ExploreIT

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Concepção e Análise de Algoritmos

Grupo 9 Turma 2
João Romão - up201806779
L. Miguel Pinto - up201806206
Tiago Alves - up201603820

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

29 de Maio de 2020

Conteúdo

1 Descrição do problema **4**
- 1.1 1ª Iteração: Cálculo de caminhos que considerem a hora de chegada aos pontos de confluência . . . 4
- 1.2 2ª iteração: Caminhos que considerem a hora de chegada e a dificuldade dos trilhos . . . 4
- 1.3 3ª Iteração: Caminhos que considerem hora, dificuldade e que maximizem o número de pontos de interesse . . . 5
- 1.4 4ª Iteração: Pontos de interesse obrigatórios . . . 5

2 Formalização do Problema **6**
- 2.1 Dados de Entrada . . . 6
- 2.2 Dados de saída . . . 6
- 2.3 Restrições de dados . . . 7
- 2.4 Funções Objetivo . . . 8

3 Perspetiva de solução **9**
- 3.1 Análise da Conectividade . . . 9
  - 3.1.1 Pesquisa em Profundidade (DFS) . . . 10
  - 3.1.2 Algoritmo de Pesquisa em Largura . . . 10
- 3.2 Pré-Processamento dos Dados de Entrada . . . 11
  - 3.2.1 Pré-processamento do grafo . . . 11
- 3.3 Pré processamento de pontos Default . . . 13
  - 3.3.1 Pré-Processamento associado ao cálculo de dificuldades . 14
- 3.4 Identificação dos problemas encontrados . . . 14
- 3.5 Considerações Globais . . . 14

4 Estratégia usada e Algoritmos implmentados **16**
- 4.1 A* Com heurística adaptada . . . 17
- 4.2 Divisão em sub-Grupos . . . 20
- 4.3 Pontos de interesse obrigatórios: . . . 20
- 4.4 Algoritmo do vizinho mais próximo . . . 21

5 Casos de utilização Implementados **22**

6 Estruturas de Dados Utilizadas **23**
- 6.1 Representação do Grafo . . . 23
  - 6.1.1 Grafo . . . 23
  - 6.1.2 Nós . . . 23
  - 6.1.3 Arestas . . . 24
- 6.2 Carregamento de Dados . . . 24
- 6.3 Representação dos mapas e caminhos encontrados . . . 24
- 6.4 Menu . . . 24

7 Análise da Conectividade **25**
- 7.1 Pesquisa em Profundidade (DFS) . . . 25
- 7.2 Análise de resultados . . . 25

8 Análise dos Algoritmos efetivamente implementados **26**
8.1 Floyd-Warshall 26
8.1.1 Análise Teórica 26
8.1.2 Análise Empírica 26
8.2 A* 27
8.2.1 Análise Teórica 27
8.2.2 Aplicação no problema 27
8.2.3 Análise Empírica 28
8.2.4 Primeira Iteração 28
8.2.5 Segunda e Terceira Iteração 29
8.3 A* Para vários pontos de confluência 30
8.3.1 Resultado obtido 31
8.4 Pontos de Interesse Obrigatórios 31
8.5 DFS 31
8.5.1 Análise Teórica 31
8.5.2 Análise Empírica 32

9 Observações em relação à Implementação **32**

10 Conclusão **33**

1 Descrição do problema

Neste trabalho, 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 sendo que cada trilho tem um grau de dificuldade e uma duração estimada, função da distância e do declive do mesmo.

A aplicação deverá ser capaz de aconselhar caminhos consoante o número de grupos pretendidos e deve ser capaz de sincronizar os grupos para que cheguem ao mesmo tempo aos pontos de confluência.

Este problema pode ser dividido em iterações principais. No entanto, de modo a implementar algumas funcionalidades adicionais, surge uma última dando um total de 4 iterações:

1.1 1ª Iteração: Cálculo de caminhos que considerem a hora de chegada aos pontos de confluência

Inicialmente, apenas é considerado que os melhores caminhos, serão aqueles que minimizem a diferença entre a hora de chegada e a hora pretendida de todos os grupos até aos pontos de afluência. Deste modo elimina-se todos os caminhos que resultam em trilhos demasiado curtos ou longos, que poderiam aparecer na pesquisa.

Caso exista um tempo pré definido entre os pontos de afluência, como 3 horas devido a preocupações de segurança, o grafo poderá ser pré processado de modo a eliminar todos os nós que estejam a um tempo superior ao pré definido a qualquer ponto de afluência.

Adicionalmente deve ser considerado que por diversos motivos, como obras nas vias públicas, certos caminhos podem tornar-se inacessíveis, sendo as arestas correspondentes no grafo ignoradas durante o pré processamento.

Pelos motivos descritos acima e de modo a identificar os destinos com pouca acessibilidade e a existência de alternativas, torna-se necessário efetuar uma análise de conectividade ao grafo.

1.2 2ª iteração: Caminhos que considerem a hora de chegada e a dificuldade dos trilhos

Numa segunda iteração, é considerada a dificuldade de cada trilho e os níveis de experiência do grupo que irá realizar o percurso. Os trilhos variam bastante no seu declive, alguns tendo até escalada, sendo fortemente desaconselhados a pessoas que não tenham tanta experiência.

Deste modo, as pessoas com um certo nível de experiência serão indicadas apenas para trilhos entre dois certos níveis de dificuldade, sendo a média equivalente à sua experiência.

Trilhos de dificuldade diferente podem partilhar troços de caminho com trilhos de uma dificuldade diferente, desde que depois sigam por outro caminho mais adequado à sua dificuldade. Esta poderá variar entre 1 e 10, sendo a um a mais fácil e 10 a mais difícil.

1.3 3ª Iteração: Caminhos que considerem hora, dificuldade e que maximizem o número de pontos de interesse

Nesta terceira fase pretende-se tornar os percursos o mais atrativo possível para os seus utilizadores, tentando no seu trilho maximizar o número de pontos de interesse que este visita. Será então feita uma relação entre o nível de dificuldade pretendido, a hora de chegada e o número de pontos de interesse visitados de modo a calcular o/os trilhos mais adequados para cada utilizador.

De modo a ser ainda mais interessante, pretende-se nesta fase eliminar ciclos de modo a os caminhantes verem o máximo de paisagens diferentes possíveis.

1.4 4ª Iteração: Pontos de interesse obrigatórios

Nesta última iteração, considera-se o caso de o utilizador querer que todos os grupos passem por determinados pontos de interesse obrigatoriamente, como por exemplo na maior atração turística do mapa, não tendo como restrição a hora em que cada grupo visita o sítio, pois não o precisam de visitar todos ao mesmo tempo.

Poderá ser interessante definir uma ordem em que são visitados os pontos de interesse, de modo a que os que estão mais próximos do ponto inicial sejam visitados antes dos que estão mais próximos do ponto final. Isto resultaria num trilho que geograficamente faria mais sentido, tendo uma sequência lógica.

Caso não seja possível passar por algum dos pontos por ser inacessivel ou por não ser possivel passar pelo ponto de interesse estando a hora marcada no próximo ponto de confluência, este será descartado.

2 Formalização do Problema

2.1 Dados de Entrada

Ln – Sequência de locais de confluência dos percursos,ordenados, sendo L(n) o n-ésimo local. Cada Local é caracterizado por

- N - Nó do Grafo correspondente
- $HConf$ – Hora de Confluência
- $TConf$ – Quanto tempo se passa em cada ponto de confluência
- PI - Ponto de partida - $L(1)$
- PF - Ponto de chegada - $L[Length(L) - 1]$

$TMax$ – Tolerância máxima entre a diferença da hora marcada e da hora de chegada de um grupo para cada ponto de confluência.

Di – Lista dos vários valores de dificuldade dos trilhos a calcular, sendo D(n) o n-ésimo valor de dificuldade.

$Ge(Ve,Ee)$ – Grafo Não dirigido, cíclico e pesado, constituído por:

- V – Vértice, representa um ponto do trilho
  1. Lat – Representa a latitude do ponto
  2. Lon – Representa a longitude do ponto
  3. Alt – Representa a altitude do ponto
  4. $Adj \subset E$, arestas que partem do vértice.
- E – Aresta, representa o caminho que une dois pontos dos trilhos
  1. Dec – Declive do caminho
  2. Ta – Tempo estimado a percorrer a aresta
  3. $Dest \in V$ - destino da aresta
- Poi – Lista dos pontos de interesse possíveis de serem visitados.

2.2 Dados de saída

LF – Lista com os locais de confluência, sendo L(n) o n-ésimo local de confluência, constituídos por

- Tc – Lista ordenada crescente das horas de chegada dos diferentes grupos ao local.

Ti – Lista de trilhos criados, sendo T(n) o n-ésimo trilho, constituídos por:

- Poi - Pontos de interesse visitados.
- D – Dificuldade do trilho
- Tt - Tempo total que demora a percorrer o trilho.
- P = Conjunto ordenado de arestas percorridas no trilho.

2.3 Restrições de dados

Dados de entrada:

- $|Ln| >= 2$, visto que tem que existir pelo menos um ponto inicial e um ponto final.
- $\forall n \in [1, |Ln| - 1], HConf[n+1] > HConf[n], n \in N$, visto que a Hora de confluência pretendida de um ponto tem que ser mais tarde do que a hora de confluência anterior.
- $TMax > 0$, visto que o tempo tem que ser positivo
- $\forall n \in [0, Di], (1 <= D(n) <= 10), n \in N \wedge (D(n) \in N)$ - Todas as dificuldades têm que estar entre 1 e 10
- $|D| > 0$, visto que precisa de se querer calcular pelo menos um caminho
- $\forall n \in [1, Ln], L(n) \in Ge, n \in N$, visto que todos os pontos de Confluência têm de pertencer ao mesmo grafo conexo.
- $\forall n \in [1, Ln], L(n) \in Poi, n \in N$, visto que pontos de confluencia tem que ser POIS.
- $\forall e \in E, -1 <= e.Decl <= 1$, declive entre -1 e 1, -100% a 100%

Dados de saída:

- $L_f \subset L_i \wedge L_i \subset L_f$, os locais de afluência têm que ser os mesmo no ínicio e no fim do algoritmo
- $T(i+1) > T(i)$, A lista de tempos retornada é ordenada
- $T(i) > 0$, visto que o tempo é sempre positivo
- $Tt > 0$
- $(Tc - HConf) < TMax$, a diferença entre a hora de chegada e a hora marcada tem que ser inferior a TMax

2.4 Funções Objetivo

O objetivo do programa é diminuir a diferença entre a hora de chegada prevista e a hora de chegada dos grupos, enquanto se tenta diminuir a distância temporal entre as chegadas dos vários grupos à zona de confluência, tentando maximizar o número de pontos de interesse visitados por trilho, sendo que cada trilho tem uma dificuldade e apenas terá uma certa tolerância a troços de dificuldade superior ou inferior.

$$f = Tc[Length(Tc)] - Tc[1]$$

Funçao que devolve a distancia temporal entre a chegada do primeiro e do ultimo grupo ao local de confluência *n*. Função deve ser minimizada de modo a que os grupos não tenham que esperar uns pelos outros para, por exemplo, almoçar.

$$g = (\sum_{i=0}^{n}(Tc[i] - HConf)/n$$

Funçao que devolve ao valor absoluto da diferença entre a media das horas de chegada dos grupos e a Hora prevista para a chegada ao local de confluência. O valor desta função deve ser minimizado de forma a garantir que os grupos cheguem aos locais perto das horas marcadas.

$$h = Length(Poi)$$

O valor desta função deve ser maximizado de modo a aumentar os pontos de interesse visitados por grupo.

$$l = \sum_{i=0}^{n}(|D[i] - ExpGrupo[i]|)/n$$

Função que minimiza a diferença entre a dificiculdade do trilho e a Experiencia do grupo que o vai realizar

Tal como referido na descrição do problema, irá ser privilegiada a minimização da função f, g e l tentando quando possível maximizar a função h

3 Perspetiva de solução

Nesta secção são apresentados os problemas encontrados ao longo das várias iterações e os principais algoritmos a serem considerados para a implementação prática deste problema. Onde for necessário, será ainda providenciado pseudo-código de modo a facilitar a compreensão dos algoritmos.

3.1 Análise da Conectividade

Para a nossa análise de conectividade, admitimos que o grafo dado não é dirigido, uma vez que os trilhos apenas serão utilizados por peões, estes podendo andar livremente em qualquer direção em todos os troços. Sendo assim, para qualquer aresta que liga um vértice X a outro Y, é sempre válido fazer tanto o percurso de X para Y como de Y para X.

Como mencionado nas restrições iniciais, podemos afirmar que todos os pontos essenciais do grafo, como os pontos de confluência, terão de pertencer todos à mesma componente fortemente conexa, visto que terá de ser possível ir de qualquer ponto útil para outro. Uma vez que estamos perante um grafo não-direcionado, a análise de conectividade torna-se mais simples , não sendo necessário recorrer a algoritmos como o de Kosaraju ou o de Tarjan. Basta optar por um algoritmo de BFS ou DFS, similares ao pseudo-código apresentado abaixo, fazendo corresponder a um conjunto de vértices a sua componente fortemente conexa.

Componenetes conexas das quais não constasse nenhum ponto de interesse poderão ser removidas do grafo.

Este procedimento permitirá verificar quais são os pontos alcançáveis a partir do ponto de partida, sendo apenas analisada a componente fortemente conexa onde o ponto de partida se inclui, acelerando assim o tempo de execução.

Durante este processamento serão também calculadas as dificuldades de cada troço e eliminados todos os pontos que estão inacessíveis, quer por dificuldade demasiado elevada quer por outras razões.

Será então apresentada uma breve descrição dos dois algoritmos que nos parecem mais pertinentes para esta análise.

3.1.1 Pesquisa em Profundidade (DFS)

Na pesquisa em profundidade, as arestas são exploradas a partir do último vértice a ser explorado, tanto quanto possível até ter de ocorrer backtracking, sendo um algoritmo de pesquisa cega. Este algoritmo tem uma complexidade temporal de $O(|V| + |E|)$ e uma complexidade espacial de $O|V|$.

Algorithm 1 DFS

```
1:  function DFS(G)
2:      n_1 = q − p + 1
3:      n_2 = r − q
4:      for v ∈ V do
5:          visited(v) = False
6:      for v ∈ V do
7:          if not visited(v) then
8:              VisitDFS(G, v)
9:  function VisitDFS(G, v)
10:     visited(v) = True
11:     for w ∈ Adj(v) do
12:         visited(v) = False
13:         if not visited(w) then
14:             VisitDFS(G, w)
```

3.1.2 Algoritmo de Pesquisa em Largura

Nesta pesquisa, explora-se primeiramente todos os vértices a que se pode chegar a partir do nó atual, só passando para os filhos destes após se ter examinado todos estes vértices. A sua complexidade temporal é de O (|V | + |E|) e a complexidade espacial de O(|V |).

O Pseudo código para ambos os algoritmos é apresentado nas figuras abaixo.

Algorithm 2 BFS

```
1:  function BFS(G)
2:      for v ∈ V do
3:          visited(v) = False
4:      Q = 0
5:      push(Q, s)
6:      visited(s) = True
7:      while Q ≠ 0 do
8:          for w ∈ Adj(v) do
9:              if not visited(w) then
10:                 push(Q, w)
11:                 visited(w) = True
```

3.2 Pré-Processamento dos Dados de Entrada

3.2.1 Pré-processamento do grafo

Previamente às iterações, o grafo Gi deve ser pré-processado, reduzindo o seu número de vértices e arestas, a fim de aumentar a eficiência temporal dos algoritmos nele aplicados. Tal como referido na análise de conectividade, serão ignoradas todas as arestas do grafo que não pertençam ao mesmo grafo conexo do ponto de partida, e serão calculadas as dificuldades de cada troço do grafo resultante, sendo assim possível ignorar os trilhos que tenham uma dificuldade demasiado elevada.

Existem diversas maneiras de calcular as distancias dos vértices uns aos outros, poupando bastantes cálculos em runtime, sendo as alternativas mais viáveis apresentadas de seguida. É de referir que apesar de todos estes algoritmos terem uma implementação bastante pesada, apenas são corridos uma vez, sendo vantajoso a longo prazo para os utilizadores.

Algoritmo de Floyd Warshall

Este algoritmo de programação dinâmica é utilizado com o objetivo de calcular as distâncias mínimas entre todos os pares de vértices num grafo, retornando uma matriz que contém as distâncias mínimas entre todos os vértices no mapa.

O algoritmo toma proveito de uma matriz de adjacência $D[i][j]$, que irá conter as distâncias mínimas entre quaisquer dois vértices i e j, que é inicializada segundo os pesos ∞ se i != j ou 0 se i = j. São então encadeados 3 ciclos que percorrem a totalidade dos vértices usando 3 variáveis i, j, k, tal que, em cada iteração, o valor da matriz em $D[i][j]$ é atualizado caso

$$D[i][j] > D[i][k] + D[k][j]$$

ou seja, caso o percurso mais eficiente de momento entre i e j tenha um peso maior ao percurso de i a k seguido do percurso de k a j, este último passa a ser o percurso mais eficiente. Por outras palavras, se um percurso que passe pelo vértice intermédio k é menor que o melhor até agora encontrado, esse percurso passa a ser o ótimo até ser, possivelmente, substituído por outro mais melhor.

A sua complexidade temporal é de $O(|V|^3)$, sendo preferível em grafos densos em que o número de arestas e da ordem do quadrado do número de vértices. A densidade de um grafo é calculado utilizando a seguinte fórmula,

$$D = |E|/(|V| * (|V| - 1))$$

variando entre 0, valor mínimo e 1, máximo.

Esta opção acabou por ser a escolhida visto que calcula a distância de todos os pontos, para todos os outros, sendo vantajoso no caso em que não existe qualquer ponto fixo.

A aplicação de um destes métodos, aumentaria a performance da restante parte do programa. Utilizando a menor distância real entre dois pontos em vez da distância estimada ajuda a que a heurística guie melhor o algoritmo. Apesar de pesados, estes métodos são apenas executados uma vez para cada grafo, trazendo um benefício a cada execução do programa.

Algorithm 3 Floyd Warshall

```
1:  function FloydWarshall(G)
2:      for i = 1 to |V| do
3:          for j = 1 to |V| do
4:              if exists edge from i to j then
5:                  D[0, i, j] = W[i, j]
6:              else
7:                  D[0, i, j] = ∞
8:      for k = 1 to |V| do
9:          for i = 1 to |V| do
10:             for j = 1 to |V| do
11:                 D[k, i, j] = min(D[k − 1, i, j], D[k − 1, i, k] + D[k − 1, k, j])
```

Algoritmo de Dijkstra

O algoritmo de *Dijkstra*, concebido por Edsger Dijkstra em 1956 e publicado em 1959, soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo. Este método tem uma complexidade temporal de

$$O((|V| + |E|) * log(|V|))$$

Devido ao contexto do problema, em que é necessário passar por vértices específicos, o algoritmo teria de ser aplicado sucessivamente, para cada ponto de interesse, tendo neste caso em específico uma complexidade de

$$O(|P| * (|V| + |E|) * log(|V|))$$

sendo $|P|$ o número de pontos de interesse.

Comparando com o algoritmo de Floyd Warshall torna-se vantajoso perante grafos esparsos e tem a vantagem de apenas calcular distâncias para pontos selecionados, o que pode melhorar bastante o tempo de execução caso o rácio de pontos de interesse por vértices totais do grafo seja reduzido, não obrigando ao cálculo de distâncias para todos os pares de pontos. Consequentemente, este deixa de ser aplicado $|V|$ vezes mas sim $|P|$ vezes, sendo P o número de pontos de interesse considerados.

No entanto, como já referido anteriormente, foi decidido que não se teria pontos fixos, sendo que qualquer ponto poderia ser escolhido em runtime. Desta maneira, o algoritmo de Floyd Warshall acabou por ser uma melhor opção por calcular a distância de todos os pontos para todos os outros.

3.3 Pré processamento de pontos Default

De modo a poupar trabalho ao utilizador, foram ainda pré cálculados alguns pontos de confluencia interessantes, de modo a mostrar a capacidade do nosso algoritmo. De modo a encontrar bons pontos, usou-se o pré processamento da conetividade, procurando o grafo com mais pontos, de modo a apenas escolher vértices nas componentes mais fortemente conexas do grafo. De seguida, escolheram-se vários pontos bastante afastados, assim como alguns pontos de cônfluencia, distâncias e dificuldades.

3.3.1 Pré-Processamento associado ao cálculo de dificuldades

Assumindo que cada nó tem uma altura, tendo esta sido generada ou já pertencendo ao grafo, será necessário fazer um pré processamento que passe por todas as arestas de modo a calcular o declive de cada uma e associar-lhe uma dificuldade em função deste último.

O declive será obtido dividindo a diferença de altura dos dois vértices pela distância real percorrida pelo utilizador no respetivo percurso, esta última correspondendo à hipotenusa:

$$Distancia = \sqrt{((|Xf - Xi|)^2 + (|Yf - Yi|)^2)}$$
$$Declive = ((Yf - Yi)/Distancia) * 10$$

Visto que tanto descidas como subidas muito abruptas exigem um grau de experiência maior por parte do utilizador, a dificuldade será tão mais alta quão mais longe esta estiver de ser plano. No entanto, apesar de o declive variar entre -1 e 1, como os trilhos com um declive médio de 50% para cima já são bastante impraticáveis , serão considerados inacessíveis, fazendo com que o declive dos percursos apenas varie entre -0.5 e 0.5.

A dificuldade será então calculada da seguinte forma:

$$Dificuldade = \lceil 20 * |declive| \rceil$$

Sendo esta igual a 0 quando se está num plano e de 10 num trilho com uma inclinação de 50%.

No entanto, visto que os grafos dados não continham alturas, optou-se por atribuir de uma maneira pseudo-aleatórea as dificuldades, favorecendo as mais baixas, visto que uma pessoa com um maior grau de expêriencia consegue facilmente realizar um percurso mais simples, enquanto que o contrário já não é possível. O algoritmo calcula um número aleatóreo entre 1 e 10, sendo que este tem 50% de hipótese de ser recalculado entre 1 e 7, e este recalculado entre 1 e 4.

3.4 Identificação dos problemas encontrados

Analisando o problema de forma superficial, reparamos que o podemos aproximar do ***Travelling Salesman Problem*** uma vez que o objetivo principal será encontrar o melhor caminho entre vários pontos. No entanto, o nosso problema apresenta várias variações do problema original, como o facto de o melhor caminho não o menor caminho, mas sim o caminho com uma duração exata.

Surge ainda a questão de que a passagem pelos pontos deverá ser feita dentro de um intervalo temporal, segundo a sequência definida. Desta forma o problema assemelha-se mais com o problema ***Vehicle Routing with Time-Windows.***

O ***Tourist Trip Design Problem*** é um problema em que o objetivo é escolher os caminhos que maximizem o número de pontos de interesse visitados, tendo em conta um conjunto de restrições e parâmetros, como o tempo demorado ou por exemplo o custo do ponto de interesse. A nossa última iteração é bastante semelhante a este problema, tendo sido examinado cuidadosamente de modo a descobrir a melhor forma de o abordar.

3.5 Considerações Globais

Tanto o Dijkstra como o A* têm uma complexidade, no pior dos casos, de b^d. Apesar disso o tempo normal de execução do Dijkstra ronda esse mesmo

valor, enquanto que o A* permite que este valor seja reduzido drasticamente, dependendo da qualidade da heurística. Torna-se difícil analisar a complexidade do A*, porém este é regra geral mais rápido que o Dijkstra.

Tanto no A* como no anytime A*, é possivel facilmente alterar a heuristica de modo a que encontre um caminho para cada grau de dificuldade, sendo que esse maximiza o numero de pontos de interesse encontrados. Quanto ao dikstra e ao bidirecttional dijkstra, desde que os caminhos estejam dentro das restrições, como estar no local certo na hora de confluência e cumprir os requesitos de dificuldade, deve se aceitar vários caminhos, escolhendo depois aquele que maximize o numero de pontos de interesse visitados.

Visto que não estamos a tratar de um problema de caminho mais curto, nada impede que caso o grupo tenha tempo de chegar ao destino, volte a visitar vezes e vezes sem conta o mesmo troço. De maneira a contornar esta dificuldade, pode definir-se uma lista de edges já visitadas de modo a que isto não aconteça.

De forma a obter uma melhoria de performance, e uma vez que não estamos focados numa solução ótima, pode ser adicionado um fator de relaxação à heurística, de forma a tornar o algoritmo mais rápido, abdicando de encontrar a resposta ótima. Se o fator for de 20%, e se considerarmos a existência de um resultado ótimo, o resultado poderá ser até 20% maior que o resultado considerado ótimo, no entanto a performance é melhorada. A heurística obtida não permite obter o valor ótimo pois não é garantida a consistência nem a admissibilidade, no entanto, como já foi dito anteriormente, não se está atrás de um resultado ótimo, não sendo assim um problema significativo.

4 Estratégia usada e Algoritmos implmentados

Tendo em conta as funções objetivo descritas anteriormente, decidiu-se que a melhor forma de maximizar as funções f,g seria aproximar o mais possível o valor do tempo de um caminho ao pretendido. Se todos os grupos tiverem tempos próximos do pretendido então a média dos valores de chegada estará proxima do pretendido. De forma análoga,a diferença entre o primeiro e último será diminuida usando esta premissa. Depois de analisados os problemas encontrados decidimos então subdividir o problema, calculando o melhor caminho entre cada par de pontos, uma vez que a sequência dos pontos de confluência está já definida. Por exemplo, num caso hipotético onde são dados 3 pontos de confluência como dados de entrada, tendo cada um uma hora associada:

1. Início - 10:00
2. Almoço - 13:00
3. Fim - 17:00

Podemos calcular de forma independente os melhores caminhos entre o início e fim, sendo depois calculados os trilhos entre o almoço e o fim.

Para resolver os problemas em tempo polinomial, decidiu-se adotar uma abordagem Gananciosa, apesar de, no pior dos casos poder ter complexidade exponencial, permite resolver problemas NP em média em tempos polinomiais, podendo porém por vezes obter um resultado não ótimo. Ainda assim a natureza do problema não permite definir com clareza o ótimo, por ser uma conjugação de tempo, número de POI's e diferentes Dificuldades, por isso a existência de resultado ótimo não seria tomado em conta, independentemente da abordagem.

Foi também usada a noção de Divisão e Conquista pois os problemas são divididos em sub-problemas menores e cada um é processado de forma independente.

Para resolver estes sub-problemas surgiram duas possíveis abordagens. A primeira consiste no pré-processamento dos dados, a cada execução e posterior utilização do algoritmo Dijkstra. A segunda consiste em utilizar uma heurística para guiar o algoritmo na direção certa, usando o algoritmo A*.

Ao adicionar a noção de dificuldade de cada caminho surgiu o problema de encontrar um algoritmo que devolvesse caminhos com dificuldades correspondentes às interessadas. Decidiu-se então para cada par de pontos, calcular um caminho para cada dificuldade recebida nos dados de entrada. Uma vez que as dificuldades foram discretizadas para valores Naturais entre 1 e 10, cada subcaminho pode apresentar no máximo 10 níveis de dificuldades diferentes.

Considerando o número de pontos de confluência P, o número de níveis de dificuldade D, se o cálculo de um caminho para cada dificuldade em cada subcaminho apresentar uma complexidade temporal de $f(n)$, a complexidade temporal média do cálculo de todos os trilhos completos, desde o início até ao fim, passando pelos pontos de confluência será de $P * D * f(n)$. Sendo P e D valores pequenos a complexidade temporal total do algoritmo depende, na sua maioria, da complexidade do cálculo de cada sub-caminho.

4.1 A* Com heurística adaptada

A abordagem mais fiável para resolver os subproblemas implementada foi utilizar o algoritmo A* com uma heurística alterada. É necessário considerar que a rapidez e a obtenção de um valor da heurística correto dependem da forma como o tempo entre os pontos é calculado. Caso seja pré processada o tempo entre os POI's e todos os pontos, a heurística utilizará este valor, conseguindo uma performance superior. Caso estes tempos não sejam calculados, a forma de calcular será a o tempo estimado em linha reta entre o ponto e o destino. Visto que o Floyd Warshall é muito pesado, apenas foi cálculado em certos mapas mais pequenos, sendo este algoritmo mais preciso nesses mapas.

O A* permite também atribuir importância, quando se aplica, à passagem por pontos de interesse e à dificuldade, em contraste com o dijkstra modificado, apresentado no relatório anterior. Pela natureza da heurística, os pontos demasiado afastados nunca serão tidos em conta, não sendo necessário fazer a limpeza do grafo para cada par de Pontos de Confluência.

Melhor caminho entre dois Pontos:

Numa primeira fase, o cálculo do melhor caminho apenas depende de quanto é que este demora de um ponto de confluência ao seguinte, sendo este aquele que minimiza a diferença entre a hora marcada e a hora de chegada. A função que queremos minimizar é

$$f(T, TP) = |T - TP|$$

com T o valor obtido e TP o valor pretendido.

Em cada iteração pretendemos que a soma dos tempos até ao ponto somada com a estimativa até ao ponto final pretendido seja o mais próximo possível do tempo pretendido. sendo assim o peso de cada nó será

$$f() = |(C + E) - TP|$$

onde C é o custo real até ao nó, E é o custo estimado do nó até ao ponto final e TP o tempo pretendido para o sub-caminho.

Melhor caminho considerando a dificuldade:

Adicionar a dificuldade de cada caminho, trata-se apenas de ajustar a heurística para o cálculo do melhor caminho, de forma a que esta integre a dificuldade no tempo. Assumindo que a partição do sub-caminho para que exista uma pesquisa por nível de dificuldade, à função a minimizar adicionamos $|D - DP|$, com D a dificuldade do edge que estamos a analisar e DP o nível de dificuldade. Neste caso apenas serão considerados edges que tenham uma dificuldade D, com $0 \leq D \leq (DP - 1)$. a função do peso passa a ser

$$f() = |(C + E) - TP| + |DP - D|$$

Sendo que cada valor será multiplicado por uma percentagem, de modo a obter a melhor relação possível entre a dificuldade dos caminhos e o tempo exato deste.

Melhor caminho Considerando os pontos de interesse:

De forma a tornar os caminhos mais interessantes, é oportuno adicionar à heurística um parâmetro que escolha os nós com mais POI's. Para isto, utilizando uma aproximação Gananciosa, será adicionado o parâmetro POI que tomará o valor 0 se um edge der a um POI e 1 se o edge não for para um. A este valor deve ser adicionado um fator de padronização, uma vez que sendo o valor 1 ou 0, este pouco influênciaria quando somado com o custo até ao nó. Assim sendo o parâmetro seria $POI * U$. Adicionando à heurística já apresentada, ficaria

$$f() = |(C + E) - TP| + |DP - D| + POI * U$$

Os valores $|(C + E) - TP|, |DP - D|$ e $POI * U$ podem ser multiplicados por um fator de importância. Cada fator representa um valor de 0 a 1 e a soma dos fatores deve ser 1. Desta forma é possível alterar a importância de cada elemento da heurística no cálculo do melhor caminho. Estes valores Serão testados na parte pŕática pois tratam-se de valores dificilmente atribuíveis sem testes.

Depois da análise empírica dos algoritmos acima descritos pode se chegar à conclusão que o Dijksta é um algoritmo lento, o que pode inviabilizar a sua utilização.

Algorithm 4 A*

```
1: function reconstruct_path(cameFrom, current)(G)
2:     total_path = current
3:     while current ∈ cameFrom.Keys do
4:         current = cameFrom[current]
5:         total_path.prepend(current)
6:     returntotal_path
   ▷ openSet é uma fila de prioridade ordenada pelo peso até ao ponto
   ▷ openSet.top é o elemento da fila com menor valor de peso
   ▷ para cada nó é guardada a distância até a esse ponto e o peso com heurística, pelo qual é ordenada a lista
   ▷ h é a heurística adotada
7: function A_Star(start, goal, h, tempoPretendido, dificPretendida)(G)
8:     openSet = todos os nós com peso ∞, start com peso 0, e todas as distancias a 0
9:     cameFrom = emptyMap
10:    while openSet != ∅ do
11:        current = openSet.top
12:        if current = goal then
13:            return reconstructPath(cameFrom, current)
14:        openSet.pop
15:        for each n in current.adjacentV do
16:            d = current.distancia
17:            d+ = distancia(current, v)
18:            tentative_gScore = h(d, e, tempoPretendido, dificPretendida)
19:            if dificPretendida + 1 >= dificuldade(current, n) then
20:                if tentative_gScore < n.peso then
21:                    cameFrom[n] = current
22:                    decreaseKey(n, tentative_gScore)
23:                    n.distancia = d
24:    return 'erro'
```

4.2 Divisão em sub-Grupos

De modo a integrar os pontos de confluência *(x,y)*, optou-se por dividir o problema em sub-Grupos, onde para cada dois pontos de confluência *(x,y)* e para cada dificuldade pretendida, é calculado o melhor caminho usando o A* referido anteriormente *u,x, x,y e y,v* para cada dificuldade.

Algorithm 5 CalculateInterestingPath*

```
1: function calculateInterestingPath(confluencePoints, hours, difficulties)(G)
2:     if SIZE(confluencePoints) ≠ SIZE(hours) then return false
3:     for dinSIZE(difficulties) do
4:         for i ∈ SIZE(confluencePoints) − 1 do
5:             c = confluencePoints[i]
6:             c1 = confluencePoints[i + 1]
7:             h = hours[i]
8:             h1 = hours[i]
9:             AStar(c, c1, h1 − h, difficulties[d], nodes)
10:            if !foundPath then                ▷ AStar without visited points
11:                AStar(c, c1, h1 − h, difficulties[d])
       return True
```

4.3 Pontos de interesse obrigatórios:

A forma de incluir pontos de interesse obrigatórios, será calcular um caminho que passe por todos os pontos, seguindo a ordem definida para os pontos de confluência. Para resolver este problema podemos considerá-lo como do tipo **Caixeiro-Viajante** onde encontramos o melhor caminho para percorrer todos os pontos, considerando o ótimo o menor caminho. No entanto, como todos os grupos têm de estar nos pontos de confluência à hora marcada, mesmo sendo dada uma ordem que geograficamente faça sentido, temos que garantir que continua ser possivel chegar não o mais rapidamente possivel ao destino, mas sim a horas.

Por exemplo, imaginando que o algoritmo nos dava a ordem:

1. A - Ponto de Confluência 12:00
2. B - Ponto de interesse
3. C - Ponto de confluencia. 15:00

Neste caso, seria estimado o tempo entre A-B e B-C. Se a soma destes dois tempos fosse menor do que o tempo necessário para chegar a C à hora marcada, seria então realizada uma ponderação de quanto tempo se deveria tentar demorar entre A-B e B-C, de modo a que ao percorrer esse caminho o utilizador chegasse à hora exata â localização destino, cumprindo a dificuldade proposta e até, tentando maximizar os pontos de interesse caso possivel. Caso fosse maior que o tempo necessário para chegar a C, então descartaria-se a hipótese de passar por todos os pontos de interesse obrigatórios.

4.4 Algoritmo do vizinho mais próximo

Trata-se de uma heurística de fácil implementação, resolvendo em tempo aproximadamente linear. Consiste em, a partir do vértice inicial, tomar sempre o vértice mais próximo que ainda não foi visitado até atingir o vértice final. Tendo em conta o nosso problema específico, caso o ponto mais próximo seja um ponto de confluência, este só é tido em conta caso o ponto de conflência anterior tenha sido explorado. Este algoritmo não encontra necessariamente a melhor solução para o problema, no entanto é mais rápido que qualquer outro algoritmo para soluções exatas. A sua implementação em pseudocódigo é a seguinte:

Algorithm 6 Algoritmo Vizinho mais próximo

```
1:  function NearestNeighbour(G, V, current, C)
    ▷ //C é uma fila dos pontos confluencia por ordem
    ▷ //V é uma lista com os ponntos obrigatórios
2:      while C != ∅ do
3:          s = min_dist(current)
4:          while (s ∈ C AND s != C.top()) do
5:              s = next_min_dist(current)
6:          while (s ∈ C AND C.size = 1 AND V.size>0) do
7:              s = next_min_dist(current)
8:          path(current) = s
9:          current = s
10:         if s = C.top() then
11:             popC
12:         else
13:             removersdeV
```

5 Casos de utilização Implementados

1. **Visualização do grafo completo** - O programa fornece a possibilidade de visualizar graficamente um grafo completo, com todos os seus nós e arestas.

2. **Visualização da Componente fortemente conexa do mapa** - O programa fornece a possibilidade de visualizar graficamente os nós e arestas da componente mais fortemente conexa do grafo.

3. **Cálculo do caminho mais adequado ao grupo escolhido** - Esta é a funcionalidade principal do programa, em que o programa dado um conjunto de restrições calcula os percursos mais adequados para o grupo.

4. **Cálculo da distância total de um trilho** - O programa deverá ser capaz de calcular a distância total percorrida por um determinado grupo

5. **Cálculo de caminhos de um ponto a outro com uma certa distância**

6. **Cálculo de caminhos de um ponto a outro com uma certa distância e dificuldade**

7. **Cálculo do caminho com POI obrigatórios** - Opção de introduzir pontos de interesse obrigatórios.

8. **Cálculo do caminho que maximize POI's** - Opção que maximize os pontos de interesse entre um ponto e outro, com um certo tempo de duração, independentemente da dificuldade.

9. **Inserção e remoção de pontos de Confluência e de pontos de interesse obrigatórios**

10. **Escolha entre várias cidades, sendo facilmente adicionada qualquer outra**

11. **Opção de ver pontos default de modo a facilitar a escolha dos pontos de confluência**

12. **Pré-Processamento de cada cidade, incluindo dificuldades, floyd warshall e conetividade**

6 Estruturas de Dados Utilizadas

Foram tidas em conta, na seleção das estruturas de dados a utilizar, a complexidade temporal e espacial. Foi dada maior importância à complexidade temporal em detrimento do espaço ocupado. Considerando que foi necessário carregar os nós e posteriormente, ao carregar os edges, encontrar os nós anteriormente carregados, interessava uma estrutura de dados que permitisse um tempo de acesso constante a um nó com um id específico, utilizando para isso a noção de key.

Tendo isto em conta, foi utilizada a unordered_map da stl, que implementa uma hash_table permitindo acesso, inserção e remoção em tempo constante, para guardar os nós de cada grafo. Cada nó guarda os edges com origem nesse mesmo nó. Desta forma foi possível fazer o carregamento dos edges e dos nós em tempo linear com o número de entradas.

Em várias partes do projeto foi usado o vector da stl pela sua simplicidade de utilização, velocidade de iteração, e principalmente pela sua complexidade espacial linear.

6.1 Representação do Grafo

6.1.1 Grafo

O grafo é representado através da classe Graph, definida em Graph.h. É constituido por dois std::unordered_map, um para os nós e outro para as arestas presentes no grafo (**std::unordered_map<long,Node*>**, **std::unordered_map<long,Edge*>**) contendo assim informação de todas as suas partes constituintes e permitindo um tempo de acesso constante de O(1).

Possui ainda outros atributos que se apresentaram úteis na elaboração do programa:

1. **std::vector<Node*> nodesVector** - útil na elaboração do Algoritmo de FloydWarshall fazendo com que o índice de um Nó num vetor seja a sua posição na matriz de distâncias mínimas.

2. **std::vector<unordered_set<int> > graphs** - armazena cada subgrafo fortemente do conexo do grafo em questão, fazendo uso dos ids dos nós constituintes.

3. **std::unordered_map<int,int> edgeDiff** - utilizado no carregamento das dificuldades atribuídas a cada aresta.

4. **std::vector<vector<int> > floydMatrix** - contém as distâncias mínimas geradas pelo algoritmo de Floyd Warshall.

6.1.2 Nós

Um Nó é representado pela classe Node, definida em Node.h, tendo como principais constituintes da sua estrutura **std::vector<Edge *> edges** (conjunto de arestas que se ligam ao nó) e **std::vector<string> tags** (conjunto de tags às quais o nó se encontra associado). É identificado pelo **id** representado por uma variável long int.

6.1.3 Arestas

Uma Aresta é representada pela classe Edge, definida em Edge.h e caracteriza-se por um **peso**, dois apontadores: para o **nó destino** e o **nó origem** e um valor de **dificuldade** gerado na fase de processamento.

6.2 Carregamento de Dados

O carregamento de dados é feito através da classe definida em GraphLoader.h e é responsável por:

1. Carregar o grafo pretendido.
2. Carregar as dificuldades associadas a cada aresta.
3. Carregar cada componente fortemente conexa previamente processada.
4. Carregar as distâncias mínimas resultantes da aplicação do algoritmo de Floyd Warshall.

6.3 Representação dos mapas e caminhos encontrados

O desenho do resultado final fica ao cargo da classe definida em GraphDrawer.h que representa os vários percursos e mapas com o auxílio do GraphViewer.

6.4 Menu

Por fim de forma a ser mais fácil para o utilizador utilizar o programa, foi criada uma interface de texto, com todas as funcionalidades do programa.

7 Análise da Conectividade

Visto que o programa não tem nenhum ponto inicial fixo, de modo a dar flexibilidade ao programa, optou-se por realizar uma versão do DFS, em pré processamento, como é explicado de sguida.

7.1 Pesquisa em Profundidade (DFS)

Este algoritmo, como já foi referido anteriormente, tem uma complexidade temporal de $O(|V| + |E|)$ e uma complexidade espacial de $O|V|$. No entanto, teria de ser corrido em runtime, sendo necessário executá-lo cada vez que se quisesse remover os nós desnecessários do grafo. A solução encontrada para este problema foi alterar este algoritmo de modo a que guardasse num ficheiro, separadamente todos os grafos conexos do grafo, sendo apenas necessário, quando o programa fosse corrido, ler desse ficheiro os grafos, verificar em qual deles está o vértice inicial e apenas processar esses vértices.

Algorithm 7 preprocessConnectivity

```
1:  function preprocessConnectivity(G)
2:      vector < vector < int >> nodes
3:      G.resetVisited(v)
4:      for v ∈ V do
5:          if not visited(v) then
6:              vector < int > connectedNodes
7:              VisitDFS(v, connectedNodes)
8:              INSERT(nodes, connectedNodes)
        return nodes
9:  function VisitDFS(v, connectedNodes)
10:     visited(v) = True
11:     for w ∈ Adj(v) do
12:         if not visited(w, connectedNodes) then
13:             VisitDFS(w, connectedNodes)
14:             INSERT(connectedNodes, w)
```

7.2 Análise de resultados

Tendo como exemplo o mapa fornecido da cidade de Lisboa que possui um total de 74622 nós, após a análise de conectividade partindo de cada ponto de interesse obtiveram-se um total de 4246 subgrafos fortemente conexos. Muitos deles representam amostras não significativas uma vez que possuem poucos nós (valores entre os 10 e os 100 nós, chegando mesmo por vezes a constituir grafos com apenas 1 ou 2 nós), no entanto para o caso de Lisboa foi possível encontrar uma amostra fortemente conexa considerável com 25657 nós.

Resultados como este foram díficeis de encontrar nos restantes mapas inicialmente fornecidos, no entanto o problema foi resolvido com a adição de novos mapas para Espinho, Penafiel e Porto, para os quais os resultados obtidos já se apresentaram significativos, sendo que todos eles possuem pelo menos um subgrafo fortemente conexo com 10000 nós ou mais, chegando o caso do mapa do Porto a atingir os 45704 nós numa componente fortemente conexa.

8 Análise dos Algoritmos efetivamente implementados

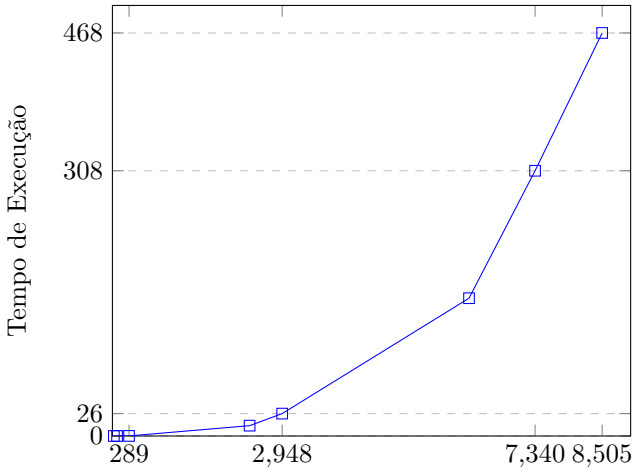
8.1 Floyd-Warshall

8.1.1 Análise Teórica

Este algoritmo apresenta uma complexidade temporal teórica de n^3(melhor e pior caso), com n o número de nós do grafo. Trata-se portanto de um algoritmo com uma complexidade temporal Polinomial, que no entanto cresce significativamente com o aumentar dos nós do grafo. Observando o nosso problema em específico não é difícil encontrar grafos com alguns milhares de nós o que pode traduzir em tempos de execução incomportáveis.

Analisando o algoritmo e a sua funcionalidade é fácil perceber que a sua complexidade espacial é de V^2 pois precisa de guardar para cada vértice V valores, correspondentes à distância de cada vértice a todos os outros.

8.1.2 Análise Empírica



Analisando os pontos e o gráfico em geral observa-se que de facto a complexidade temporal aumenta significativamente com o aumento dos vertices do grafo. Apesar de, à primeira vista poder parecer exponencial, consegue-se aproximar os pontos do gráfico a uma curva polinomial do tipo

$$ax^3 - bx^2 + cx + d$$

.

8.2 A*

8.2.1 Análise Teórica

O algoritmo A* apresenta uma performance teórica média difícil de calcular dada a existência de uma heurística que guia o processo de procura. No entanto, no pior dos casos a sua complexidade temporal e espacial assemelha-se à complexidade do algoritmo Dijkstra $O(b^d)$, com b o branching factor e a distância entre os pontos inicial e final.

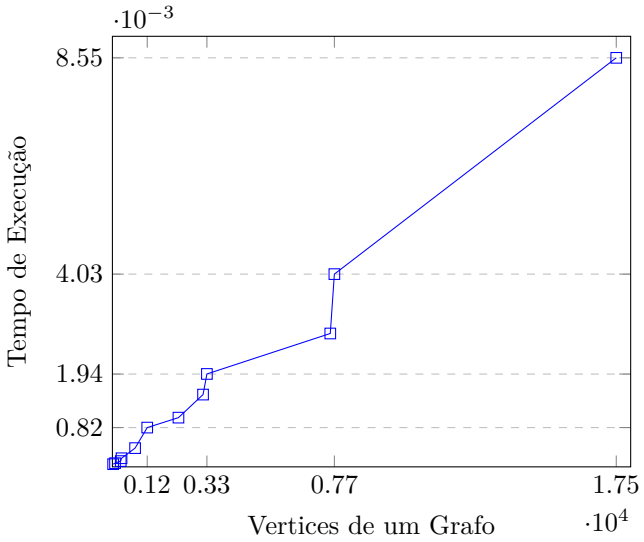
8.2.2 Aplicação no problema

Este algoritmo foi essencial no desenvolvimento deste trabalho, tendo sido a base de funcionamento deste. Contrariamente com o que foi dito nas Iterações, o trabalho foi implementado de uma maneira ligeiramente diferente, em que as primeiras três iterações, a distância, dificuldade e pontos de interesse foram programadas apenas considerando dois pontos de confluência e apenas diferindo na heurística deste algoritmo. É apresentado de seguida uma versão simplificada do algoritmo,assim como alguns exemplos das heurísticas utilizadas.

Listing 1 A*

```
1: double Graph::AStar(long int origin,long int  target,
2:                     long int targetDistance, int difficulty,
3:                     vector<Node> *nodesVisited,string AStarType){
4:
5:     initNodes(nodes[origin],nodes[target],nodesVisited);
6:     PriorityQueue q;
7:     q.insert((nodes[origin]));
8:     while( ! q.empty()){
9:         auto v = q.extractMin();
10:        v->visited = true;
11:        //Caso encontre o target e o caminho seja aceit vel
12:        if (v->getId() == nodes[target]->getId())
13:            if((abs(v->getDist()-targetDistance)/targetDistance) < 0.1)
14:                printInformation()
15:                return 0;
16:        for(auto e : v->getEdges()){
17:            auto oldDist = e->getDestination()->getDist();
18:            //Apenas tem em conta pontos n o explorados
19:            if(!e->getDestination()->visited){
20:                if(getRelaxFunction(v, e->getDestination(),
21:                                    e->getWeight(), targetDistance,
22:                                    e->getDifficulty(),
23:                                    difficulty,AStarType)){
24:                    //se a o valor do weight for inferior ao anterior
25:                    //atualiza a posi   o do node na fila
26:                    if (oldDist == INF) q.insert(e->getDestination());
27:                    else q.decreaseKey(e->getDestination());
28:                }
29:            }
30:        }
31:    }
32:
33:    return 0;
34: }
```

8.2.3 Análise Empírica



Torna-se difícil observar uma regressão para os pontos no gráfico devido á sua oscilação, no entanto ovbserva-se que estes valores podem ser aproximados de forma satisfatória uma curva do tipo

$$ax$$

tratando-se portanto, analisando os resultados empíricos, de um algoritmos linear. Ainda assim deve-se notar que os pontos apresentam valores reduzidos(da ordem dos milissegundos) o que implica que estes não sejam muito fidedignos uma vez que estão sujeitos a muita interferência de atrasos introduzidos pelo sistema operativo e pelo computador em sí.

8.2.4 Primeira Iteração

Tal como referido anteriormente, as três primeiras iterações apenas variavam na heurística, sendo a primeira, que dado dois pontos de confluência calcula o caminho que dê o tempo/distancia mais aproximado ao valor dado, calculada usando a seguinte heurística.

Listing 2 Relax Distance

```
1: bool Graph::relaxDistance(Node *v,Node *w, double tam_edge,
2:                           long int targetDistance,
3:                           int edge_difficulty){
4:     localWeight = abs(v->getDist() + tam_edge
5:                       + w->getDistTarget()
6:                       - targetDistance);
7:     if((localWeight < w->getWeight()) && v->path != w)
8:         w->setDist( v->getDist()+tam_edge);
9:         w->setWeight(localWeight);
10:        w->path = v;
11:        w->setSummedDifficulties(
12:                (v->getSummedDifficulties() +
13:                edge_difficulty*tam_edge));
14:        return true;
15:    return false;
16: }
```

8.2.5 Segunda e Terceira Iteração

Listing 3 relax

```
1: bool Graph::relax(Node *v,Node *w, double tam_edge,
2:                     long int targetDistance, int edge_difficulty,
3:                     int difficulty,bool withPoi){
4:     double ave_diff =(v->getSummedDifficulties()+
5:                     edge_difficulty*tam_edge)/
6:                     (v->getDist()+tam_edge);
7:     float medDiff = abs(float((ave_diff-difficulty) / ave_diff));
8:     double medDist =abs(v->getDist() + tam_edge
9:                     + w->getDistTarget() - targetDistance)
10:                    / targetDistance;
11:    //Soma as partes da heuristica e divide-as conforme a import ncia
12:    double localWeight = 0.9 * medDist +  0.1 * medDiff;
13:    //se o n   w n o tive um POI aumenta o weight em 1.
14:    if(!w->getTags().size()) localWeight++;
15:
16:    //Se a dificuldade media variar entre +- 2
17:    if(abs(edge_difficulty)<=difficulty+2){
18:        if(nodeUpdate(localWeight,w,v,tam_edge,edge_difficulty,false))
19:            return true;
20:    }
21:        //Ele adiciona mais valor a dificuldade
22:        //Se a dificuldade for 5 ent o varia entre 0 e 9
23:    else if(abs(edge_difficulty-difficulty) <= 4
24:            || edge_difficulty-difficulty < 0)
25:    {
26:        localWeight = 1.05*(0.9* medDist + medDiff);
27:        if(!w->getTags().size() && withPoi) localWeight += 1.05;
28:        if(nodeUpdate(localWeight,w,v,tam_edge,edge_difficulty,true))
29:            return true;
30:    }
31:    else { //Ele adiciona mais valor a dificuldade
32:        localWeight = 1.2*(0.9*medDist + medDiff);
33:        if(!w->getTags().size() && withPoi) localWeight += 1.2;
34:        if(nodeUpdate(localWeight,w,v,tam_edge,edge_difficulty, true))
35:            return true;
36:    }
37:    return false;
38: }
```

8.3 A* Para vários pontos de confluência

De modo a cumprir com os requesitos, foi necessário criar não só um algoritmo que resolvesse os problemas indicados anteriormente para dois pontos de confluência, mas para vários. Desta maneira, visto que já sabíamos a ordem pretendida dos pontos de confluência, chegou-se à conclusão que a melhor maneira de implementar isto seria fazendo um uso repetido do A*, para cada dois pontos de confluência.

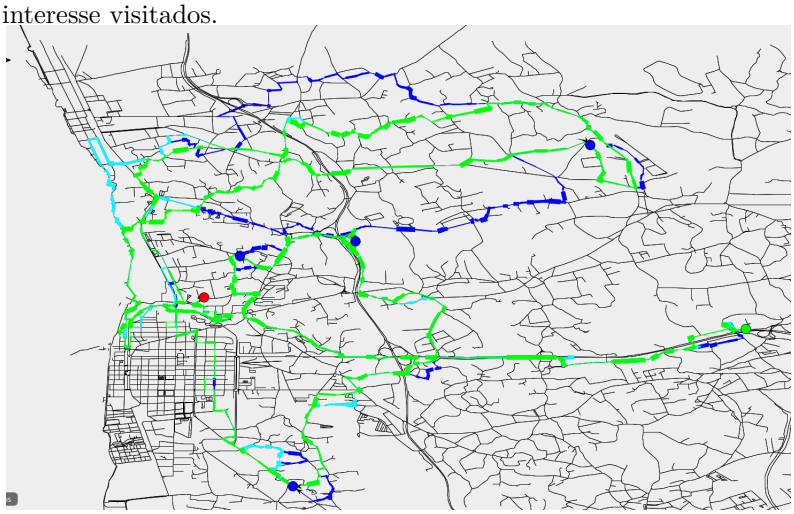
Listing 4 A* Confluence Point

```
1: bool Graph::calculateInterestingPath(
2:                                      vector<int> confluencePoints,
3:                                      vector<int> hours,
4:                                      vector<int> difficulties,
5:                                      int TMax
6:                                      )
7:     {
8:     if(confluencePoints.size() != hours.size()){
9:         Print(Each point does not have a corresponding hour)
10:        return false;
11:    }
12:
13:    for(int d = 0;d<difficulties.size();d++) {
14:        for (int i = 0; i < confluencePoints.size() - 1; ++i) {
15:            vector<Node> nodes;
16:            //preenche o vetor nodes com os pontos
17:            // j   visitados pelo grupo atual
18:            for(int i1=d*(confluencePoints.size()-1);
19:                i1<d*(confluencePoints.size()-1)+i;
20:                i1++)
21:            {
22:                nodes.insert(nodes.end()
23:                             ,pointsToDraw.at(i1).begin(),
24:                             pointsToDraw.at(i1).end());
25:            }
26:            AStar(confluencePoints[i],
27:                    confluencePoints[i + 1],
28:                    hours[i + 1] - hours[i],
29:                    difficulties.at(d),&nodes);
30:
31:            //se n o for possivel reconstruir o caminho,
32:            //recalcula-se o caminho, sem ter em conta
33:            //os pontos j  visitados
34:            if(!pointsToDraw.back().size()){
35:                pointsToDraw.pop_back();
36:                AStar(confluencePoints[i], confluencePoints[i + 1],
37:                      hours[i + 1] - hours[i], difficulties.at(d));
38:            }
39:        }
40:    }
41:    return true;
42: }
```

8.3.1 Resultado obtido

Na imagem apresentada podemos observar o resultado da aplicação do algoritmo. O ponto inicial encontra-se marcado a verde, o final a vermelho e os pontos de confluência a azul escuro.

Neste caso é possível observar três caminhos distintos associados a uma dificuldade de grupo, cada um com cor característica, resultado da melhor combinação encontrada entre distância pretendida, dificuldade e quantidade de pontos de interesse visitados.



8.4 Pontos de Interesse Obrigatórios

De forma a implementar a 4ª iteração foi necessário resolver o problema do caixeiro viajante adaptado ao nosso problema específico. Para tal foi usado o algoritmo do vizinho mais próximo, modificado para que a ordem dos Pontos de Confluência fosse mantida e para que nenhum ponto de Interesse fosse visitado depois do último ponto de Confluência. Depois de calculado o trajeto mediu-se as distância estimadas entre cada par de pontos da sequência e estimou-se a distância a percorrer.

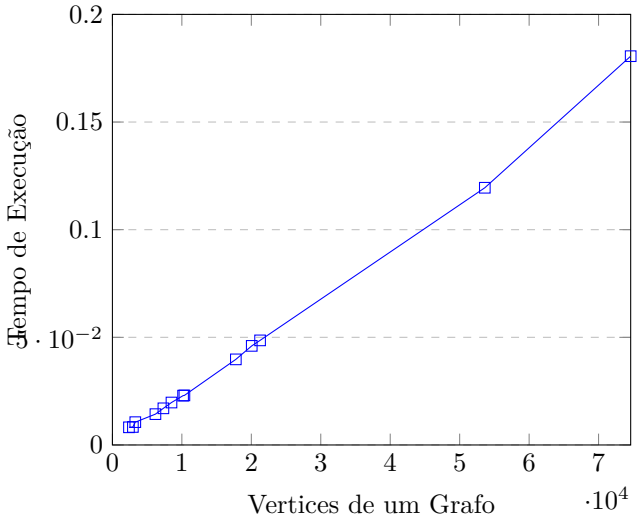
Consideremos os Pontos de Conflência A,B,C e os pontos de interesse d,e,f. Considerando que o algoritmo deolve a sequência A,d,B,e,f,C. Calcula-se as distâncias entre dois pontos consecutivos, A->d;d->B, divide-se a distância pretendida entre A e B e faz-se uma interpolação linear. Feito isto basta correr o A* para cada par consecutivo da sequência, com uma distância pretendida igual à distância interpolada.

8.5 DFS

8.5.1 Análise Teórica

O algoritmo DFS apresenta uma complexidade temporal linear, com o pior caso O(|V|+|E|) uma vez que deve explorar todos os pontos do grafo. Trata-se portanto de um algoritmo com complexidade baixa, sendo muito útil para verificar a conectividade em tempo real. A sua complexidade espacial é de, no pior dos casos, O(V). O dfs guarda os pontos não explorados ligados a pontos já explorados. O pior dos casos representa portanto o caso onde o se inicia num nó que esteja ligado diretamente a todos os outros nós do grafo. Desta forma todos os nós são guardados numa fila, ocupando o espaço V;

8.5.2 Análise Empírica



Observando os pontos e o gráfico, torna-se obvio que se trata de um algoritmo com complexidade linear, tal como descrito teoricamente, podendo ser aproximado por uma curva do tipo

$$ax$$

9 Observações em relação à Implementação

Aquando da implementação dos algoritmos, decidiu-se que se usaria a distância em vez do tempo, como descrito anteriormente. Esta decisão foi tomada pois tornou a implementação menos complexa e facilitou a interpretação dos resultados.

10 Conclusão

Dado o problema inicial, decidimos em várias etapas de implementação, as 4 iterações consideradas no início do relatório, aumentando o nível de complexidade a cada iteração.

De forma a aumentar a velocidade de processamento em cada execução, decidiu-se analisar as formas de pré-processar os dados, especialmente o grafo de entrada. Este também tinha o objetivo eliminar pontos não úteis para o problema, como pontos sem ligação a Pontos de Interesse. Foi ainda calculada o tempo mínimo entre os pontos, de forma a usar uma estimativa na heurística mais próxima à do valor real. Para isto foi tomada em consideração os algoritmos de Floyd-Warshall e o algoritmo Dijkstra.

Na implementação do algoritmo em si, usámos a abordagem de divisão e conquista de modo a tornar os problemas mais fáceis de resolver e de implementar. Para a resolução de cada caso base, discutimos a implementação de dois algoritmos, o A* baseado numa heurística, mais rápido a executar, e o Dijkstra, que tem a vantagem de ser mais fácil de implementar, ambos variações dos algoritmos originais de modo a se adaptar ao problema. Foram ainda considerados algoritmos de aproximação de forma a poder obter resultados aceitáveis em tempo real como a divisão do subcaminho em subcaminhos, que apesar de não devolver o caminho ótimo, devolve caminhos sub-ótimos num tempo consideravelmente melhor. Um exemplo é o Anytime A*, que permite que o algoritmo seja parado a qualquer momento, não devolvendo a melhor solução, mas sim uma aproximada num tempo razoável.

Começámos por considerar que o nosso problema se assemelhava a um problema do Caixeiro Viajante, verificámos que por causa da ordenação obrigatório dos Pontos de Confluência, este não era o caso. O problema pode, de facto ser associado ao problema do Vehicle Routing Problem with Window Time e ao problema do Touristic Trip Planning com algumas variações ao algoritmo original.No entanto, caso se realize a última iteração, esta passa a ser uma variação do problema do Caixeiro Viajante para cada subcaminho.

Os conhecimentos adquiridos na unidade curricular até ao momento foram essenciais na realização deste trabalho, tendo sido utilizados muitos conhecimentos adquiridos nas aulas na realização deste. Foram utilizados conceitos como programação dinâmica, bruteforce, algoritmos de divisão e conquista, algoritmos gananciosos, de retrocesso e muitos mais.

Concluindo, pensamos que conseguimos realizar uma boa abordagem ao problema, tendo ultrapassado todos os problemas propostos pelo enunciado. O trabalho foi dividido igualmente por todos os membros do grupo, sendo que este foi um resultado de discussão constante entre todos.

Referências

[1] Hanse, A. Eric & Zhou, Rong (2007). A Heuristic Search. In Journal of Artificial Intelligence Research 28 (2007), pp. 267-297.

[2] Likhachev,Maxim & Gordon,Geoff & Thrun,Sebastian (2004). ARA*: Anytime A* with Provable Bounds onSub-Optimality,
https://papers.nips.cc/paper/2382-ara-anytime-a-with-provable-bounds-on-sub-optimality.pdf

[3] Bansal,Nikhil & Blum,Avrim & Chawla,Shuchi & Meyerson,Adam (2004). Approximation Algorithms for Deadline-TSP and Vehicle Routing with Time-Windows,
http://pages.cs.wisc.edu/ shuchi/papers/timewindows.pdf?fbclid=IwAR0wm-jQljso7crMsggmB-5qXIvZx_5f0C16Ol5-35OR6ChYlyOG10mVzY

[4] Gavalas,Damianos & Konstantopoulos,Charalampos & Mastakas, Konstantinos & Pantziou,Grammati. A Survey on Algorithmic Approaches for Solving Tourist Trip Design Problems,
https://www.researchgate.net/publication/271921760_A_survey_on_algorithmic_approaches_for_solving_tourist_trip_design_problems

[5] Korf, Richard E.(2000). Recent Progress in the Design and Analysis of Admissible Heuristic Functions,
https://www.aaai.org/Papers/AAAI/2000/AAAI00-212.pdf

[6] Sneha Sawlani (2017). Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks. A Thesis Presented to the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science University of Denver,
https://digitalcommons.du.edu/cgi/viewcontent.cgi?article=2303&context=etd

[7] Meyavuz (2017). Dijkstra vs Bi-directional Dijkstra Algorithm on US Road Network,
https://meyavuz.wordpress.com/2017/05/14/dijkstra-vs-bi-directional-dijkstra-comparison-on-sample-us-road-network/

[8] Andrew V. Goldberg (Microsoft Research), Chris Harrelson (Google), Haim Kaplan (Tel Aviv University), Renato F. Werneck (Princeton University) (2006). Efficient Point-to-Point Shortest Path Algorithms,
https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf

[9] GeeksforGeeks. Connected Components in an undirected graph,
https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/