelo projeto foram de facto alcançados, contribuindo sobretudo para um maior desenvolvimento individual e coletivo.

Esforço dedicado por cada elemento – PARTE II

Pedro Vale - up201806083@fe.up.pt
Tarefas:

- Auxílio na realização das restantes tarefas

Esforço dedicado: 20%

Rui Pinto - up201806441@fe.up.pt
Tarefas:

- Implementação da interface gráfica (*GraphViewer)*;
- Estruturação das classes *Graph*, *Vertex* e *Edge*;
- Desenvolvimento dos algoritmos: *Dijkstra, A**, *Clustering, Square Clustering*, *Strongly Connected Components (SCC)*, *Search Algorithms* (*DFS* e *BFS*);
- Relatório.

Esforço dedicado: 40%

Tiago Gomes - up201806658@fe.up.pt
Tarefas:

- Implementação do Menu, das classes referentes às *Taks*, *Picket*, *Company e Time*;
- Leitura de todos os ficheiros (*Nodes*, *Edges*, *Tasks*, *Pickets*);
- Elaboração dos algoritmos: *Pairing* e *TSP Nearest Neighbour*;
- Relatório.

Esforço dedicado: 40%

Bibliografia (parte II)

- Slides das aulas teóricas
- Depth-first-search e Breadth-first-search, https://stackoverflow.com/questions/3332947/when-is-it-practical-to-use-depth-first-search-dfs-vs-breadth-first-search-bf
- Algoritmo de Dijkstra, https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra
- Algoritmo A*, https://en.wikipedia.org/wiki/A*_search_algorithm
- Champeaux, Dennis and Sint, Lenie. "An improved bidirectional heuristic search algorithm". Journal of the ACM 24, no. 2(1977), https://dl.acm.org/doi/10.1145/322003.322004
- Champeaux, Dennis. "Bidirectional heuristic search again". Journal of the ACM 30, no. 1(1983), https://dl.acm.org/doi/10.1145/322358.322360
- Clustering, https://www.csc2.ncsu.edu/faculty/nfsamato/practical-graph-mining-with-R/slides/pdf/Graph_Cluster_Analysis.pdf
- TSP, https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- TSP, https://en.wikipedia.org/wiki/Travelling_salesman_problem
- TSP,https://www.researchgate.net/publication/289195926_On_the_Nearest_Neighbor_Algorithms_for_the_Traveling_Salesman_Problem
- Chrono, https://en.cppreference.com/w/cpp/chrono/steady_clock/now