U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

ExploreIT: Caminhadas e Patuscadas

Conceção e Análise de Algoritmos
25 de maio de 2020

GRUPO 5 – TURMA 3

Caio Nogueira – up201806218@fe.up.pt

Diogo Almeida – up201806630@fe.up.pt

Miguel Silva – up201806388@fe.up.pt

Índice

Descrição do Tema

No âmbito da sua política de fortalecimento do espírito de equipa, uma empresa decidiu oferecer aos seus trabalhadores um fim de semana numa região rural, cheia de locais de interesse ligados por trilhos para a realização de caminhadas.

Os trilhos podem ter diferentes graus de dificuldade, sendo por vezes desaconselhados a pessoas pouco habituadas a estas andanças. Dada a dimensão do grupo, as pessoas poderiam, ao longo do dia, optar por efetuar diferentes conjuntos de trilhos, com a restrição de haver locais de encontro de todo o grupo, nomeadamente para o almoço (num restaurante rústico da zona, numa aldeia remota) e para o final do dia.

Neste projeto, pretende-se implementar uma aplicação que, dado um conjunto de pontos de interesse, locais de confluência e trilhos, produza circuitos que possam ser percorridos pelos trabalhadores. Cada trilho tem um grau de dificuldade e uma duração estimada, função da distância do mesmo.

A dificuldade de alguns dos trilhos pode fazer com que certos pontos de interesse se tornem inacessíveis (em tempo útil), inviabilizando a sua utilização nos circuitos gerados. Desta forma, é necessário ter em atenção os pontos de interesse com pouca acessibilidade, para que possam ser identificados.

Esta aplicação faz também uso de mapas reais, extraídos do OpenStreetMaps (www.openstreetmap.org) e coordenadas geográficas dos pontos de interesse.

Identificação e Formalização do Problema

O problema anterior pode ser representado por um **grafo dirigido pesado**, em que os **vértices** representam os locais da região, as **arestas** representam os vários trilhos que conectam estes locais e os **pesos das arestas** caracterizam a distância e o declive de cada trilho.

Dados de Entrada

- **G (V, E)** - Grafo dirigido pesado contendo o mapa dos trilhos em questão. Este contém as seguintes variáveis:
    - **V** - Conjunto de vértices do grafo, sendo que cada vértice representa local. Cada vértice possui:
        - **Id** - Identificador do vértice;
        - **Type** - NULL se não for um ponto de interesse/local de encontro;
        - **Adj ∈ E** - Lista com todas as arestas adjacentes a V.
    - **E** - Arestas que representam o trilho entre dois locais. Estas contêm:
        - **Id** - Identificador da aresta;
        - **Weight** - Função da distância e declive entre os pontos de interesse (usada para calcular a dificuldade/duração do trilho);
        - **Dest ∈ V** - Vértice de destino.
- **DurMax** – Duração máxima que um trilho pode ter.
- **DifMax** – Dificuldade máxima que um trilho pode ter.
- **Size** – Tamanho total do grupo.

Dados de Saída

- **DurTotal** – Duração total do circuito, calculada através dos valores de distância entre arestas (peso) deste.
- **B** – Vértice que representa o ponto de encontro para o almoço.
- **F** – Vértice que representa o ponto de encontro ao final do dia
- **P** – Sequências ordenadas de vértices que representam os circuitos constituído pelos vários trilhos entre o ponto inicial e o ponto final (F) que podem ser percorridos pelos trabalhadores durante o dia, considerando a dificuldade e a duração dos mesmos.

Restrições

Os dados acima especificados, quer de entrada, quer de saída, denotam o seguinte conjunto de restrições:

- **Size > 1.**
- **Para todos os vértices:**
    - **Type(V[i]) = "meeting_point" v "point_of_interest"**
- **Para todas as arestas**:
    - **Weight(E[i]) > 0**, uma vez que este representa a distância de um trilho. Por consequência, temos também **DurTotal > 0**, uma vez que esta é calculada através do peso total de todas as arestas.
- **$F \in V \wedge F = P_f \wedge Type(P_f) =$ "meeting_point"**, o local final tem de ser o último vértice da sequência ordenada que representa o circuito. Além disso, tem também de ser um ponto de encontro.
- **$B \in V \wedge Type(B) =$ "meeting_point"**, o local para o almoço dos trabalhadores tem de existir e tem de ser um ponto de encontro.

Função Objetivo

A solução ótima para este problema é obtida ao garantir que todos os trabalhadores se encontrem nos pontos de encontro previstos e que lhes seja atribuído um circuito que respeite as suas preferências (dificuldade) e horário estabelecido. Desta maneira, a função objetivo deve retornar vários circuitos com a dificuldade, distância e pontos de encontro aceitáveis.

Perspetiva de Solução

A descrição da solução para este problema é abordada através da exposição das várias técnicas de conceção e algoritmos utilizados.

Técnicas de Conceção/Implementação

Pré-processamento do grafo

Numa primeira abordagem ao tema, verificamos se existe pelo menos um ponto de encontro no grafo dado, bem como se é possível percorrer a trilha mais curta (entre a entrada da região rural e a ponto de encontro para o almoço, ambos aleatoriamente gerados).

Encontrar o trilho a partir de vários pontos iniciais

Após a verificação anterior, pretendemos neste ponto encontrar o caminho mais curto entre o ponto inicial do grafo, gerado aleatoriamente, e o ponto de encontro para o almoço, usando para tal o algoritmo de Dijkstra. Deste modo conseguimos ainda eliminar os pontos que não se encontram a uma distância entre 4000-5000 m e que têm acessibilidade reduzida. Assim, eliminamos pontos que não se encontrem a uma distância razoável para a hora de almoço (assumindo que os trabalhadores percorrem o circuito a uma velocidade de 1 km/h).

Encontrar o caminho entre o ponto de encontro para o almoço e o ponto de encontro final

Neste ponto, dependendo da escolha do utilizador, os trabalhadores poderão percorrer o resto do circuito em conjunto ou novamente em trilhos separados.

Para a primeira opção, é utilizado novamente o algoritmo de Dijkstra entre o ponto de encontro do almoço e vários pontos de encontro finais possíveis desde que se encontrem a uma distância que permita aos trabalhadores cumprir com o horário estipulado anteriormente.

Para a segunda opção é gerado aleatoriamente um ponto de encontro final, sendo usado Dijkstra para verificar se este ponto se encontra a uma distância que permite finalizar o trilho dentro do horário estipulado pelos trabalhores, caso esta condição não se verifique, é gerado novamente um novo ponto de encontro final e assim sucessivamente.

Após termos provado a possibilidade de finalizar o trilho desde o ponto de almoço até ao ponto de encontro final, interessa-nos agora obter vários trilhos possíveis, um para cada trabalhador. Assim, são criados pontos intermédios para cada trabalhador sendo utilizado o algoritmo de Dijkstra entre o ponto do almoço até ao ponto intermédio e novamente entre o ponto intermédio até ao ponto de encontro final gerado, por fim, somam-se as distâncias dos dois caminhos e verifica-se se a passagem pelo trilho criado é adequada para a o tempo restante.

Distribuição dos caminhos gerados pelos vários trabalhadores

Por fim, através do uso de várias *queues* (filas) de *paths*, uma para cada dificuldade, estes circuitos são distribuidos pelos trabalhadores consoante a sua destreza e *tour level*. A filtragem de caminhos não feita inicialmente, é realizada neste ponto do processamento.

Análise dos Algoritmos

Pesquisa em largura

O algoritmo de pesquisa em largura é um dos algoritmos mais simples, sendo a base de outros algoritmos, como o de Dijkstra, que é também utilizado neste projeto.

Partindo de um vértice origem, exploramos todas os vértices alcançáveis a partir desse mesmo vértice, repetindo (posteriormente) este processo para os seguintes. Deste modo, descobre primeiro os vértices que se encontram mais perto da origem, passando sucessivamente aos restantes (que se encontram mais longe da origem). Para o contexto do problema presente, este algoritmo é usado para verificar os caminhos que são possíveis de percorrer através do vértice origem, sendo também usado para filtrar trilhos de dificuldade maior que a estipulada.

A complexidade temporal deste algoritmo é O(|V| + |E|), ou seja, linear relativamente ao tamanho do grafo (número de vértices e arestas), dado que cada vértice é visitado no máximo uma vez.

Quanto à complexidade espacial deste algoritmo, dado que o número máximo de vértices processados será |V|, concluímos que será também linear (O(|V|).

```
BFS(G, s):
1.      for each v ∈ V do discovered(v) ← false
2.      Q ← ∅

3.      ENQUEUE(Q, s)
4.      discovered(s) ← true

5.      while Q ≠ ∅ do
6.          v ← DEQUEUE(Q)
7.          pre-process(v)
8.          for each w ∈ Adj(v) do
9.              if not discovered(w) then
10.                 ENQUEUE(Q, w)
11.                 discovered(w) ← true
12.         post-process(v)
```

Pesquisa em profundidade

O algoritmo de pesquisa em profundidade pretende explorar as arestas a partir do vértice descoberto mais recentemente que ainda tenha arestas a sair dele. Quando todas as arestas do vértice forem exploradas, a função retorna de modo a explorar as arestas do vértice a partir do qual o vértice inicial foi descoberto, sendo, deste modo, um algoritmo recursivo.

Tal como o algoritmo de pesquisa em largura, a pesquisa em profundidade possui uma complexidade temporal linear, dado que cada vértice é visitado apenas uma vez. Para o problema presente, tal como a pesquisa em largura, este algoritmo é usado para percorrer o grafo do mapa e excluir trilhos de dificuldade elevada, a escolha do algoritmo usado para esta pesquisa é decidido pelo utilizador do programa.

A complexidade espacial deste algoritmo também é semelhante à do algoritmo descrito anteriormente. Apesar deste não fazer uso de uma fila, o facto da pesquisa em profundidade ser recursiva faz com que as chamadas recursivas da função guardem na stack |V| vértices (no pior caso), sendo assim O(|V|).

```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E} (∀ v ∈ V)

DFS(G):
1.  for each v ∈ V
2.      visited(v) ← false
3.  for each v ∈ V
4.      if not visited(v)
5.          DFS-VISIT(G, v)

DFS-VISIT(G, v):
1.  visited(v) ← true
2.  pre-process(v)
3.  for each w ∈ Adj(v)
4.      if not visited(w)
5.          DFS-VISIT(G, w)
6.  post-process(v)
```

Algoritmo de Dijkstra

O algoritmo de Dijkstra é um algoritmo ganancioso que tem como objetivo calcular o caminho mais curto entre dois vértices de um grafo dirigido pesado, sem arestas de peso negativo.

Este algoritmo tem um comportamento semelhante ao de Pesquisa em Largura, distinguindo-se deste pelo uso de uma fila de prioridade alterável onde os vértices prioritários são aqueles que permitem minimizar o peso total das arestas, daí ser um algoritmo ganancioso. No problema apresentado, os vértices prioritários serão aqueles com trilhos de menor dificuldade que a estipulada.

Ao processar cada vértice, é guardada informação do vértice anterior, até encontrar o vértice no topo da fila de prioridade, percorrendo-se, por fim, o caminho encontrado pelo sentido contrário, dando-se uma reconstrução do caminho que será retornado.

Na implementação do programa, este é o algoritmo que trata da geração da maioria dos circuitos para os clientes, testando as várias possibilidades e distâncias entre pontos de encontro.

Em relação à sua eficiência, a complexidade temporal do algoritmo é O((|V|+|E|) * log |V|). O número de extrações ou inserções na fila será, no máximo, |V|, sendo cada uma destas operações realizada em tempo logarítmico no tamanho da fila, de tamanho máximo |V|, sendo o tempo de execução de O(|V| * log |V|).

Quanto à operação de reordenação de vértices, esta é feita, no máximo, |E| vezes (uma vez por cada aresta), podendo também ser realizada em tempo logarítmico no tamanho da fila (|V|), resultando num tempo de execução de O(|E| * log |V|).

Assim a complexidade temporal total será O(|V| * log |V| + |E| * log |V|), ou seja, O((|V|+|E|) * log |V|).

```
DIJKSTRA(G, s):   // G=(V,E), s ∈ V
1.    for each v ∈ V do
2.       dist(v) ← ∞
3.       path(v) ← nil
4.    dist(s) ← 0
5.    Q ← ∅   // min-priority queue
6.    INSERT(Q, (s, 0)) // inserts s with key 0
7.    while Q ≠ ∅ do
8.       v ← EXTRACT-MIN(Q) // greedy
9.       for each w ∈ Adj(v) do
10.         if dist(w) > dist(v) + weight(v,w) then
11.            dist(w) ← dist(v)+ weight(v,w)
12.            path(w) ← v
13.            if w ∉ Q then  // old dist(w) was ∞
14.               INSERT(Q, (w, dist(w)))
15.            else
16.               DECREASE-KEY(Q, (w, dist(w)))
```

Algoritmo A*

Este algoritmo trata-se de um melhoramento do algoritmo de Dijkstra, sendo em tudo idêntico a este exceto na forma em como se dá a ordenação dos vértices na fila de prioridade.

Assim, com recurso a uma função heurística o algoritmo dá prioridade aos vértices que se encontram mais próximos do destino somando-se ao peso total (peso entre o vértice atual e a origem) o valor da estimativa do peso do próprio vértice ao vértice de chegada, arredondado para baixo.

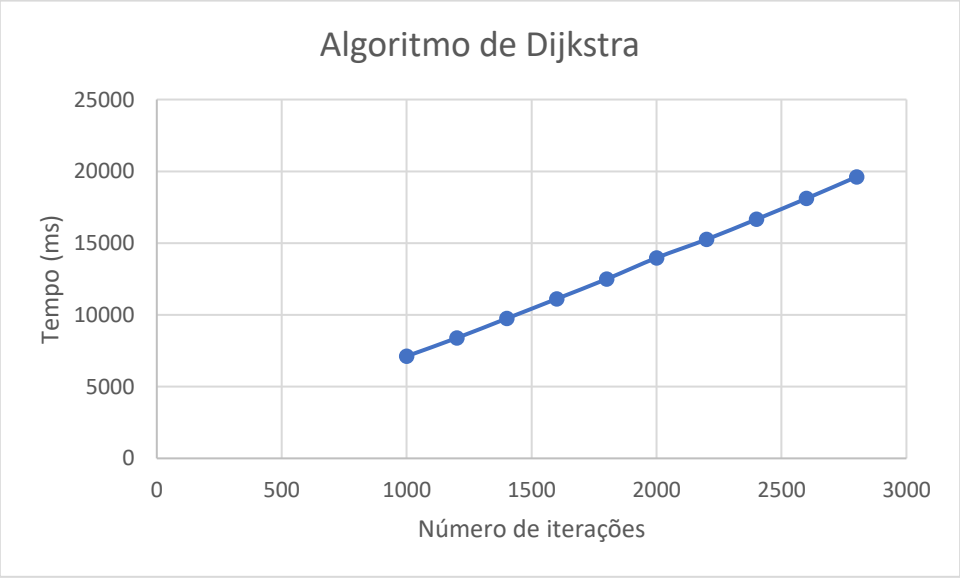
Este algoritmo é utilizado no programa para a mesma função do algoritmo de Dijkstra, sendo mais eficiente em questões de tempos de processamento.

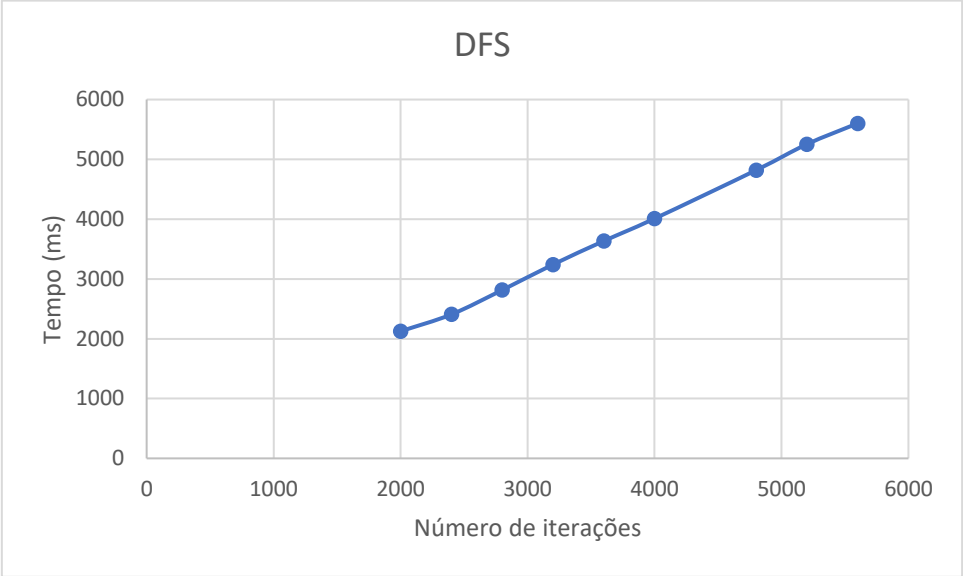
Este método nem sempre garante a solução ótima, a não ser que sejam cumpridas certas condições como, por exemplo, o peso das arestas ser distâncias em km, entre outras.

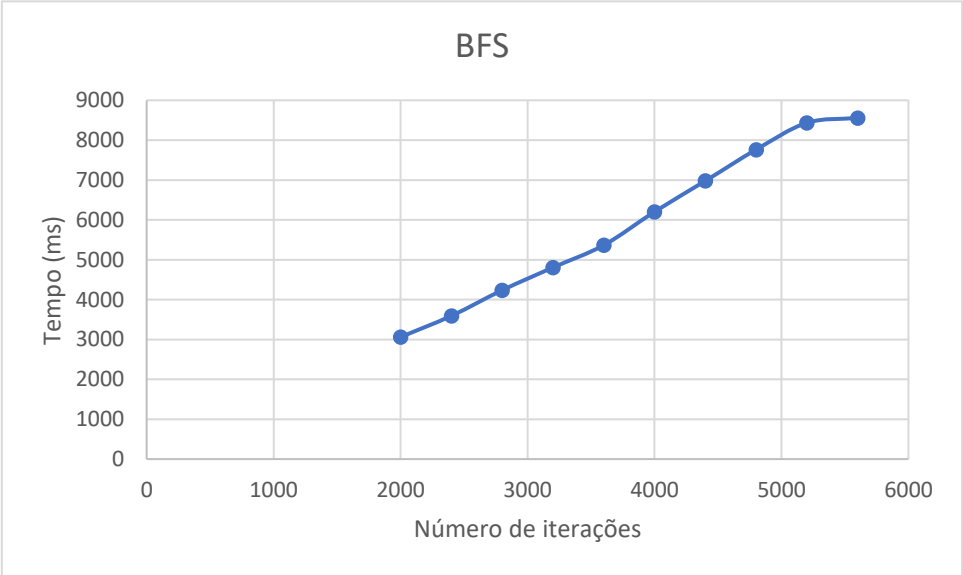
Análise temporal empírica dos algoritmos

Para efetuar uma análise temporal ao processamento do programa necessária a implementação da classe Timer. Esta é instanciada quando se inicia a geração de caminhos para os trabalhadores, e mede o seu tempo de execução.

Verifica-se que ao aumentar o número de trabalhadores ou ao selecionar a opção de gerar diferentes caminhos (do ponto de encontro para o almoço até ao ponto final de encontro), o tempo de processamento aumenta, no entanto este é relativamente baixo. Quanto à análise temporal empírica de todos os algoritmos utilizados em concreto obtemos os seguintes gráficos:







Observando os gráficos obtidos, concluímos que os algoritmos têm complexidade temporal linear, tal como esperado na análise temporal teórica

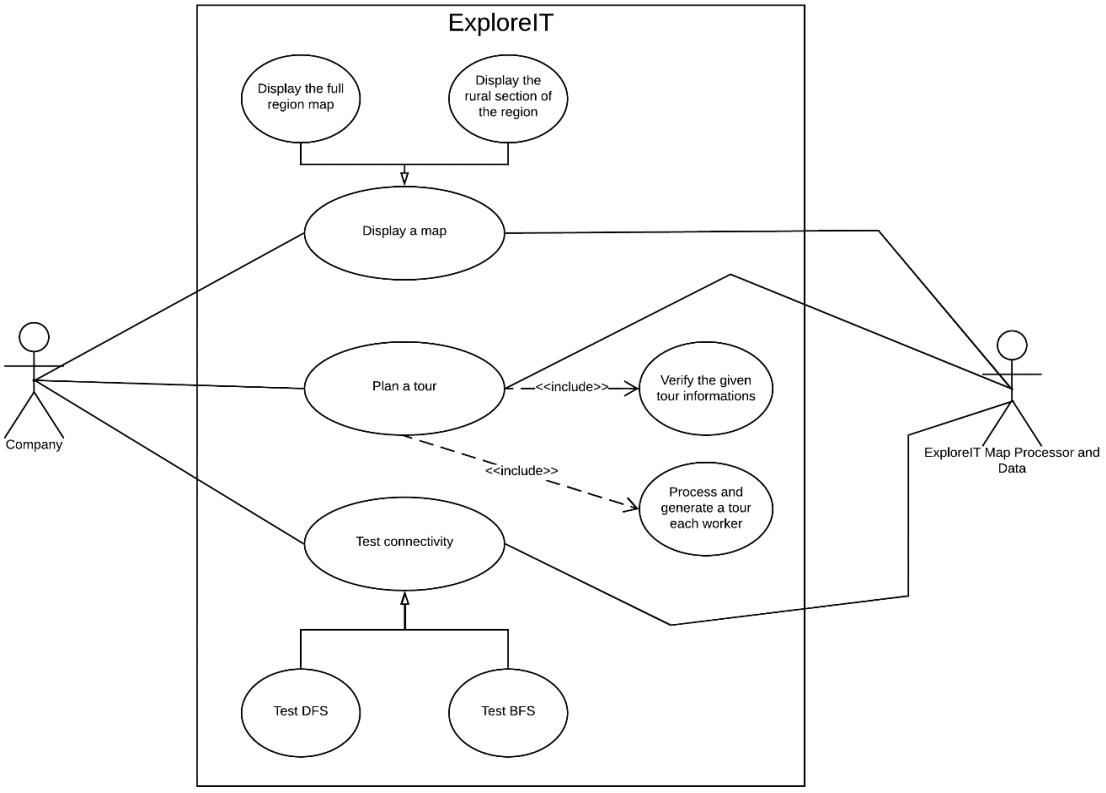
Identificação dos Casos de Utilização e Funcionalidades / Principais Casos de Uso

Na aplicação implementada, está presente uma interface simples que permite a interação com o utilizador recorrendo a vários menus. Nesta, a empresa pode visualizar os mapas da região e agendar um *tour* para os seus trabalhadores.

As duas principais funcionalidades que se procuram apresentar ao utilizador é a **visualização do mapa de uma região** e o **planeamento de um circuito**, existindo também a opção de **avaliação da conectividade do mapa.**

Nesta **primeira opção**, é possível visualizar a secção da região selecionada (a cidade completa ou apenas a sua região rural).

Na **segunda opção**, já é necessário o fornecimento do tamanho do grupo, bem como a capacidade técnica de cada elemento e a duração máxima do percurso. Após a inserção destas informações , o programa irá verificar através de diversos algoritmos se existem caminhos diferentes para cada trabalhador que se adequem às suas preferências (dificuldade e horários) dadas anteriormente.

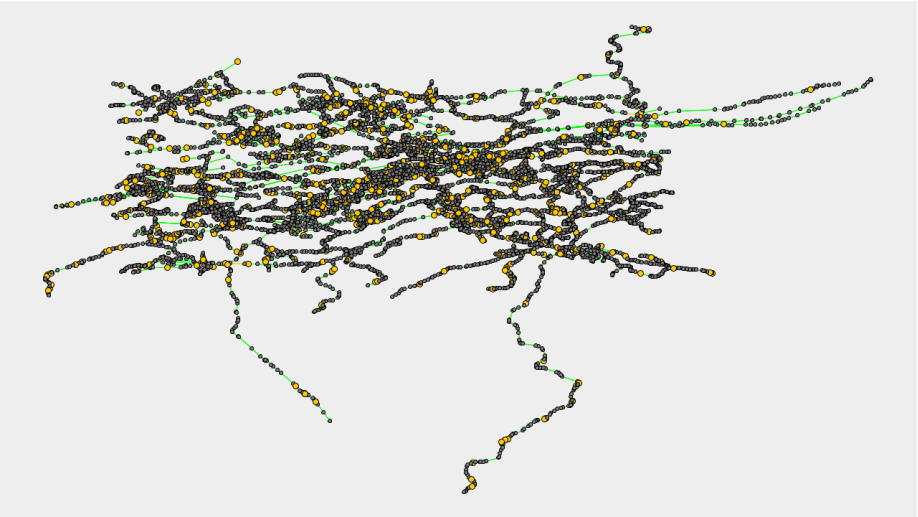


Conectividade dos Grafos Utilizados

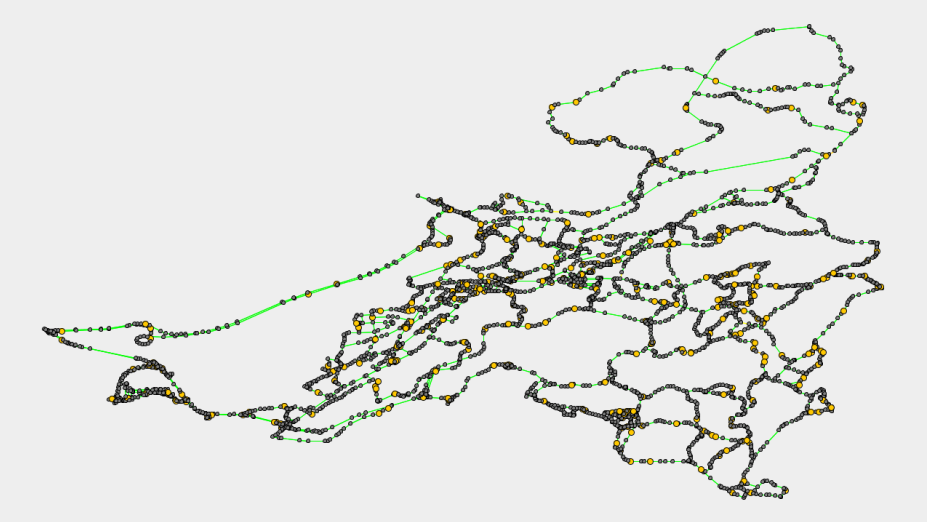
Para os dados do programa, foram utilizados dois mapas da cidade de Penafiel, penafiel_full e penafiel_strong que, no contexto do problema, são apresentados como o mapa da cidade completa e o mapa da região rural da cidade (área fortemente conexa).

O grafo da cidade completa (penafiel_full), é utilizado para a avaliação de conectividade e filtragem dos caminhos por dificuldade, uma das opções presentes no menu.

Já o grafo da área fortemente conexa (penafiel_strong), é utilizado para o planeamento e geração dos circuitos para os trabalhadores.



Grafo da cidade completa (penafiel_full)



Grafo da área fortemente conexa (penafiel_strong)

Estruturas de Dados Utilizadas

Ao longo da execução do programa, são utilizadas várias estruturas de dados, quer para implementar os algoritmos necessários, quer para guardar os dados resultantes dos mesmos.

De modo a implementar os algoritmos de Dijkstra e A*, utilizámos a implementação de uma mutable_priority_queue que foi fornecida para as aulas práticas.

Os caminhos devolvidos pelo programa são armazenados num vetor de objetos da classe Path, sendo posteriormente atribuídos a cada um dos trabalhadores, sendo que estes também são armazenados num vetor (de Workers).

A atribuição de caminhos aos trabalhadores é feita agrupando os caminhos em 3 diferentes filas (queue), sendo que cada fila possui caminhos de uma mesma dificuldade. Deste modo, maximizamos a diversidade dos caminhos atribuídos aos trabalhadores.

Por último, os dados relativos ao mapa utilizado foram guardados num grafo dirigido pesado, constituindo a base de todo o processamento.

Conclusão

Ao longo deste relatório, foi apresentada uma possível solução para o problema proposto. Foram também expostas e debatidas as várias técnicas e algoritmos que constituem esta solução, tal como as diferentes etapas de implementação do programa.

Durante a realização do programa, fomos sentindo algumas dificuldades em vários aspetos do seu desenvolvimento. Primeiramente, tivemos dificuldade na formulação de uma solução, algo que pensamos ser devido ao carácter vago e pouco preciso do enunciado, esta "dificuldade" acabou por se tornar numa mais valia, dando-nos liberdade para a implementação das *features* que achamos necessárias para o programa. Tivemos também dificuldade no próprio desenvolvimento da solução pretendida, algo relacionado com o elevado número de fatores a ter em conta para uma correta abordagem da solução (criação de trilhos tendo em atenção a sua dificuldade, a necessidade de pontos em comum e do horário a cumprir), tendo sido experimentados e testados vários algoritmos para resolver o problema.

Para concluir, todas as tarefas necessárias para a elaboração do relatório foram distribuídas igualmente por todos os elementos do grupo, estando cada um constantemente a par do progresso e desenvolvimento do mesmo. Assim, o esforço dedicado por cada elemento do grupo foi igualmente repartido.

Bibliografia

- **Relatórios fornecidos pelo docente.**
- **Slides apresentados na aula teórica.**
- **Implementação da classe Timer utilizada no programa**
  https://gist.github.com/mcleary/b0bf4fa88830ff7c882d
- **https://en.wikipedia.org/wiki/Depth-first_search**
- **https://en.wikipedia.org/wiki/Breadth-first_search**
- **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm**
- **https://en.wikipedia.org/wiki/A*_search_algorithm**