Faculdade de Engenharia da Universidade do Porto

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Projeto de Concepção e Análise de Algoritmos:

TourMateApp: rotas turísticas urbanas adaptáveis

Turma 2, Grupo 1:

Breno Accioly de Barros Pimentel	up201800170@fe.up.pt
Diogo Miguel Borges Gomes	up201806572@fe.up.pt
Luís Filipe Sousa Teixeira Recharte	up201806743@fe.up.pt

Índice

Descrição do tema

No âmbito da unidade curricular concepção e análise de algoritmos foi-nos atribuído o tema: **TourMateApp** - rotas turísticas urbanas adaptáveis. Esta app consiste numa ferramenta que tem por objetivo fornecer aos utilizadores, segundo os seus interesses e os tempos que estes disponibilizam, os circuitos da rede, onde estes se encontram, que passam num maior número de pontos de interesse, usando um meio de transporte também à escolha do utilizador. Esta aplicação será útil a turistas e a pessoas em viagens de trabalho.

Para a implementação dos algoritmos associados a esta aplicação teremos de definir um grafo direcionado, alvo destes algoritmos, com um conjunto de nós que representam os pontos da cidade e um conjunto de arestas com pesos que retratam as distâncias e tempos entre os nós.

Avaliamos também a conectividade do grafo, com intenção de evitar que locais de interesse (POIs) se encontrassem em zonas inacessíveis da rede. Estas zonas poderão estar sem acesso se, por exemplo, houverem obras em redes públicas.

Identificação do problema

O problema em foco pode ser dividido em 5 partes fundamentais:

1. **Verificar a possibilidade de deslocação entre os dois pontos**

   Devido a vários fatores e de acordo com o meio de transporte a ser utilizado, a deslocação do utilizador pode ou não ser possível.

   Quando é escolhido o transporte público o percurso é adaptado a partir do vértice que contém a paragem, sendo adicionado um percurso a pé até essa paragem.

   Só é possível realizar o percurso pedido, caso todos os pontos estejam ligados entre si, o que significa que estes **devem fazer parte de uma componente fortemente conexa do grafo**. Caso não seja possível realizar este percurso, o utilizador é informado.

2. **Cálculo de distâncias e tempos**

   Assumindo que é possível efetuar a deslocação, é necessário calcular as distâncias entre os vértices do grafo e o tempo que as mesmas demoram a ser percorridas pelos meios de transporte (usam-se as velocidades médias dos meios de transporte para calcular este tempo).

3. **Encontrar o percurso mais curto**

   Dando uso aos algoritmos, é necessário encontrar um caminho possível entre os dois pontos. Começamos por procurar o caminho mais curto sem ter em conta as preferências do utilizador.

4. **Encontrar todos os percursos possíveis**

   Apesar do caminho mais curto ser o melhor em relação ao tempo, pode não satisfazer as preferências do utilizador relativamente aos seus pontos de interesse. Portanto é necessário encontrar percursos alternativos que possam abranger um maior número de pontos de interesse.

5. **Escolha de percursos de acordo com as preferências**

   É necessário filtrar os percursos de acordo com as preferências do utilizador. Descartam-se todos os percursos que ultrapassem o tempo definido, e dá-se prioridade aos percursos que abrangem um maior número de pontos de interesse.

Formalização do problema

Input

G<Vi,Ei>: Grafo dirigido pesado, que representa a rede da cidade, composto por:

- **V:** Pontos da cidade
  - **POI:** tag que identifica o ponto de interesse (null se o vértice não for POI);
  - **ID:** identificador do vértice;
  - **Adj:** conjunto de arestas que partem do vértice;

- **E:** Arestas (que retratam as estradas da rede da cidade)
  - **dist:** peso da aresta (distância entre dois pontos);
  - **t:** tempo para percorrer a distância;
  - **ID:** identificador da aresta;
  - **dest:** vértice de destino;

Pi: ponto de partida do utilizador;
Pf: ponto de chegada do utilizador;

Transporte: Meio de transporte a utilizar;

Se o utilizador escolher transportes públicos também serão adicionados ao input:
A[i]: Conjunto das rotas dos autocarros disponíveis ao utilizador que contêm;
M[i]: Conjunto das rotas dos metros disponíveis ao utilizador;

Te: Tempo disponibilizado pelo utilizador.

Output

list<G<Vs,Es>>: Lista de grafos dirigido pesado conexo que contém os caminhos disponíveis ao utilizador. Os vértices e arestas têm os mesmos atributos do grafo de entrada.
nPOI: número de pontos de interesse de cada caminho;
Ts: Tempo de demora a percorrer cada caminho.

Restrições

- Te>=Ts;
- $\exists$ vi $\in$ Vi : vi = Pi;
- $\exists$ vi $\in$ Vi : vi = Pf;
- Te>0;
- nPOI>=0.

Função objetivo

O objetivo passa por recomendar caminhos de forma a maximizar o número de pontos de interesse pelo qual o utilizador passa de acordo com o perfil e preferências do mesmo, entre os pontos enunciados pelo utilizador à priori, dentro de um espaço de tempo limitado. Sendo assim as funções objetivo são as seguintes:

- **max(nPOIs);**
- **nPOI=count(G<Vs>)**

Perspetiva de solução

Com o objetivo de encontrar o melhor itinerário, foi realizada a análise dos algoritmos e estruturas capazes de encontrar diferentes pontos de interesse num determinado intervalo de tempo, seguindo uma trajetória definida com os pontos inicial e final por parte do utilizador.

Atendendo a esses requisitos, poderá ser utilizado o algoritmo de **Dijkstra**. Esse consegue encontrar o caminho mais curto em grafos pesados. Além disso, como pesos negativos não existem nas arestas do grafo a ser implementado, este é válido na resolução do caso. O algoritmo usa uma estratégia gananciosa, escolhendo os vértices mais apropriados nas suas iterações, e possui tempo de execução **O ((|V| + |E|) log |V|)**. [1]

O grafo a ser utilizado deve incluir informações acerca dos pontos de interesse nos seus vértices. As suas arestas devem considerar a distância e os diferentes tempos de percurso: a pé, de carro ou de transporte público.

Do mesmo modo, pode ser implementado variações do algoritmo, a fim de aumentar a eficiência. Como, por exemplo, o **algoritmo A***.

Para encontrar a menor distância, d, entre dois vértices usando as suas latitudes(Φ) e longitudes(λ), é possível usar a fórmula de Haversine:

$$d = 2r\sin^{-1}\sqrt{\sin^2\left(\frac{\Phi_2-\Phi_1}{2}\right) + \cos(\Phi_1)\cos(\Phi_2)\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}$$

Onde r, é o raio da Terra. É importante notar que o resultado da fórmula é uma aproximação, já que a Terra não é uma esfera perfeita.

Também deve ser necessário um algoritmo para deteção de pontos de articulação no grafo, como forma de evitar que pontos de interesse sejam escolhidos em zonas inacessíveis da rede. Pensando nisso, o algoritmo de **pesquisa em profundidade (depth-first search)** fornece um tempo linear **O(|E|+|V|)** para encontrar todos os pontos de articulação num grafo conexo.

Dessa forma, a partir de um vértice qualquer, é realizado uma pesquisa em profundidade, numerando-os à medida que são visitados. Para cada vértice, v, é definido o seu número de pré-ordem ***Num(v)***. Então, é calculado o menor número de vértices que se atinge com zero ou mais arestas na árvore e possivelmente uma aresta de retorno, ***Low(v)***, para cada vértice, v, na árvore de visita em profundidade.

Dada essas informações, a raiz é um ponto de articulação se e somente se tiver mais que um filho na árvore, uma vez que, caso essa tenha dois filhos, a remoção da raiz divide o grafo em diferentes árvores e, se esta possui apenas um filho, a sua remoção apenas a desconecta. Qualquer outro vértice v é um ponto de articulação se tiver um filho w, tal que ***Low(v) ≥ Num(v)***. [2]

Algoritmos efetivamente implementados:

No decorrer do projeto foram utilizados o algoritmo de pesquisa em profundidade (**DFS**), de **Dijkstra** e o de **Yen**. Estes algoritmos funcionaram todos sobre o grafo onde o mapa estará guardado.

Pesquisa em profundidade (DFS):

O algoritmo de pesquisa em profundidade é utilizado na nossa aplicação para avaliar a conectividade dos grafos.

Pseudo-código

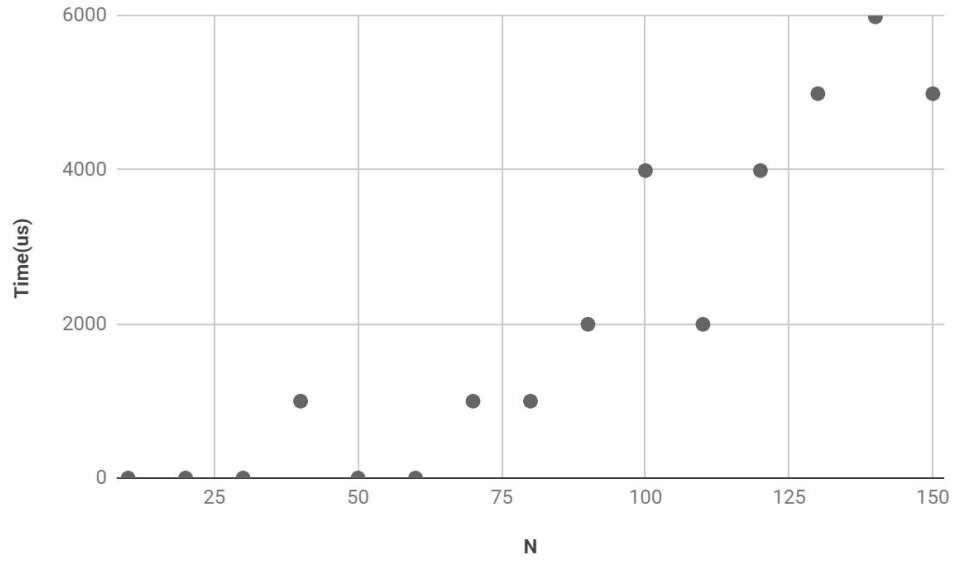
```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E} (∀v ∈ V)

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)
```

Análise teórica

Como o algoritmo DFS visita todos os vertices e todas as arestas do grafo a sua complexidade temporal é **O(|E| + |V|)**.

A complexidade espacial dependerá do pré processamento e pós processamento que é feito neste algoritmo. No caso do algoritmo implementado no nosso projeto como guardamos todos os vértices num vetor (de forma a verificar a conectividade usando uma função auxiliar) a complexidade espacial será O(|V|).

Análise temporal empírica



Algoritmo de Dijkstra:

O algoritmo de Dijkstra tem como função encontrar o caminho mais curto entre dois pontos de um grafo direcionado ou não direcionado, tendo em conta o peso das arestas do mesmo. É de salientar que o Dijkstra usa uma fila de prioridade mínima para ir guardando os próximos vértices a processar dando prioridade aos de menor distância ao vértice de partida.

Este algoritmo servirá como auxiliar ao algoritmo de Yen que será posteriormente explicado.

Pseudo-código

```
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)))
```

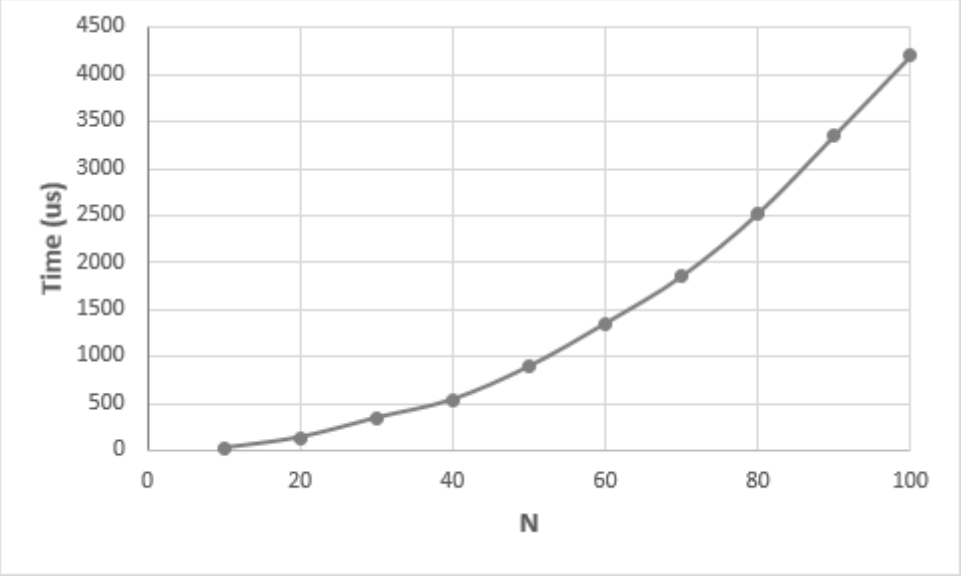
Tempo de execução:
O((|V|+|E|) * log |V|)

Análise teórica

O tempo de execução **O((|V|+|E|)*log|V|)** é resultado da soma do tempo que o algoritmo demora na extração e inserção de |V| elementos na fila de prioridades (cada extração/inserção demora log(N)) com o tempo que a operação Decrease-Key da fila leva a ser feita (|E|*log|E|).

Como neste algoritmo se armazenam, no máximo, todos os vértices na fila de prioridade a complexidade espacial é O(|V|).

Análise temporal empírica



Algoritmo de Yen:

O algoritmo foi usado para encontrar um determinado número de caminhos entre dois pontos, sendo esses os pontos de início e destino escolhidos pelo utilizador. Dessa forma, a partir de um percurso inicial escolhido pelo algoritmo de Dijkstra, o algoritmo de Yen procede ao fazer desvios no caminho removendo arestas que garantem que o próximo percurso escolhido será diferente do anterior.

Este foi necessário para estudar o conjunto de rotas disponíveis ao utilizador e para recomendar o melhor resultado.

Pseudo-código

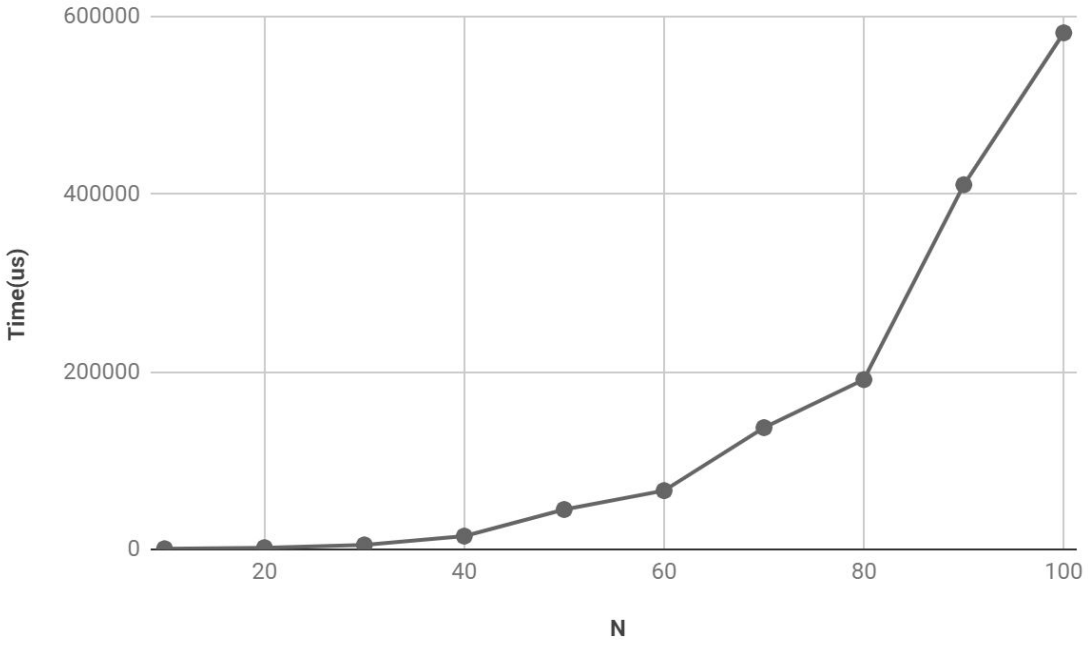
```
YenKSP(sourceID, destID, maxTime):
   // Determine the shortest path from the source to the destID.
  A[0] ← Dijkstra(sourceID, destID);
  // Initialize the set to store the potential kth shortest path.
  B ← [];

  for k from 1 to K:
     // The spur node ranges from the first node to the next to last node in the previous k-shortest path.
     for i from 0 to size(A[k − 1]) − 2:
        // Spur node is retrieved from the previous k-shortest path, k − 1.
        spurNode ← A[k-1].node(i);
        // The sequence of nodes from the source to the spur node of the previous k-shortest path.
        rootPath ← A[k-1].nodes(0, i);

        for each path p in A:
           if rootPath == p.nodes(0, i):
              // Remove the links that are part of the previous shortest paths which share the same root path.
              remove p.edge(i,i + 1) from Graph;

        // Calculate the spur path from the spur node to the destID.
        spurPath ← Dijkstra(spurNode, destID);

        // Entire path is made up of the root path and spur path.
        totalPath ← rootPath + spurPath;
        // Add the potential k-shortest path to the heap.
        if (totalPath not in B):
           B.append(totalPath);

        // Add back the edges and nodes that were removed from the graph.
        restore edges to Graph;

     if B is empty:
        // This handles the case of there being no spur paths, or no spur paths left.
        break;
     // Add the lowest cost path becomes the k-shortest path.
     A[k] ← B[0];
     B.pop();
  return A;
```

Análise teórica

A complexidade depende do algoritmo de Dijkstra O(|E| + |V|*log|V|) e do número de chamadas desse para computar os caminhos alternativos O(K*|V|), onde K é o número de caminhos, com o tempo de execução resultante sendo **O(K*|V| (|E| + |V|*log|V|))**.

Para armazenar o vetor A de caminhos escolhidos e a heap B de possíveis caminhos, além de guardar o estado inicial do grafo para retornar as arestas removidas no processo, a complexidade espacial resultante é **O(|V|² + K|V|)**.

Análise temporal empírica



Os grafos utilizados nos testes são bidirecionais, influenciando assim o seu tempo de execução, entretanto, a sua complexidade não foi alterada.

Principais casos de uso implementados

A aplicação contém uma interface com diversos menus onde o user pode navegar. O menu inicial é composto por **User settings**, **Tour** e **Graph Settings**.

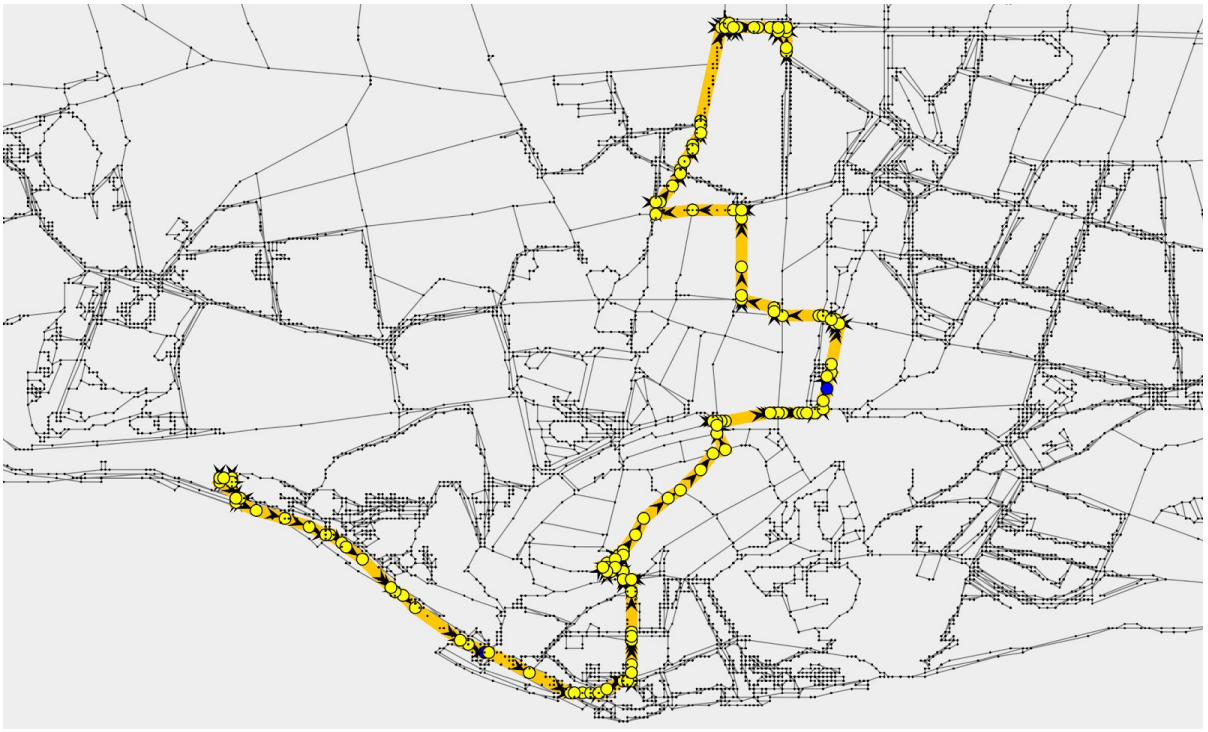
- **User settings:** aqui o user pode alterar as suas informações e adicionar preferências por onde quer passar na realização do seu caminho.
- **Tour:** O sub menu principal do programa, é onde se cria a tour, se pede os pontos iniciais e finais, o tempo disponível e o transporte. É aqui que se mostram os percursos recomendados.
- **Graph settings:** Sub menu onde é possível fazer load de grid graphs apenas.

Depois de avaliadas as preferências do utilizador, e obtida a informação necessária, a aplicação diz ao utilizador se é ou não possível realizar a rota indicada. Se for possível é-lhe mostrado o melhor itinerário e outras sugestões.

Exemplos de percursos criados:

- ➔ **Vértices e arestas a rosa representam a linha de metro implementada por nós.**
- ➔ **Percursos a pé representados por vértices amarelos e arestas laranjas,** que são as deslocações do utilizador à entrada mais próxima do metro e à saída mais próxima do ponto final.

➔ **Percurso a pé representado por vértices amarelos e arestas laranjas,** que representam o percurso sugerido ao utilizador. **Vértices azuis** são pontos turísticos indicados pelo utilizador como preferenciais.



Notas Finais

No decorrer desta componente prática tentámos várias abordagens ao problema, para além da nossa proposta de solução com outros algoritmos mas sem sucesso devido à ineficiência da maior parte deles, até encontrarmos o algoritmo Yen(k shortest paths) que se insere perfeitamente na nossa proposta.

No entanto, apercebemo-nos que a nossa proposta de solução não é de todo a mais eficiente. Embora o algoritmo **Yen seja um ótimo algoritmo para encontrar os k caminhos mais rápidos**, não é tão bom para encontrar o caminho ótimo que contém o maior número de preferências do utilizador, uma vez que isso obriga a calcular um número bastante elevado de caminhos possíveis, o que num grafo com demasiados nós se torna muito ineficiente.

Assim, a solução que pensámos que fosse melhor para além da nossa proposta seria usar o Dijkstra para calcular a distância entre todos os pontos de interesse inseridos pelo utilizador.

Após isso, criar um grafo composto apenas por nós que são pontos de interesse e aí sim aplicar o algoritmo Yen. O número de caminhos ia ser muito mais limitado o que iria resultar numa resolução melhor e mais rápida para o problema.

Infelizmente esta ideia surgiu-nos tarde demais para a conseguirmos implementar a tempo da entrega do projeto.

Conclusão

A realização deste trabalho permitiu-nos obter uma melhor compreensão da aplicação dos algoritmos ao mundo real, além disso permitiu-nos discussões interessantes acerca de otimização e análise de algoritmos.

Após pesquisas intensas sobre o problema em questão encontramos um novo algoritmo, que não é muito conhecido, no entanto ajudou-nos a implementar a nossa proposta de solução, embora nos tenhamos apercebido durante o desenvolvimento do projeto que esta não é a melhor abordagem ao problema.

Concluímos, portanto, que os objetivos pretendidos neste relatório foram atingidos, quer a nível individual quer a nível coletivo, uma vez que cada elemento domina agora os temas lecionados na unidade curricular.

Esforço dedicado por cada elemento do grupo

Todos os elementos do grupo trabalharam de forma igual, discutindo ideias, soluções e otimizações para o problema em questão. O trabalho foi distribuído por todos.

Bibliografia

[1] Cormen, Thomas H. *Introduction to algorithms (Third Edition)*
[2] Weiss, Mark Allen. *Data structures and algorithm analysis in C++ (Fourth Edition)*
[3] *https://en.wikipedia.org/wiki/Yen%27s_algorithm#Description*