U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Conceção e Análise de Algoritmos

TripMate: planeador de viagens multimodais

MIEIC – Turma 6 – Grupo C – Tema 11

(28 de Maio de 2019)

Ana Filipa Campos Senra up201704077@fe.up.pt
David Dinis up201706766@fe.up.pt

Índice

Descrição do tema a ser implementado

TripMate: planeador de viagens multimodais

No dia-a-dia de um utilizador de transportes públicos, pode haver a necessidade de realizar várias viagens a depender das atividades planeadas para um dia, iniciando e terminando geralmente na morada de residência. Por exemplo, um estudante realiza geralmente a viagem casa-escola-casa, que poderá ser ainda combinada com a necessidade de deslocações para localidades onde o estudante realiza atividades extracurriculares, como desporto. Já no caso de um trabalhador, poderão haver deslocações em intervalos do trabalho com objetivos diversos, como a necessidade de uma compra, ou alguma atividade de curta duração, ou mesmo uma reunião de trabalho com cliente. As várias atividades, com as respetivas restrições temporais (instante de início e tempo estimado de duração) compõem a agenda de um indivíduo para um dia.

Neste trabalho, pretende-se implementar uma aplicação que, dada a agenda de atividades de um indivíduo, sugira o plano de viagens na rede de transportes públicos, que poderá conter percursos de autocarro e metro, assim como trechos de caminhada a pé, entre paragens. O plano de viagem deverá ter em consideração os tempos estimados de deslocação total, com o objetivo de atender as restrições temporais das atividades, nomeadamente o seu instante de início e tempo estimado de duração. Sempre que uma nova atividade for adicionada à agenda de um dia, o itinerário deverá ser recalculado.

Formalização do Problema

Dados de Entrada

1. Vi – Conjunto de pontos (num mapa) caracterizados por:
    a. Coordenadas – Indicam a posição do ponto.
    b. Adj ⊆ Ei – Conjunto de arestas que partem do vértice.

2. POI (Que pertencem a Vi) que representa o conjunto de pontos indicados pelo utilizador. (sendo que POI[0] representa o ponto inicial e POI[n+1], sendo n o número de pontos entre o ponto inicial e final, representa o ponto final), ordenados pela hora de início. São caracterizados por:
    a. Coordenadas – representam a posição no mapa.
    b. Hora de Início da Atividade.
    c. Duração da Atividade.

3. Gi(Vi,Ei) – Grafo dirigido cíclico pesado em que os vértices, Vi, representam os vários pontos do mapa e as arestas, Ei, conjunto de meios de transporte que ligam os vértices, unidirecionais ou bidirecionais.

Dados de Saída

1. Gi(Vf,Ef) – Grafo dirigido cíclico pesado em que os vértices, Vf que representa o melhor caminho entre os pontos POI. Possui as mesmas características descritas para os homólogos supra.
2. Vf – Conjunto de vértices que representam o melhor caminho entre o POI[0] e POI[n+1] (passando por todos os outros pontos POI), ordenados.
3. W – Peso total de todas as arestas percorridas no caminho.

Restrições

Restrições dos Dados de Entrada

O peso de cada Aresta deverá ser maior que zero, pois representa sempre grandezas positivas (por exemplo, o tempo de deslocação entre dois pontos).

Restrições dos Dados de Saída

O peso total deverá ser maior ou igual a zero.

O ponto de partida, POI[0], deverá ser o primeiro vértice no conjunto de pontos que representam o melhor caminho entre os POI, Vf.

O ponto de final, POI[n+1], deverá ser o último vértice no conjunto de pontos que representam o melhor caminho entre os POI, Vf.

Todos os pontos indicados por POI deverão estar contidos, por ordem, no conjunto de pontos que representam o melhor caminho entre o POI[0] e POI[n+1], representado por Vf.

Função Objetivo

A solução ótima do problema em análise no presente relatório será minimizar o tempo de deslocação entre os vários pontos ou a distância percorrida. Desta forma, a solução ótima passa pela minimização da respetiva função:

$g(n) = \sum n \in C [Peso(n)] = W$, sendo que Vf representa o conjunto de vértices que representam o melhor caminho entre os pontos POI.

Casos de Utilização e Funcionalidades a serem suportados

O programa a desenvolver tem como objetivo calcular a melhor rota possível dependendo do plano de atividades do utilizador. O utilizador terá introduzir os locais por onde irá passar e os respetivos dados: hora de início e tempo de duração da atividade.

Deste modo, será permitido ao utilizador escolher:

1. Critério de pesquisa:
    a. Menor distancia percorrida;
    b. Menor tempo de viagem;

2. Visualização da viagem sobre o mapa seguindo os critérios de pesquisa
3. Visualização do mapa de toda a rede de transporte públicos (de uma determinada área).

Estruturas de Dados Utilizadas

Representação de um Grafo e estruturas relacionadas

A representação de um grafo foi feita com base na estrutura Graph definida em Graph.h. Esta estrutura é composta pelo conjunto de vértices que compõem o grafo (representado por um std::vector). Foi implementado um género de "look up table" para determinar a localização dos vértices através do seu id, visto que se optou por conservar o id extraído dos dados fornecidos. Optou-se por a conservação do id, para efeitos de testes dos algoritmos ao longo do desenvolvimento do trabalho. Com a implementação desta "look up table" conseguiu-se diminuir o tempo de execução do programa. O grafo possui métodos que permitem a sua manipulação, inserção, remoção, entre outros. Foi, também, nesta classe implementados vários algoritmos necessários ao longo do trabalho. Os vértices do grafo são representados pela classe Node definida em Node.h, que é caracterizada pela classe Spot que possui, por sua vez, o número de identificação (elemento que torna o Node único no grafo), pelas suas coordenadas espaciais bem como pelas suas coordenadas reais e pelas estações de transportes públicos que representa (PublicTransp). O node, também é composto pelo o conjunto das arestas que ligam o vértice a outros vértices do grafo (representado por um std::vector). As arestas do grafo são representadas pela classe Edge, que é composta unicamente pelo seu peso, pelo apontador para o vértice de Origem, pelo apontador para o vértice de Destino e pelo tipo de transporte que representam.

Algoritmos que operam sobre o Grafo

Na implementação dos algoritmos bidirecionais houve a necessidade de objetos não pertencentes a classe Graph. Deste modo, optou-se pela criação de classes que estendem a classe Graph de modo a poder utilizar os recursos desta e a permitir a criação/organização de novos recursos inerentes ao novo algoritmo.

Representação de Paragens de Transportes Públicos

Implementou-se a classe TranspStop que serve como base para as paragens de transporte público, possuindo os seguintes atributos: id do vértice mais próximo da paragem, o código da Paragem e a linha.

A classe Bus estende a supra e representa uma paragem de autocarro e possui os seguintes atributos: código da Zona, o Município, o distrito, a morada e o tipo de paragem (Abrigada, Poste ou Marcação.

A classe Subway estende a classe TranspStop não possuindo quaisquer atributos extra, pois a informação fornecida apenas continua estes atributos.

Menu

A classe Menu representa toda a organização do trabalho. Esta classe tem como atributos o Grafo de toda a cidade e a agenda inserida pelo utilizador. Esta classe 'faz' a comunicação com o utilizador tanto para adicionar novos pontos na agenda como para mostrar o Grafo da cidade ou do percurso inserido na agenda.

Graph Viewer

Este módulo foi-nos fornecido por parte da equipa docente da Unidade Curricular e é responsável por tornar possível a visualização gráfica dos grafos manipulados pelo programa.

Mutable Priority Queue

A classe Mutable Priority Queue foi-nos fornecida pela equipa docente da Unidade Curricular e é uma classe auxiliar aos algoritmos implementados. Esta representa uma fila mutável de prioridade.

Edge

A classe Edge representa as determinadas ligações entre os pontos do mapa, tendo como atributos o ponto de origem, o ponto de destino, o peso associado e o tipo de ligação, quer seja a pé, de autocarro ou metro. Esta classe é usada para calcular o caminho mais rápido, que utilizará os transportes públicos.

PublicTransp

Esta classe irá guardar todos os pontos dos autocarros e dos metros.

Spot

Representa um ponto do mapa em que tem coordenadas em x e y e coordenadas em latitude e longitude para o utilizador poder dizer o seu ponto inicial. Esta classe também tem uma instância da classe PublicTransp que irá servir para verificar se este spot tem algum transporte público.

Info_calendar

A classe info_calender representa uma atividade, em que cada atividade terá o local, a hora de início e o tempo da atividade.

Calender

Esta classe irá guardar todas as atividades que o utilizador pretender ir.

Vertex

A classe vertex tem todos os pontos do grafo, que irá ser utilizado nos algoritmos.

MapParser

Esta classe tem como por objetivo preencher o grafo com os pontos que nos foram fornecidos nos ficheiros de texto, como também ler os ficheiros dos transportes públicos, e também preencher o grafo com os edges, ligando assim os pontos do mapa.

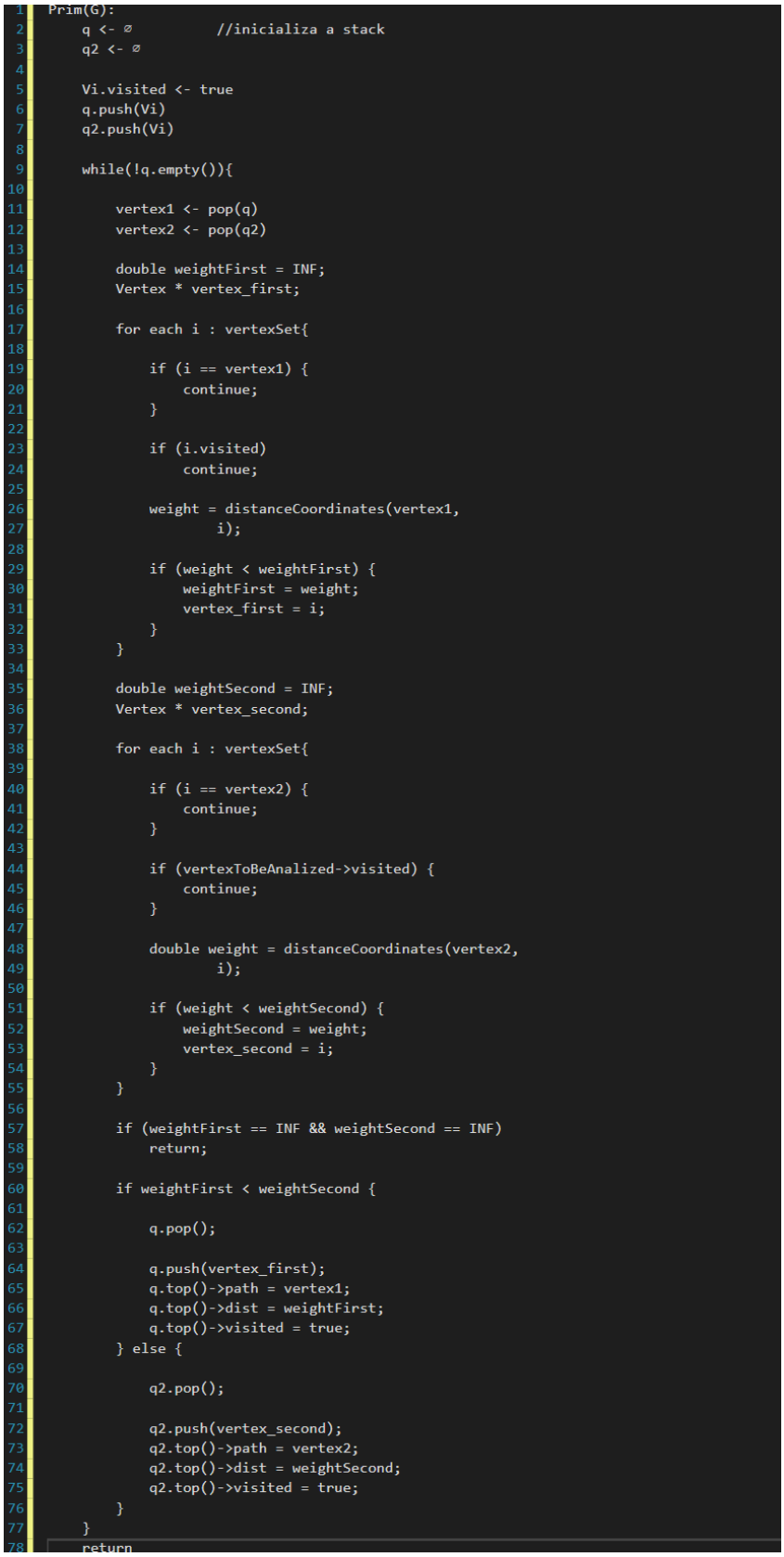
Solução Implementada

Esta secção descreve a solução implementada, tanto para encontrar o caminho ótimo para as viagens do utilizador, tanto para a obtenção das linhas de transportes públicos.

Obtenção das Linhas de Transportes Públicos

Antes de qualquer algoritmo de procura do caminho mais rápido entre os pontos inseridos na agenda pelo utilizador, mostrou-se necessário obter as linhas dos transportes públicos. Como a informação que nos foi fornecida sobre os transportes públicos foi apenas da localização das paragens e das linhas que passavam por esta, vimo-nos obrigados a implementar um algoritmo para simular os serviços de transporte público.

O algoritmo implementado foi o Algoritmo de Prim com algumas modificações. O Algoritmo de Prim tem como objetivo a obtenção de um grafo onde a soma total das arestas é minimizada e onde todos os vértices são interligados. No entanto, o nosso objetivo para além do descrito supra, era, de igual forma, que cada vértice só tivesse conectado, no máximo, a dois outros (não permite ramificações).



```
Prim(G):
    q <- ∅          //inicializa a stack
    q2 <- ∅

    Vi.visited <- true
    q.push(Vi)
    q2.push(Vi)

    while(!q.empty()){

        vertex1 <- pop(q)
        vertex2 <- pop(q2)

        double weightFirst = INF;
        Vertex * vertex_first;

        for each i : vertexSet{

            if (i == vertex1) {
                continue;
            }

            if (i.visited)
                continue;

            weight = distanceCoordinates(vertex1,
                    i);

            if (weight < weightFirst) {
                weightFirst = weight;
                vertex_first = i;
            }
        }

        double weightSecond = INF;
        Vertex * vertex_second;

        for each i : vertexSet{

            if (i == vertex2) {
                continue;
            }

            if (vertexToBeAnalized->visited) {
                continue;
            }

            double weight = distanceCoordinates(vertex2,
                    i);

            if (weight < weightSecond) {
                weightSecond = weight;
                vertex_second = i;
            }
        }

        if (weightFirst == INF && weightSecond == INF)
            return;

        if weightFirst < weightSecond {

            q.pop();

            q.push(vertex_first);
            q.top()->path = vertex1;
            q.top()->dist = weightFirst;
            q.top()->visited = true;
        } else {

            q2.pop();

            q2.push(vertex_second);
            q2.top()->path = vertex2;
            q2.top()->dist = weightSecond;
            q2.top()->visited = true;
        }
    }
    return
```

Figura 1 – Pseudocódigo do algoritmo de Prim

O algoritmo terá, no melhor caso, complexidade O(|E|*log(|V|) e, no pior, O(V^2).

Caminho ótimo da agenda do utilizador

Mostra-se imperativo relembrar ao leitor que optamos por usar a seguinte estratégia: obter o caminho ótimo entre locais adjacentes ordenados pelas horas descritas na agenda. Deste modo, a solução descrita abaixo, aplica-se a obter o melhor caminho entre dois locais (sendo que o caminho final foi obtido 'adicionando' todos os subcaminhos).

Iteração 1: Verificação da possibilidade de Navegar entre dois Locais Adjacentes

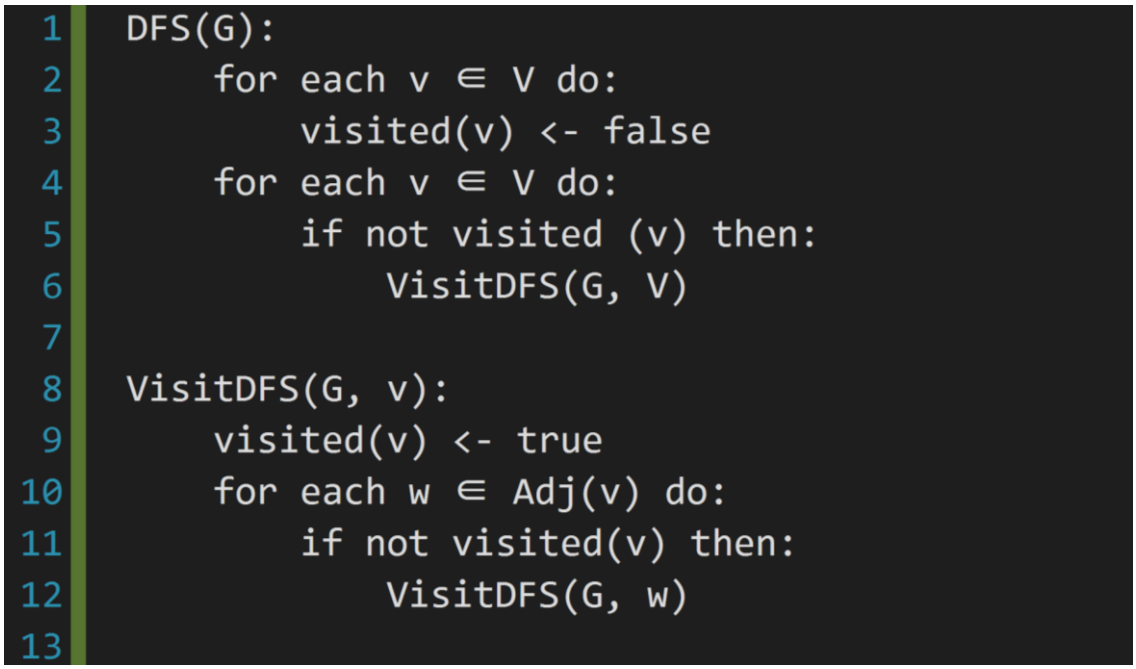
Nesta iteração analisamos a possibilidade de chegar a um vértice de Destino a partir de um Vértice de Destino. Para o fazermos basta fazermos uma simples pesquisa no grafo a partir do vértice de origem.

Esta análise pode ser feita de duas maneiras díspar: através da Pesquisa em Profundidade ou através da Pesquisa em Largura.

Pesquisa em Profundidade

O algoritmo de Pesquisa em Profundidade apresenta-se como um algoritmo de estrutura recursiva: todas as arestas são exploradas do vértice recentemente descoberto.

A sua implementação poderá ser feita recursivamente ou com recurso a uma pilha. Optamos no entanto, pela solução recursiva, pois ao estarmos a trabalhar com grafos relativamente pequenos, o ganho em rapidez em relação a uma implementação através de uma pilha é bastante apelativo. Foi utilizada uma variável em cada vértice para representar se o vértice já tinha sido visitado pelo algoritmo. Se no vértice de destino o booleano "visited" for verdadeiro, então há conectividade entre os dois vértices.



```
1  DFS(G):
2      for each v ∈ V do:
3          visited(v) <- false
4      for each v ∈ V do:
5          if not visited (v) then:
6              VisitDFS(G, V)
7
8  VisitDFS(G, v):
9      visited(v) <- true
10     for each w ∈ Adj(v) do:
11         if not visited(v) then:
12             VisitDFS(G, w)
13
```

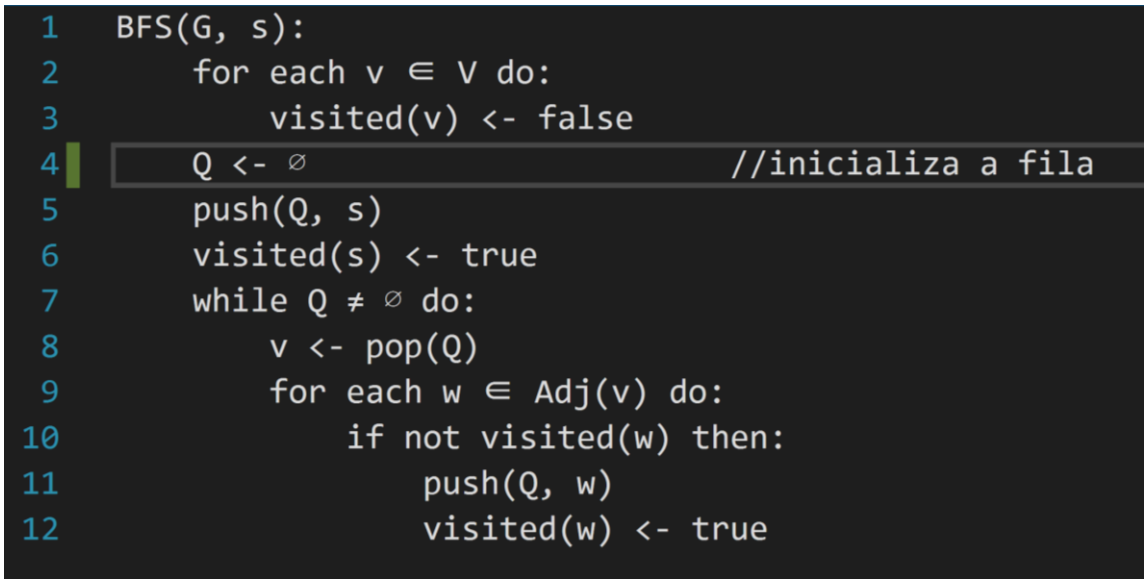
Figura 2 – Pseudocódigo do algoritmo Pesquisa em Profundidade

A complexidade temporal deste algoritmo é O(|V| + |E|): é linear no tamanho total do grafo (V representa o número de vértices do grafo e E o número de arestas).

Quando à complexidade espacial, ao termos uma variável especial para representar se este vértice foi visitado, podemos, afirmar que é de complexidade O(|0|).

Pesquisa em Largura

O algoritmo de pesquisa em profundidade assenta num conceito aposto ao supra: são exploradas todas as arestas a partir do vértice em análise, passando só depois para o vértice seguinte. Este algoritmo é, usualmente, implementado através do recurso a uma fila: o vértice no topo desta é analisado e os seus adjacentes são colocados no final desta (First In First Out).



```
1   BFS(G, s):
2       for each v ∈ V do:
3           visited(v) <- false
4       Q <- ∅                          //inicializa a fila
5       push(Q, s)
6       visited(s) <- true
7       while Q ≠ ∅ do:
8           v <- pop(Q)
9           for each w ∈ Adj(v) do:
10              if not visited(w) then:
11                  push(Q, w)
12                  visited(w) <- true
```

Figura 3 – Pseudocódigo do algoritmo Pesquisa em Largura

A complexidade deste algoritmo é O(|V| + |E|), tal como na Pesquisa em Profundidade.

A complexidade espacial é, a contrário do algoritmo apresentado supra O(|V|), devido à fila de prioridade.

Iteração 2: Melhor percurso entre dois Locais

A problemática de obter o melhor percurso entre dois locais é um problema já muito estuado na teoria de grafos. Deste modo, para a resolução desta problemática recorremos à implementação do algoritmo Dijkstra e, posteriormente, à implementação do algoritmo A* (um melhoramento do anterior).

Com a finalidade de reduzir o tempo de execução do programa, os algoritmos apresentados de seguida foram apenas executados no final de uma pesquisa em profundidade. Deste modo, podemos evitar o processamento desnecessário ao aplicar algoritmos complexos quando não conectividade entre dois pontos.

Algoritmo Dijkstra

O algoritmo Dijkstra, introduzido por Edsger W. Dijkstra, tem como finalidade calcular o melhor caminho entre a origem e qualquer outro vértice do grafo.

Devido à natureza da problemática a resolucionar, optamos por na nossa implementação do algoritmo fazê-lo fixar em apenas encontrar o caminho ótimo entre o vértice de origem e o vértice de destino.

Como este é um algoritmo de natureza *greedy* e por garantir sempre o melhor caminho entre os vértices de origem e destino, torna-se um algoritmo bastante apelativo.

O algoritmo possui um comportamento semelhante ao de Pesquisa em Largura. No entanto, possuem uma diferença chave: ao invés de utilizar uma fila como um contentor auxiliar, utilizada uma fila de prioridade onde os vértices com menor peso total desde a origem possuem prioridade.

Na nossa implementação, ao encontrar o vértice de destino no topo da fila de prioridade, o algoritmo dá-se como concluído procedendo-se à reconstrução do caminho. Para o fazer optamos por guardar em cada vértice a informação do vértice anterior do melhor caminho da origem até este.

Quando o algoritmo é utilizado para encontrar o caminho mais rápido optamos por fazer uma estimativa bastante livre do tempo que demoraria a percorrer uma aresta mediante o transporte que essa aresta representa. Esta estimativa foi feita através da distância das coordenadas em quilómetros de dois vértices, e dos seguintes valores médios para a velocidade média de cada transporte: apeado 5 Km/H; metro 50 Km/H; e, autocarro 45 Km/H. Ao autocarro e metro foi adicionado 1 minuto adicional por cada paragem que este efetua.

```
Dijkstra(G,Vi,Vf):
    q <- ∅          //inicializa a fila de prioridade

    q.push(Vi)

    while(!q.empty()){

        v <- pop(q)

        if equals(v, Vf) then
            return;

        for each w: Adjacent(v) do

            oldDist = distance(w)
            if distance(v) + weight(v,w) < distance(w)
                distance(w) <- distance(v) + weight(v,w)
                path(v) <- w

                if(oldDist == INF)
                   q.insert(w)
                else
                   q.updateQueue
```

Figura 4 – Pseudocódigo do algoritmo Dijkstra

Deste modo, o algoritmo possui duas fazes crucias: obter o melhor caminho e a reconstrução do mesmo.

A primeira fase, obtenção do melhor caminho, pode ser subdivida em duas fazes: a primeira subfase consiste em preparar os vértices para a execução da segunda subfase, podendo ser executada em O(|V|).

A segunda subfase consiste na construção do melhor caminho. Como foi mencionado supra, este algoritmo é em tudo semelhante à Pesquisa em Largura com a exceção de utilizar uma Fila de Prioridade ao invés de uma fila. Ao contrário do que acontece na fila, a inserção de um elemento numa fila de prioridade é feita em tempo logarítmico O(log (n)) e a remoção em tempo linear. Portanto, a complexidade temporal desta fase do algoritmo é O(|V| + |E|)*log(|V|).

A segunda fase do algoritmo, reconstrução do caminho ótimo, é feita, no pior dos casos, em tempo linear relativamente ao número de vértices no grafo, O(|V|).

Podemos, assim, concluir que a nossa implementação do algoritmo de Dijkstra tem complexidade temporal O(2*|V| + |E|)*log(|V|).

Algoritmo A*

Após a implementação do algoritmo anteriormente descrito, procedeu-se à implementação do Algoritmo A*.

O Algoritmo A*, introduzido em 1968 por Peter Hart, Nils Nilsson e Bertram Raphael, pode ser visto como uma extensão do Algoritmo Disjkstra. Este usa uma função heurística para o "guiar" na pesquisa pelo destino, avançando no sentido onde a custo de viajar entre o ponto atual ao destino é menor.

Podemos assim concluir que o algoritmo de Dijkstra pode ser visto como o Algoritmo A* com uma função heurística igual a 0. Logo, o pseudocódigo do Algoritmo A* é em tudo semelhante ao algoritmo de Dijkstra, visto que a única diferença está na ordenação da queue.

No entanto, o algoritmo A* apenas garante a solução ótima se a função heurística é sempre menor (ou igual ao) custo verdadeiro entre os vértices. Neste algoritmo, apenas implementado para a distância mais curta, utilizamos a função heurística que representa o tempo de deslocação com velocidade média igual ao do metro entre dois pontos (sendo que a distância foi as das coordenadas dois pontos). Deste modo, como esta será sempre menor ou igual ao peso verdadeiro entre dois vértices, o algoritmo A* obtém o caminho ótimo na nossa implementação.

Deste modo, o Algoritmo A* é em tudo semelhante ao algoritmo de Dijkstra e, por isso, possui a mesma complexidade temporal, (O((|V|+|E|)*log(|V|)), e a mesma complexidade espacial, O(|V|).

Algoritmos Bidirecionais

Foram implementados todos algoritmos bidirecionais homólogos aos descritos suma.

Um algoritmo Bidirecional de Pesquisa passa por fazer uma pesquisa em dois sentidos: uma no sentido da origem até ao destino e a outra no sentido do destino até à origem. A sua condição de terminação não será, como seria de esperar, encontrar um ponto em comum entre as duas pesquisas.

Seja v o primeiro vértice comum encontrado entre as duas pesquisas. É importante salientar que o caminho encontrado poderá não ser o ótimo. Podemos concluir, assim, a condição de paragem sugerida supra, não é suficiente.

Deste modo, mostra-se necessário manter o tamanho do melhor caminho encontrado até ao momento. Seja w, o tamanho do melhor caminho encontrado. Inicialmente, este será igual ao infinito e será atualizado sempre que for encontrado um novo vértice, v, que cumpra esta condição df(v) + dr(v) < w, sendo que df representa a distância na pesquisa do sentido da origem ao destino e dr a distância na pesquisa contrária. Quando a soma da distância da origem até aos vértices nos topos das filas em ambas as pesquisas for igual ou maior a w, o algoritmo poderá parar.

Logo, a condição de paragem é w < d(topf) + d(topr).

Os algoritmos Bidirecionais terão, em teoria, no pior caso a mesma complexidade temporal que os seus homólogos. No entanto, esta poderá não ser a melhor métrica para os avaliar, pois em casos reais, estes tendem a cobrir uma distância bastante menor que os algoritmos direcionais. Se R for a distância em linha reta entre o ponto inicial e final, uma pesquisa direcional terá um custo de O(R^2), enquanto a bidirecional terá O(2*(R/2)^2), podendo ser até duas vezes mais rápido.

```
Dijkstra(G, Gi, Vi,Vf):
    q <- ∅          //inicializa a fila de prioridade
    q2 <- ∅         //inicializa a fila de prioridade

    q.push(Vi)
    q2.push(Vf)

    TotalWeight = INF
    spotFinish = Vf

    while(!q.empty() && !q2.empty()){

        v <- pop(q)
        v2 = pop(q2)

        totalWeightTmp = v.dist + v2.dist

        if totalWightTmp >= TotalWeight
            break;

        if equals(v, Vf) then
            break;

        if(Gi.findVertex(v).dist != INF){

        totalWeightTmp = Gi.findVertex(v).dist + v.dist

            if(TotalWeight > totalWeightTmp){
                TotalWeight = totalWeightTmp
                spotFinish = v
            }

        }

        for each w: Adjacent(v) do{

            oldDist = distance(w)
            if distance(v) + weight(v,w) < distance(w)
                 distance(w) <- distance(w) + weight(v,w)
                 path(v) <- w

                 if(oldDist == INF)
                    q.insert(w)
                else
                    q.updateQueue
        }


        //Inverted Graph

        if equals(v2, Vf) then
            break;

        if(G.findVertex(v2).dist != INF){

        totalWeightTmp = G.findVertex(v2).dist + v2.dist

            if(TotalWeight > totalWeightTmp){
                TotalWeight = totalWeightTmp
                spotFinish = v2
            }

        }

        for each w: Adjacent(v) do{

            oldDist = distance(w)
            if distance(v2) + weight(v2,w) < distance(w)
                 distance(w) <- distance(v) + weight(v2,w)
                 path(v) <- w

                 if(oldDist == INF)
                    q2.insert(w)
                else
                    q2.updateQueue
        }

    }
```

Figura 5 – Pseudocódigo do algoritmo Bidirecional Dijkstra ou A*

Conclusão

O trabalho a que o presente relatório se refere mostrou-se bastante desafiante. A falta de experiência com as estruturas de dados e com os algoritmos a desenvolver, tornou algo desafiante a elaboração da aplicação.

No desenrolar deste trabalho foram encontradas 3 grandes dificuldades:

No início do desenvolvimento da aplicação foi referente à organização dos dados tanto do grafo como dos transportes públicos.

Podemos, de igual modo, referir que o desenvolvimento dos algoritmos bidirecionais foi desafiante.

No entanto, a grande dificuldade encontrada foi o desenvolvimento do algoritmo para replicar as linhas de transporte público, devido ao desconhecimento à priori do algoritmo de Prim e à sua modificação adequada.

Mostra-se necessário referir que os dados fornecidos referentes às paragens dos autocarros estavam extremamente pobres na métrica das linhas, o que resultou em linhas bastante irreais. O contrário podemos afirmar sobre as linhas referentes ao metro.

Gostaríamos de agradecer ao Professor Francisco Andrade por toda a ajuda dada durante as aulas Teórico-práticas. Gostaríamos, também, de agradecer ao Monitor da Unidade Curricular Xavier Fontes, pois foi uma preciosa ajuda preciosa na elaboração deste trabalho.

A distribuição da carga de trabalho a que se refere o presente relatório foi a seguinte:

O desenvolvimento de todos os algoritmos, da sua descrição no presente relatório (Solução Implementada), da visualização dos grafos e da grande maioria das funções para obter as informações tanto do grafo como das paragens de transportes públicos ficou a cabo da Filipa Senra.

A elaboração do Menu, a elaboração do relatório à exceção da Solução Implementada, e a documentação do código ficou a cabo do David Dinis.

A elaboração da estrutura do código foi realizada em conjunto pelos dois autores deste relatório.

Bibliografia

Brian Kallehauge, Jesper Larsen, Oli B.G. Madsen, Marius M. Solomon. “Vehicle Routing Problem With Time Windows.” s.d. http://www.alvarestech.com/temp/vrptw/Vehicle%20Routing%20Problem%20with%20Time%20Windows.pdf (acedido em 19 de Março de 2019).

Gimel’farb, Georgy. “Shortest Paths - Dijkstra Bellman-Ford Floyd All-pairs paths.” University of Auckland. s.d. https://www.cs.auckland.ac.nz/compsci220s1c/lectures/2016S1C/CS220-Lectures27-29.pdf (acedido em 20 de Maio de 2019).

Goldberg, Andrew V., Chris Harrelson, Haim Kaplan, e Renato F. Werneck. “Efficient Point-to-Point Shortest Path Algorithms.” Princeton.edu. s.d. http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf (acedido em 17 de Maio de 2019).

Google. Vehicle Routing Problem with Time Windows. 19 de Abril de 2019. https://developers.google.com/optimization/routing/vrptw#time_window_constraints (acedido em 20 de Abril de 2019).

Semantic Scholar. “ Intro to Algorithms .” Semantic Scholar. 8 de Abril de 2016. https://pdfs.semanticscholar.org/4599/add6498fc5bb26260aa54c87bf7855a9f7f5.pdf (acedido em 2019 de Maio de 20).

Wikipedia. Vehicle routing problem. 19 de Janeiro de 2019. https://en.wikipedia.org/wiki/Vehicle_routing_problem (acedido em 20 de Abril de 2019).