Mestrado Integrado em
Engenharia Eletrotécnica e
de Computadores

TÉCNICO LISBOA

Algoritmos e Estruturas de Dados

2014/2015 - Segundo ano - Primeiro semestre

Relatório do projeto

ISTravel

Grupo número 13

TOMÁS CARDOSO	número 79007	tomas.cardoso@tecnico.ulisboa.pt
JOÃO VIEIRA	número 79191	joaomiguelvieira@tecnico.ulisboa.pt

9 de Maio de 2015.

Docente: **Carlos Bispo**

Conteúdo

Introdução

Descrição do problema

O problema proposto pelo enunciado consiste, muito sinteticamente, em analisar um mapa de cidades que é transmitido à aplicação sob a forma de ficheiro de texto descrevendo todas as ligações entre as cidades vizinhas (numeradas de 1 a V) e encontrar caminhos ótimos entre duas cidades segundo critérios de tempo e custo. As informações fornecidas acerca deste mapa foram:

- Uma ligação entre duas cidades é bidirecional, isto é, existe da cidade A para a cidade B e, igualmente, da cidade B para a cidade A;
- Cada ligação é caracterizada por:
  - Meio de transporte;
  - Horário da ligação (hora do primeiro transporte, hora do último transporte e periodicidade da ligação);
  - Custo da viagem.
- As caracterizações das ligações não têm de ser congruentes com a realidade.

Pode ainda existir mais do que uma ligação entre duas cidades.

A análise deste mapa permite a resolução do problema principal que é imposto: encontrar o caminho ótimo em tempo ou o caminho ótimo em custo entre duas cidades do mapa. Adicionalmente podem rejeitar-se ligações que sejam caras demais ou demorem demasiado tempo. Para esse efeito nem sequer se considera que tais ligações existem. Finda a resolução do problema pode ainda ser imposto um limite de tempo ou custo máximos para a solução, sendo que estas restrições têm, obrigatoriamente, que concordar com o critério de análise definido. Quando tal acontece verifica-se se a solução encontrada respeita o limite imposto. Se tal não acontecer a conclusão é que não há solução para o problema.

Abordagem

A natureza deste problema sugere claramente uma análise com recurso a grafos. Neste caso os vértices do grafo a analisar são as cidades do mapa enquanto as arestas correspondem às ligações entre elas. Visto que é dito que as ligações entre cidades são recíprocas, o grafo resultante não é direcionado. O problema que se pretende resolver tendo o grafo é do tipo "origem para destino" que passa por resolver um problema de "origem para todos". O algoritmo utilizado na resolução deste problema é o conhecido algoritmo de *Dijkstra* com algumas modificações. Este algoritmo é descrito em mais detalhe na secção 2.1 deste relatório. Este algoritmo determina a árvore de caminhos mais curtos entre o vértice de origem fixado e todos os outros vértices do grafo. Se não houver caminho entre dois vértices significa que os dois pertencem a sub-grafos que não estão ligados, o que na prática significa que não existe caminho possível entre as duas cidades.

Há ainda que referir que, devido a questões de otimização de memória, o modelo de representação escolhido para o grafo foi de listas de adjacências. Isto deve-se ao facto de, na generalidade, a representação por matriz de adjacências ser completamente avassaladora no que toca à gestão de memória. A representação por matriz de adjacências tem uma complexidade total de utilização de memória de $O(N^2)$, que pode ser reduzida para $O(\frac{1}{2}N^2 - \frac{N}{2})$ (se apenas for alocada metade da matriz - triangular

superior ou inferior), o que para grafos esparsos significa uma má gestão de memória, pois apenas os índices a_{ij} cujas cidades i e j possuam uma ligação serão relevantes, sendo todas as outras posições de memória desnecessárias.

Capítulo 1

Arquitetura

1.1 Funcionamento das rotinas principais

O fluxograma ilustrado na figura 1.1 representa a arquitetura da classe `ISTravel`. O código primário de funcionamento da aplicação baseia-se neste fluxograma, sendo que todas as funções mais específicas que são chamadas pelo mesmo estão implementadas noutras classes.

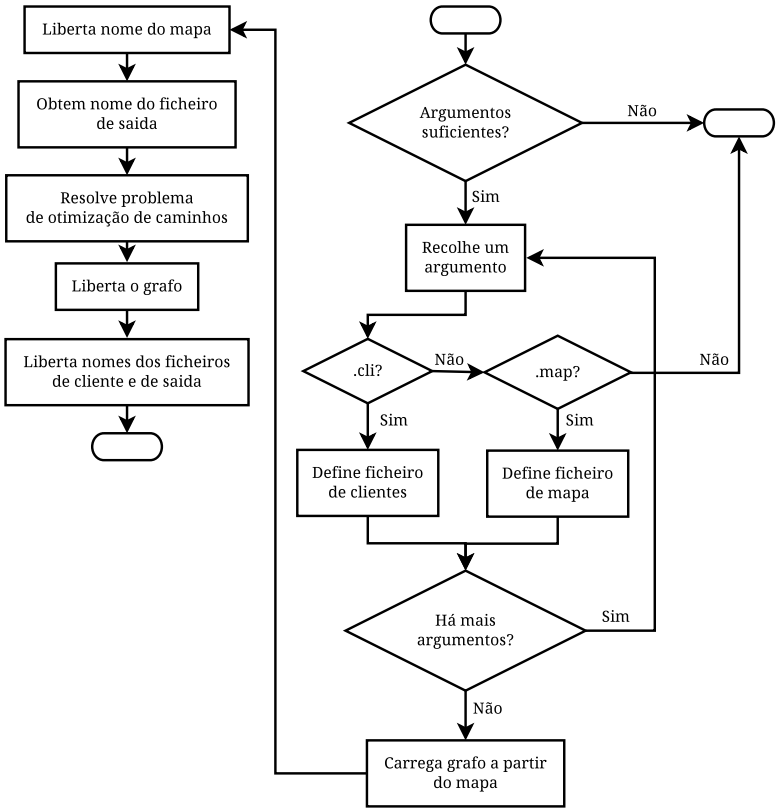

Figura 1.1: *Fluxograma da classe `ISTravel`.*

O funcionamento da função principal, apesar de estar esquematizado na figura 1.1 está, de certa forma, oculto por detrás da função `computeBestPath`, implementada pela interface `Utilities`. Esta interface não é mais do que uma lista de funções utilizadas pelo programa principal ou por outras funções da própria interface. De modo a deixar claro o funcionamento geral do programa, esquematiza-se ainda, na figura 1.2 o funcionamento da função `computeBestPath`.

1.2 Subsistemas funcionais e estruturas de dados

A estrutura escolhida durante o desenvolvimento do projeto apresenta-se na figura 1.3. A linguagem utilizada faz uma analogia com linguagens de programação orientadas a objetos, mas um dos objetivos

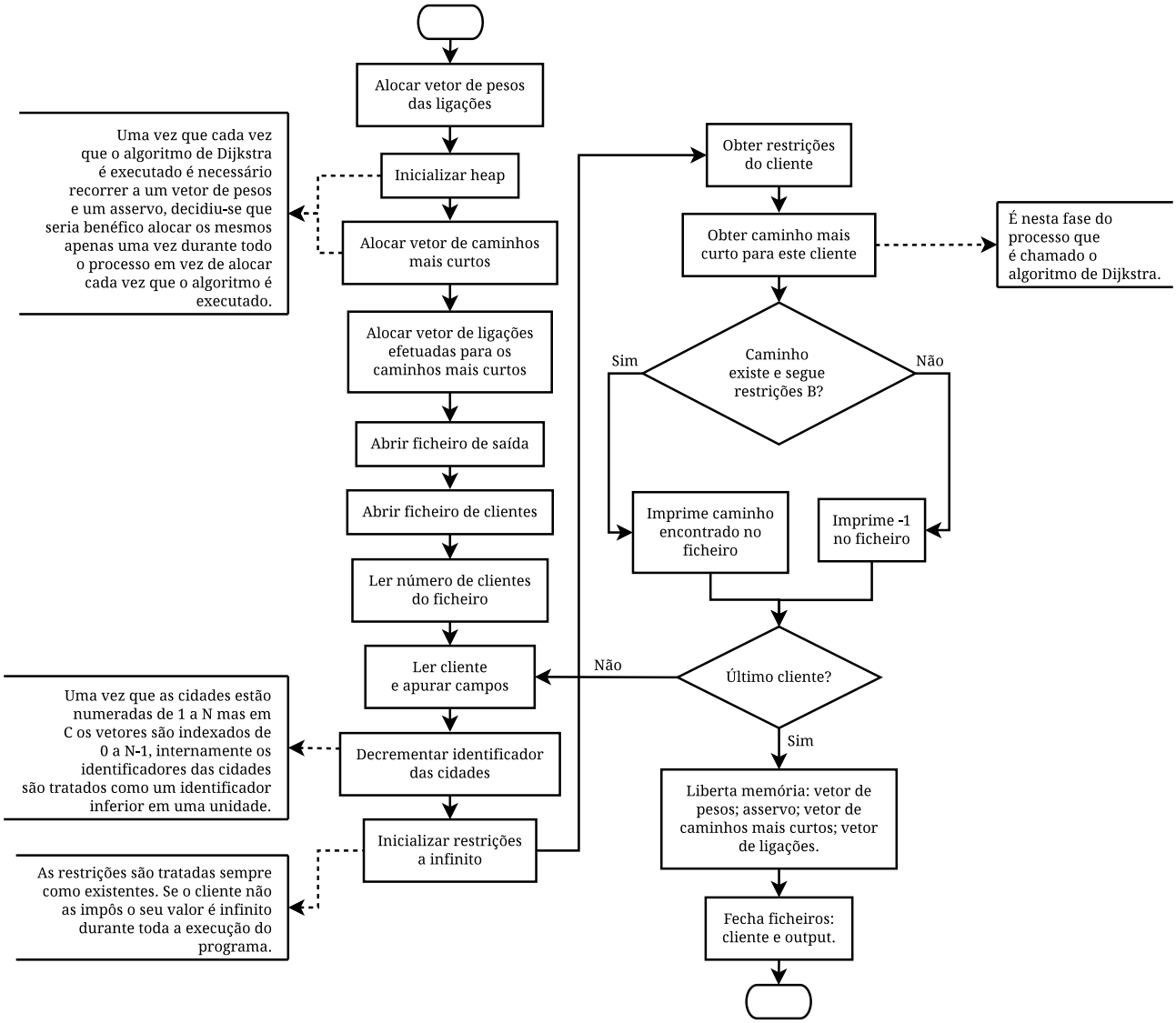

Figura 1.2: *Fluxograma da rotina* `computeBestPath` *implementada em* `Utilities.c`.

do planeamento deste projeto foi exatamente que assim fosse. Dos seis ficheiros de código além do ficheiro contentor do programa principal, apenas quatro deles representam subsistemas independentes, sendo que os os ficheiros `Defs.c` e `Utilities.c` implementam apenas funções sem declararem nenhuma nova estrutura de dados.

1.2.1 `Heap`

O subsistema `Heap` implementa um tipo muito particular de acervos. Os elementos aceites neste acervo são inteiros, todos eles são diferentes e o valor de nenhum deles excede o valor do último índice possível do acervo. Por outras palavras os seus elementos tem de ser os próprios índices da tabela. Além disso o critério de comparação que verifica se um elemento é maior do que outro é o peso do primeiro ser menor do que o peso do segundo. A condição de acervo utilizada é a de que o pai tem de ser sempre maior do que os seus dois filhos.

A estrutura de dados utilizada por esta classe dispõe de três vetores de inteiros:

- Um vetor que armazena os pesos dos elementos do acervo. O peso do elemento i encontra-se armazenado na posição i deste vetor;
- Outro que contem as posições dos V elementos do acervo. A posição do elemento i encontra-se na posição i deste vetor;

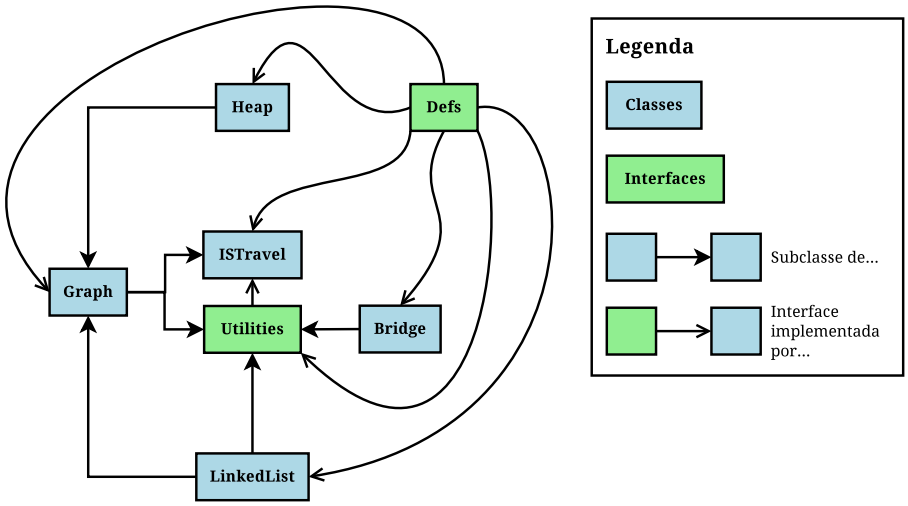

Figura 1.3: *Estrutura principal e relação entre os subsistemas presentes no projeto.*

- Um último vetor que contém os elemento do acervo sendo que o elemento na posição 0 é sempre o mais prioritário.

Além dos vetores anteriormente enumerados, o acervo contem ainda a informação de qual o tamanho máximo que pode atingir e quantos elementos ele tem numa determinada altura. Este tipo de acervo é local, isto é, não é dinâmico, pelo que no ato de construção há que referir qual o tamanho máximo que o mesmo pode atingir. A figura 1.4 ilustra um exemplo de um acervo de tamanho 10.

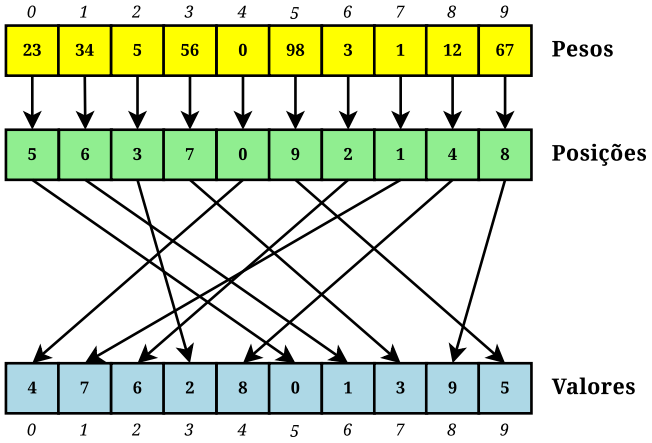

Figura 1.4: *Acervo de tamanho 10 com todas as posições ocupadas.*

Os ficheiros onde este subsistema está implementado são `Heap.c` e `Heap.h`, sendo este último o *header*. As assinaturas das funções existentes neste subsistema apresentam-se de seguida.

```
1  /* Elimina todos os elementos do heap */
   void cleanHeap( Heap * h );
3
   /* Insere um elemento no heap sem o reorganizar */
5  int directInsert( Heap * h , int element );

7  /* Corrige um elemento em relacao aos seus inferiores */
   void fixDown( Heap * h , int k );
9
   /* Corrige um elemento em relacao aos seus superiores */
11 void fixUp( Heap * h , int k );

13 /* Liberta o heap */
   void freeHeap( Heap * h );
15
```

```
/* Funcao de sorting baseada em acervos */
void heapSort( Heap * h );

/* Converte uma tabela num heap */
void heapify( Heap * h );

/* Insere um elemento no heap e repoe a condicao de acervo */
int insert( Heap * h , int element );

/* Cria um novo heap */
Heap * newHeap( int size , int * priority );

/* Remove o maior elemento do heap */
int removeMax( Heap * h );

/* Verifica se uma tabela e um acervo */
int verifyHeap( Heap * h );

/* Corrige a posicao de um elemento do acervo */
void incPriority( Heap * heap , int index );

/* Verifica se um acervo esta vazio */
int isHeapEmpty( Heap * heap );
```

1.2.2 `LinkedList`

A classe `LinkedList` implementa listas genéricas simplesmente ligadas. A estrutura principal deste subsistema contém apenas dois ponteiros, um para o elemento seguinte e outro para uma estrutura desconhecida que está "armazenada" no elemento da lista.

A figura 1.5 ilustra a estrutura da lista.

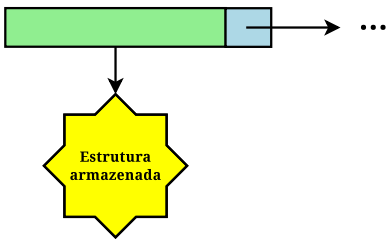

Figura 1.5: *Estrutura de um elemento do subsistema* ***`LinkedList`****.*

Os ficheiros onde este subsistema está implementado são `LinkedList.c` e `LinkedList.h`, sendo este último o *header*. As assinaturas das funções existentes neste subsistema apresentam-se de seguida.

```
/* Inicia uma nova lista */
LinkedList * initList();

/* Insere um elemento no inicio da lista */
LinkedList * insertUnsortedItemList( LinkedList * list , Item item );

/* Insere um elemento na lista segundo um criterio de ordenacao */
LinkedList * insertSortedItemList( LinkedList * list , Item item , int ( * lessThan )(
    Item , Item ) );

/* Retorna o elemento seguinte */
LinkedList * getNextNodeList( LinkedList * node );

/* Retorna o elemento anterior */
LinkedList * getPreviousNodeList( LinkedList * list , LinkedList * node );

/* Retorna o conteudo de um elemento */
Item getItemNode( LinkedList * node );
```

```
/* Liberta uma lista */
void freeList( LinkedList * list , void ( * freeStructure )( Item ) );

/* Itera uma lista */
void forEach( LinkedList * list , void ( * function )( Item ) );

/* Retorna o tamanho de uma lista */
int getListLenght( LinkedList * list );

/* Insere um elemento na segunda posicao de uma lista (nao altera o inicio) */
void insertWithNoReturn( LinkedList * list , Item item );
```

1.2.3 `Bridge`

Apesar de diminuto, o sistema `Bridge` também se pode considerar um subsistema funcional. As estruturas operadas neste sistema são nada mais do que campos de características respeitantes a uma determinada viagem. Os métodos destas estruturas são apenas o seu construtor, um libertador de memória e os *setters* e *getters* respeitantes aos seus campos. Os campos contidos nesta estrutura são:

- Transporte envolvido;
- Duração da viagem;
- Preço;
- Momento em que ocorre a primeira ligação;
- Momento a partir do qual não se efetuam mais ligações diárias;
- Periodicidade da ligação.

Esta estrutura nada sabe sobre as cidades envolvidas nesta ligação.

Os ficheiros onde este subsistema está implementado são `Bridge.c` e `Bridge.h`, sendo este último o *header*. As assinaturas das funções existentes neste subsistema apresentam-se de seguida.

```
/* Constroi uma nova estrutura */
Bridge * newBridge( int transportation , int duration , int price , int first_travel ,
    int no_more_travels , int period );

/* Liberta memoria de uma estrutura */
void freeBridge( Item bridge );

/* Getter para o tipo de transporte de um peso */
int getTransportation( Bridge * bridge );

/* Getter para a duracao de um peso */
int getDuration( Bridge * bridge );

/* Getter para o preco de um peso */
int getPrice( Bridge * bridge );

/* Getter para o horario da primeira viagem */
int getFirstTravel( Bridge * bridge );

/* Getter para o tempo a partir do qual nao existem mais viagens */
int getNoMoreTravels( Bridge * bridge );

/* Getter para o periodo entre viagens */
int getPeriod( Bridge * bridge );
```

1.2.4 `Graph`

A classe `Graph` é talvez a mais completa das anteriormente enunciadas. O tipo de dados que a mesma manipula são estruturas de grafos baseados em listas de adjacências. Além da estrutura principal que contem o grafo, as listas de adjacências dos seus vértices e outros dois campos que são o número total de vértices e de arestas, existe ainda uma estrutura de dados secundária denominada `Link`. Esta estrutura é responsável por armazenar a informação respeitante a uma adjacência de um vértice no grafo, e tem apenas dois campos: um que indica o vértice ao qual o "dono do *link*" tem a adjacência e outro que é um ponteiro para o peso dessa ligação que é abstrato ao grafo. A figura 1.6 representa sucintamente as estruturas de dados.

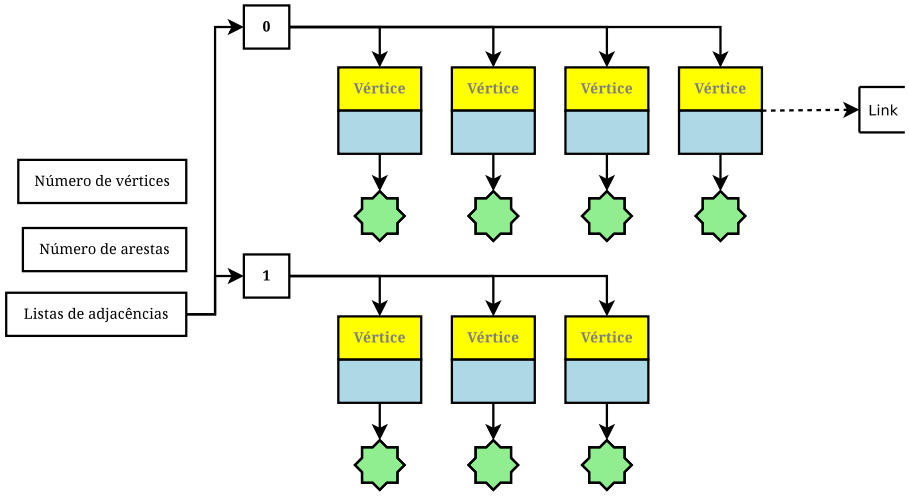

Figura 1.6: *Ilustração da estrutura de um grafo baseado em listas de adjacências (meramente ilustrativo - a quantidade de listas de adjacências está incompleta, etc).*

Acerca dos métodos deste subsistema, para além do algoritmo de *Dijkstra* não há muito que dizer. Existem apenas construtores e destruidores das estruturas, *setters* e *getters* para os campos das mesmas e uma função que imprime no ecrã o conteúdo qualitativo das listas de adjacências do grafo, isto é, que vértices é que são adjacentes entre si.

Os ficheiros onde este subsistema está implementado são `Graph.c` e `Graph.h`, sendo este último o *header*. As assinaturas das funções existentes neste subsistema apresentam-se de seguida.

```
1  /* Iniciliza um grafo */
   Graph * graphInit( int vertices , int bridges , void ( * freeWeight )( Item ) );
3
   /* Insere uma aresta */
5  void graphLinkInsert( Graph * graph , int city1 , int city2 , Item weight );

7  /* Liberta a memoria associada a um grafo */
   void freeGraph( Graph * graph );
9
   /* Cria uma nova aresta */
11 Link * newLink( int vertice , Item weight );

13 /* Calcula a arvore de caminhos mais curtos dado um vertice de origem */
   void dijkstra( Graph * graph , int origin , int * shortest_path_tree , Item * bridges ,
        int ( * weigh )( int , Item , Item * ) , int * weights , Heap * priority_queue );
15
   /* Getter para o vertice de uma aresta */
17 int getVertice( Link * link );

19 /* Getter para o peso de uma aresta */
   Item getWeight( Link * link );
21
   /* Getter para o numero de vertices de um grafo */
23 int getNumberOfVertices( Graph * graph );
```

```
25 /* Imprime as adjacencias de um grafo */
   void printGraph( Graph * graph );
27
   /* Getter para as adjacencias de um vertice */
29 LinkedList * getAdjacencies( Graph * graph , int vertice );

31 /* Setter para o peso de uma aresta */
   void setWeight( Link * link , Item weight );
```

1.3 Interfaces

Como referido, existem ainda dois pares de ficheiros que não constituem subsistemas funcionais mas sim interfaces com funções utilizadas por outros subsistemas. Seguidamente mostram-se as assinaturas das funções da interface `Utilities` que estão descritas em `Utilities.h`

```
   /* Mostra ao utilizador a correta utilizacao do programa */
2  void usage( char ** argv );

4  /* Verifica se um ficheiro tem uma determinada extensao */
   int equalExtentions( char * string , char * extention );
6
   /* Salva uma string noutra */
8  char * saveString( char * string );

10 /* Obtem o nome do ficheiro de destino atraves do ficheiro de input */
   char * getOutputFileName( char * input_filename , char * input_file_extention , char *
       output_file_extention );
12
   /* Carrega a estrutura do grafo a partir do ficheiro */
14 Graph * getMapFromFile( char * map_filename );

16 /* Resolve sucessivamente os problemas de otimizacao dos clientes do ficheiro de input
       */
   void computeBestPath( Graph * map , char * client_filename , char * output_filename );
18
   /* Pesa uma aresta do grafo e retorna qual dos pesos e mais vantajoso (pode haver mais
       do que um) */
20 int weigh( int counter , Item bridge , Item * efective_weight );

22 /* Retorna o tempo de uma viagem e o tempo de espera */
   int getTime( int counter , Bridge * bridge );
24
   /* Imprime o caminho da cidade de origem ate a cidade de destino no ficheiro */
26 void printPath( FILE * output_file , int * shortest_path_tree , Bridge ** bridges , int
        destination , int origin );

28 /* Calcula o preco total de uma viagem */
   int getTotalPrice( int * shortest_path_tree , Bridge ** bridges , int origin , int
       destination );
30
   /* Calcula o tempo total de uma viagem */
32 int getTotalTime( int * shortest_path_tree , Bridge ** bridges , int origin , int
       destination );

34 /* Verifica se uma aresta segue as restricoes do tipo A */
   int followRestrictionsA( Bridge * bridge );
36
   /* Verifica se uma aresta segue as restricoes do tipo B */
38 int followRestrictionsB( int weight );

40 /* Define uma restricao */
   void defineRestriction( char * restriction , char * value );
42
   /* Liberta o peso de uma aresta */
```

```
44 void freeWeight( Item weight );
```

A interface `Defs.h` é mais pequena. Seguidamente mostram-se as assinaturas das funções que esta implementa.

```
/* Controla os erros que o programa pode gerar */
void makeException( int error_code );

/* Codifica um transporte */
int getTransportationFromString( char * transportation_s );

/* Codifica uma restricao */
int getRestriction( char * restriction_s );

/* Codifica o criterio de otimizacao */
int getOptimizationCriterium( char * criterium );
```

Capítulo 2

Algoritmos

Os principais algoritmos usados neste projeto encontram-se implementados pelas funções `dijkstra` e `getMapFromFile`. Esta última é responsável por carregar toda a estrutura do grafo, enquanto o conhecido algoritmo de *Dijkstra* calcula a árvore de caminhos mais curtos dado um vértice de referência. Existem outros algoritmos secundários mas não menos importantes para o correto funcionamento do programa. Neste capítulo será abordado também o método `getTime` que calcula o tempo que dura uma determinada ligação entre duas cidades.

2.1 Algoritmo de *Dijkstra*

O algoritmo de *Dijkstra* é, senão um dos mais importantes, um dos mais utilizados na teoria de grafos. Este algoritmo determina os caminhos mais curtos entre um vértice fonte e todos os outros vértices do grafo, se existir algum caminho. Existe também um outro algoritmo, o de *Floyd*, que resolve o género de problema "todos para todos".

A complexidade do algoritmo de *Dijkstra* utilizando acervos é $E \lg V$, sendo E o número de arestas e V o número de vértices.

A figura 2.1 ilustra o funcionamento da variante com *heaps* do algoritmo. Uma outra modificação efetuada foi o rompimento do algoritmo assim que seja encontrado o primeiro peso infinito. Isto deve-se ao facto de que, sendo o primeiro elemento do acervo o de maior prioridade, se o primeiro elemento tiver peso infinito todos os outros também têm, e, por isso, não possuem qualquer ligação ao grafo.

2.2 `getMapFromFile`

O algoritmo implementado em `getMapFromFile` é o responsável por carregar toda a estrutura do grafo a partir do ficheiro. A figura 2.2 ilustra o funcionamento desta rotina.

A complexidade deste algoritmo é de $O(EV)$. Isto porque se optou por fazer inserção direta das adjacências no grafo em vez de criar uma estrutura auxiliar que diminuísse a complexidade temporal deste processo. Como podem existir mais do que uma aresta entre duas cidades, decidiu-se que a melhor forma de otimizar temporalmente a análise do grafo seria agrupar as arestas que ligam duas cidades. Assim sendo, o campo `weight` de uma aresta não é, de facto, um peso mas uma lista de pesos em que o melhor será escolhido de acordo com os critérios de otimização e restrições atuais. Visto que por cada aresta que se quer inserir no grafo há que verificar se essas duas cidades já estão ligadas (verificar se uma está na lista de adjacências da outra), é necessário verificar, no pior caso, toda a lista de adjacências que, mais uma vez no pior caso, pode ter tamanho V. Assim sendo a complexidade total de inserção aproxima-se por $O(EV)$.

2.3 `getTime`

A rotina `getTime` constitui um algoritmo simples com complexidade de $O(1)$. A opção de explicitá-la aqui surgiu pelo facto de ser uma rotina importante que tem de determinar o tempo total de uma

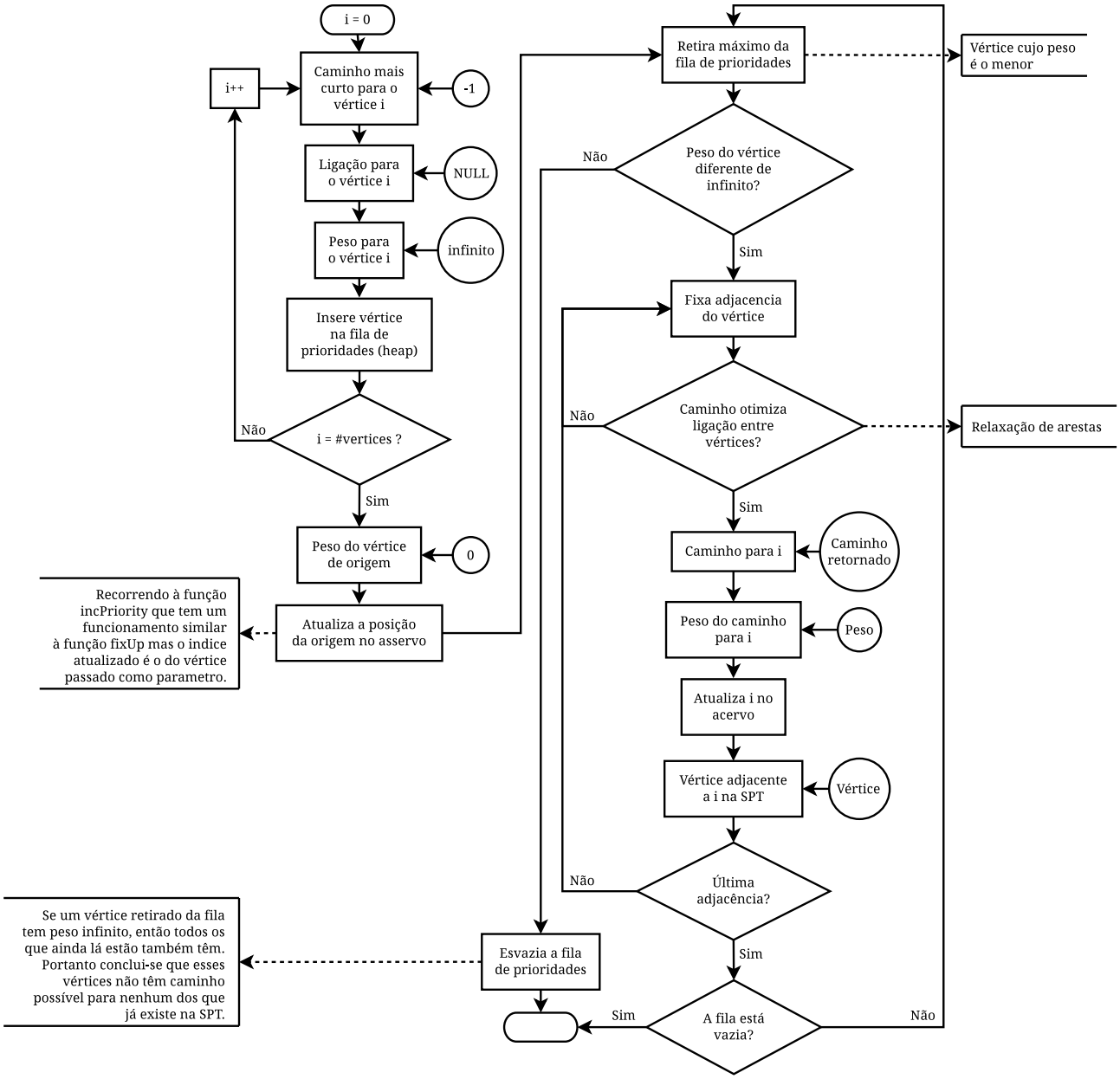

Figura 2.1: *Ilustração do funcionamento do algoritmo de* Dijkstra *com acervos.*

ligação que passa por determinar o tempo de espera por um transporte. A imagem 2.3 ilustra o funcionamento desta rotina.

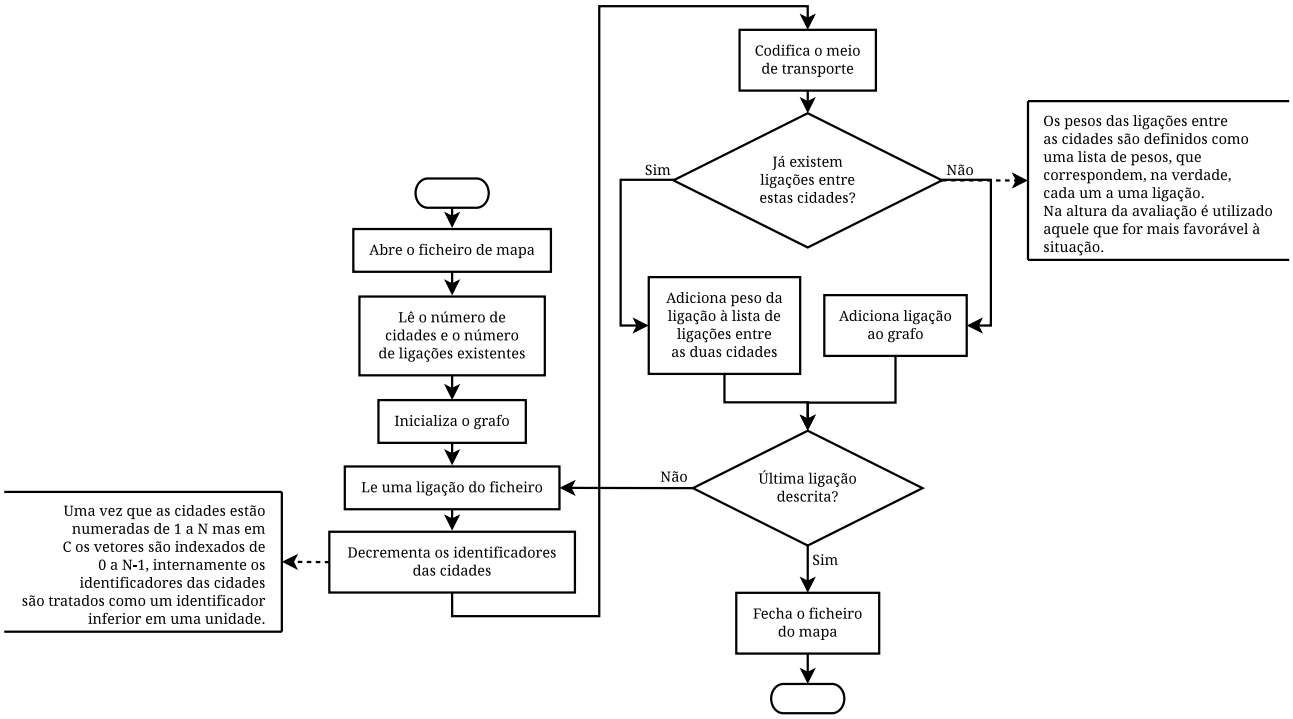

Figura 2.2: *Fluxograma da rotina* `getMapFromFile`.

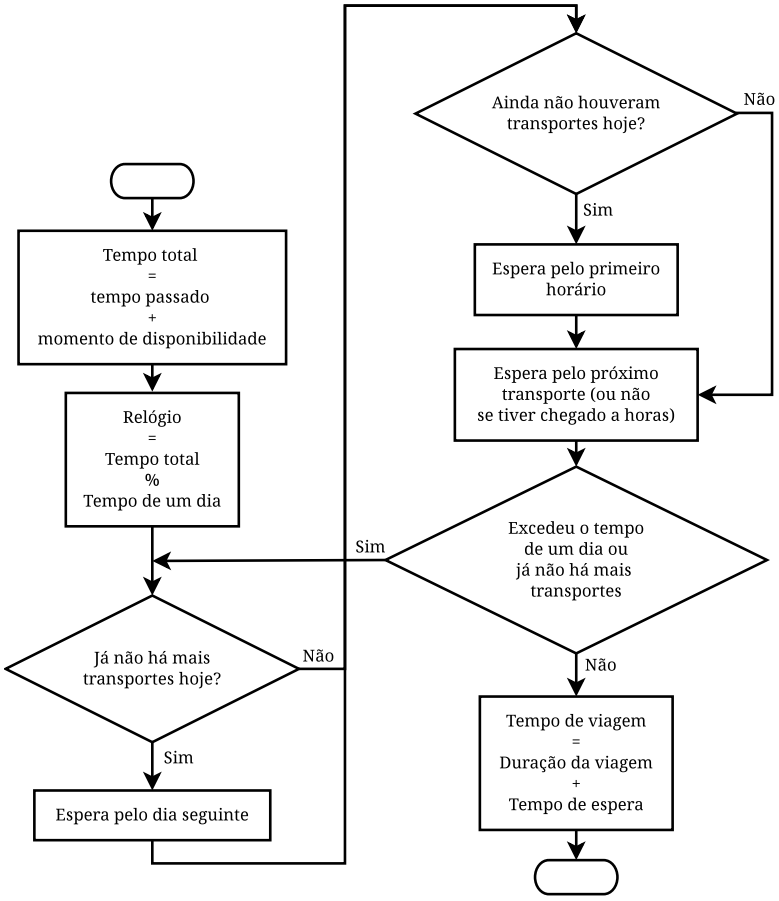

Figura 2.3: *Fluxograma da rotina* `getTime`.

Capítulo 3

Análise dos requisitos computacionais

3.1 Carregamento do mapa

Sem dúvida alguma que a construção do grafo constitui um processo com uma complexidade temporal nada desprezável. Como analisado na secção 2.2, a complexidade deste processo é proporcional a EV. Esta complexidade poderia ser diminuída através da utilização de uma matriz de adjacências temporária para onde seriam carregadas as arestas para posteriormente serem colocadas nas listas de adjacências do grafo. Acontece que este processo aumenta a complexidade de memória proporcionalmente a V^2, o que não é, de todo, desejável. Assim sendo opta-se por pagar o preço do tempo de computação extraordinário.

3.2 Principais algoritmos

Grande parte dos algoritmos recursivos utilizados neste programa apresentam complexidade linear. São exemplos: `getTotalTime`, `freeList`, `printPath`, etc.

O algoritmo de *Dijkstra* com recurso a acervos, como já foi dito anteriormente, tem uma complexidade da ordem de $E \lg V$.

Todos os restantes métodos do programa têm complexidade não superior a $O(N)$, pelo que, excetuando o carregamento do mapa (que predomina), a maior das complexidades de tempo é dada pelo algoritmo de *Dijkstra*.

3.3 Complexidade de memória

Acerca da utilização de memória, esta terá o seu pico durante a execução do algoritmo de *Dijkstra*. Nesta fase do programa todo o grafo está em memória, além de um acervo que contém dois vetores de tamanho V e três vetores adicionais, um de ponteiros para os pesos das ligações, outro com os pesos numéricos e um último que contém a *shortest path tree*. Desta forma conclui-se que a complexidade total de memória do programa é proporcional a $E + 6V \rightsquigarrow O(E + V)$.

Capítulo 4

Análise crítica

4.1 Funcionamento do programa

Aquando da primeira submissão no *Mooshak*, o programa passou em apenas 8 dos 20 testes de depuração para correto funcionamento devido a uma solução não ótima. Isto acontecia porque, no algoritmo de *Dijkstra*, quando era retirado o máximo da fila de prioridades, não era feita a atualização e ordenação do vetor de posições. Após a correção deste problema, o programa já funcionou corretamente para todos os testes em termos da solução encontrada, tendo apenas chumbado em alguns por questões de memória ou tempo de execução. Com isto, foram feitas mais algumas alterações ao nível das estruturas utilizadas, o que permitiu passar a 18 testes dos 20. Todas as decisões em termos de estruturas de dados e algoritmos foram sempre feitas tentando encontrar o melhor balanço entre memória ocupada e tempo de execução do programa, o que permitiu, no fim, concluir grande parte dos testes. Apesar de a solução proposta não ser ótima considera-se que a mesma é uma boa aproximação do que seria um algoritmo ótimo.

4.2 Melhorias possíveis e reconhecidas

O facto do programa desenvolvido reprovar em dois dos 20 testes ao qual foi submetido prova que a solução não é ótima. Acerca da complexidade de memória, pensa-se que seria possível reduzir a mesma atacando o problema nos acervos: reduzir o número de vetores dos quais o acervo depende (pois têm todos tamanho V). Seria possível também eliminar o vetor de ponteiros para pesos, contornando esta questão com outras estratégias, e isto deveria bastar para produzir alterações significativas no programa.

No que toca à complexidade temporal, o verdadeiro problema está no carregamento do acervo. Julga-se que este problema poderia ser solucionado se as arestas fossem carregados para uma tabela de acervos inicialmente (a posição i da tabela conteria todas as arestas respeitantes ao vértice i) e, por analogia com o algoritmo de *Dijkstra*, as arestas seriam retiradas uma por uma do acervo seguindo a ordem dos índices dos vértices, o que faria com que fossem mais facilmente agrupadas. Este processo teria uma complexidade total menor que a do método usado que é proporcional a EV. A razão que levou à decisão de não proceder a esta modificação (além da escassez de tempo para a implementação) foi que este processo poderia representar um aumento da complexidade de memória, pelo que seria uma tentativa que significaria algum risco.

Capítulo 5

Exemplo detalhado

Considere-se o seguinte ficheiro de mapa:

```
 10 9
5 4 autocarro 33 100 0 1440 60
4 10 autocarro 33 100 0 1440 60
10 1 autocarro 33 100 0 1440 60
1 8 autocarro 33 100 0 1440 60
8 6 autocarro 33 100 0 1440 60
6 9 autocarro 33 100 0 1440 60
9 3 autocarro 33 100 0 1440 60
3 2 autocarro 33 100 0 1440 60
2 7 autocarro 33 100 0 1440 60
```

Figura 5.1: *Exemplo de ficheiro de mapa.*

O mapa é carregado para o grafo através da rotina `getMapFromFile`. Neste caso o grafo terá 10 vértices e 9 arestas entre eles. Cada ligação é verificada separadamente, para apurar se já existe alguma ligação entre as duas cidades a que a mesma respeita. Depois da análise concluída a aresta deve ser inserida no grafo de uma das duas maneiras: o peso é adicionado a uma lista de pesos de uma aresta já existente ou é criada uma aresta de raiz.
Seguidamente a função `computeBestPath` iniciará a resolução do problema principal, e começará a ler o ficheiro de clientes. Neste caso só existe um cliente:

```
1 5 7 0 custo 1 B2 1000.
```

Começa-se por processar a informação relativa a este cliente, lendo, por ordem, o número de cliente, a cidade de origem, a cidade de destino, o momento a partir do qual o cliente está disponível para viajar, o critério de otimização escolhido, o número de restrições e poderão ser lidos um ou dois campos adicionais correspondentes às restrições.
O critério de otimização é codificado pela função `getOptimizationCriterium`, as restrições são inicializadas a infinito e, por fim, as restrições impostas pelo cliente são definidas nos respetivos campos pela rotina `defineRestriction`.
Após toda a informação sobre o cliente ter sido recolhida procede-se à resolução do problema recorrendo para isso ao algoritmo de *Dijkstra*.
O algoritmo de *Dijkstra* começará por fazer a inicialização dos vetores dos quais vais fazer uso: todas as posições da `shortest path tree` a -1, todas as posições do vetor de pesos a ∞ e todas as posições do vetor de *bridges* a `NULL`. Todos os vértices são ainda adicionados ao acervo. Seguidamente define que o peso da origem é 0 (o que acontece por definição), e então o processo de cálculo da SPT começa. O vértice cujo peso é o mais baixo (mais prioritário) é retirado da fila prioritária, que no caso da primeira iteração será a própria origem cujo peso (critério de ordenação do acervo) é zero. De seguida procede-se à iteração da lista de adjacências do vértice retirado da fila, e para cada uma das adjacências verifica-se se esta minimiza a distância à origem de algum vértice ou caminho (relaxação de aresta e

caminho, respetivamente), e, se for o caso, este vértice entra para a SPT para a posição dada pelo vértice retirado da fila (que lhe é adjacente) e a sua posição no acervo é corrigida. O peso da aresta é calculado externamente pela rotina `weigh` que calcula o peso tendo em conta as restrições do tipo A (se a aresta não as respeitar o peso será ∞) e o critério de otimização. No fim de analisar todas as adjacências do vértice volta-se a retirar outro da fila, que será novamente aquele que tiver peso menor. Se o seu peso for infinito o processo termina, pois os vértices restantes não têm ligação ao grafo, caso contrário repete-se o processo anterior até que tal aconteça ou que a fila fique vazia.
Partindo do vértice 5, o algoritmo de *Dijkstra* irá procurar, a cada iteração, o vértice que está mais próximo da origem. Quando encontra o que está mais próximo, adiciona à posição indexada por este na franja o seu sucessor na *shortest path tree*.
A tabela 5.1 descreve o percurso do algoritmo para este exemplo desde a primeira iteração até à última em que é obtida a árvore de caminhos mais curtos.

	1	2	3	4	5	6	7	8	9	10
5	∞	∞	∞	5/100	-	∞	∞	∞	∞	∞
4	∞	∞	∞	-	-	∞	∞	∞	∞	4/200
10	10/300	∞	∞	-	-	∞	∞	∞	∞	-
1	-	∞	∞	-	-	∞	∞	1/400	∞	-
8	-	∞	∞	-	-	8/500	∞	-	∞	-
6	-	∞	∞	-	-	-	∞	-	6/600	-
9	-	∞	3/700	-	-	-	∞	-	-	-
3	-	3/800	-	-	-	-	∞	-	-	-
2	-	-	-	-	-	-	2/900	-	-	-
7	-	-	-	-	-	-	-	-	-	-

Tabela 5.1: *Algoritmo de* Dijkstra *aplicado ao exemplo.*

A franja final após a execução do algoritmo de *Dijkstra* dá a árvore de caminhos mais curtos, sendo que em cada índice correspondente a um vértice está a cidade à qual este está ligado na SPT.

1	2	3	4	5	6	7	8	9	10
10	3	9	5	-1	8	2	1	6	4

Tabela 5.2: *Franja final (corresponde à SPT).*

A árvore de caminhos mais curtos, para este caso em específico, é linear, e pode ser representada por

`5 - 4 - 10 - 1 - 8 - 6 - 9 - 3 - 2 - 7.`

Após a execução do algoritmo de *Dijkstra*, o programa imprime no ficheiro de saída o caminho encontrado para cada cliente.
Se o caminho não existir ou se não for cumprida a restrição do tipo B, caso exista, é impresso -1 na linha do cliente, se não, é chamada a função `printPath`, uma função recursiva, responsável por imprimir no ficheiro o caminho encontrado para o cliente segundo as diretrizes impostas.
Finalmente, é necessário imprimir também o tempo total da viagem, chamando a função `getTotalTime`, caso não tenha sido calculado pelo algoritmo de *Dijkstra* e o preço chamando a rotina `getTotalPrice`, caso o mesmo não tenha sido calculado.
Neste exemplo, o cliente quer ir da cidade 5 para a 7, com o critério de otimização de custo e uma restrição do tipo B2, que diz que o custo total da viagem não pode ser superior a 1000 euros.
Assim, recorrendo à SPT, verifica-se que o cliente terá de passar por todas as cidades para chegar à cidade 7. Utilizando o programa temos o seguinte ficheiro de saída:

```
1 5 autocarro 4 autocarro 10 autocarro 1 autocarro 8 autocarro 6 autocarro 9
autocarro 3 autocarro 2 autocarro 7 513 900
```

Como se pode verificar, o custo total calculado é 900, que é menor que 1000, restrição imposta pelo cliente, pelo que foi encontrado um caminho dentro dos parâmetros impostos pelo cliente.

Bibliografia

(1) Acetatos das aulas teóricas.