Algoritmos e Estruturas de Dados

MEEC

# Projeto "ISTravel"

*Autores:*

Ana Beatriz Neto Cerqueira (78209) -
anabiacerqueira@gmail.com
Francisco Maria Girbal Eiras (79034) -
francisco.girbal@gmail.com

*Grupo:*
**42**

*Docente:*

Luís Manuel Marques Custódio

9 de Dezembro de 2014

# Contents

# 1 Introdução ao problema

Neste projeto foi-nos proposto o desenvolvimento de um programa agente de viagens.

O seu objetivo centra-se em determinar e sugerir a melhor rota e os meios de transporte necessários, de modo a levar um cliente de uma determinada cidade até à cidade desejada, de acordo com critérios e algumas restrições especificadas.

As cidades estão definidas numa rede de percursos predefinida e fornecida ao programa como input (**ficheiro.map**), sendo também necessário um ficheiro que contenha as informações do cliente (**ficheiro.cli**). A solução é apresentada, por fim, num outro ficheiro (**ficheiro.sol**). Cada ligação é definida pelas cidades que liga, pelo meio de transporte considerado (avião, comboio, barco ou autocarro), pela duração da viagem, pelo seu custo e por um conjunto de parâmetros relativos à sua frequência. Por outro lado, o cliente define a cidade de origem e de destino, podendo escolher o critério a minimizar e ainda, no máximo, duas restrições a aplicar ao itinerário calculado.

Para atingir este objetivo, utilizámos uma variedade de estruturas de dados, como *graphs*, *heaps*, entre outras, e algoritmos, tais como o $Dijkstra$. No final, a solução, apesar de não totalmente eficiente, produziu o resultado esperado, cumprindo as expectativas.

# 2 Abordagem escolhida

Inicialmente, o problema foi reduzido à representação do mapa/rede de ligações mediante a utilização de uma estrutura de dados. Devido ao facto do objetivo do programa ser encontrar o caminho mais curto entre duas cidades, a abordagem baseou-se na representação da rede através de um $grafo$, em que cada cidade corresponde a um vértice do mesmo, e cada aresta a uma ligação (no mapa) entre as cidades.

Numa análise mais detalhada dos dados do enunciado, optou-se por escolher a representação do $grafo$ por **lista de adjacências** devido ao facto de que este seria acedido rapidamente num caso médio, poupando memória significativamente (em oposição a uma matriz de adjacências).

Posteriormente, analisou-se a segunda parte do projeto: leitura dos dados do cliente e obtenção da solução. De forma a poupar memória, os dados dos clientes nunca foram guardados sequencialmente, sendo antes processados de acordo com a ordem de entrada dos mesmos. Para os processar e encontrar a solução foi utilizado o algoritmo de $Dijkstra$, com a variante de utilização de um *acervo* para representar os vértices adjacentes. No algoritmo verificam-se as restrições de ligações. Após a obtenção do caminho, este é escrito no ficheiro de *output* através de uma função recursiva, que verifica também a restrição de custo e tempo total.

# 3 Arquitetura do programa

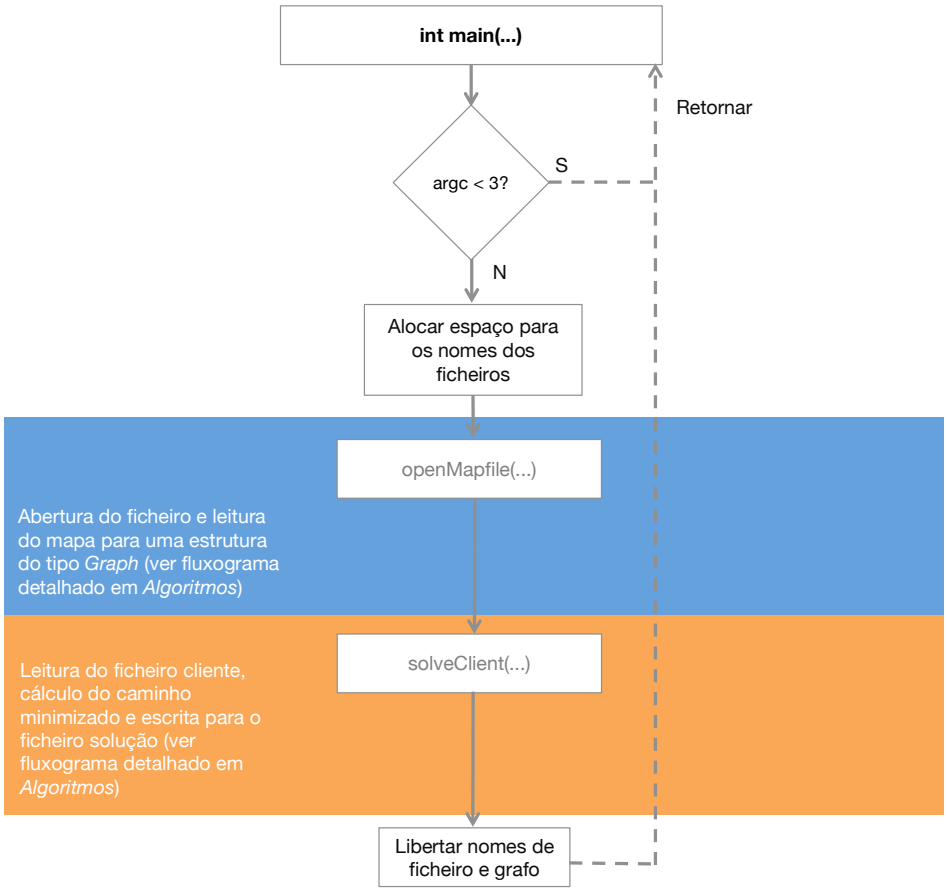


Figure 1: Fluxograma geral do programa em *istravel.c*

# 4 Estruturas de Dados e Algoritmos Utilizados

## 4.1 Estruturas de Dados

Para este projeto foram criadas 7 estruturas de dados independentes essenciais para uma solução mais eficiente do problema.

### 4.1.1 *link* - **struct** *node*

```
struct node {
   int v;
   Info* nodeInfo;
   link *next;
};
```

A estrutura *link* foi criada de modo a possibilitar a implementação de uma **linked list** necessária, devido à representação do grafo por lista de adjacências. Esta estrutura armazena, portanto, um vértice (o fim da ligação iniciada em cada uma das posições do vector *adj* de *Graph* (ver a seguir), uma $Info*$ que contém a informação da aresta e ainda um ponteiro para a próxima, *next*.

### 4.1.2 *Graph* - **struct** *graph*

```
struct graph{
   int V;
   int E;
   link **adj;
};
```

*Graph* é uma estrutura que representa um grafo por lista de adjacências. Como tal, é essencial que este tenha um inteiro, $V$, que contém o número de vértices (neste caso cidades) do mapa, outro, $E$, que contenha o número de arestas do grafo e ainda uma tabela de $link*$ que corresponde à lista de vértices adjacentes ao vértice índice (ver figura).

### 4.1.3 *transporte*

```
typedef enum{aviao, comboio, barco, autocarro} transporte;
```

Uma descrição através de um **enum** dos vários meios de transporte existentes (a ser utilizada por $Info$).

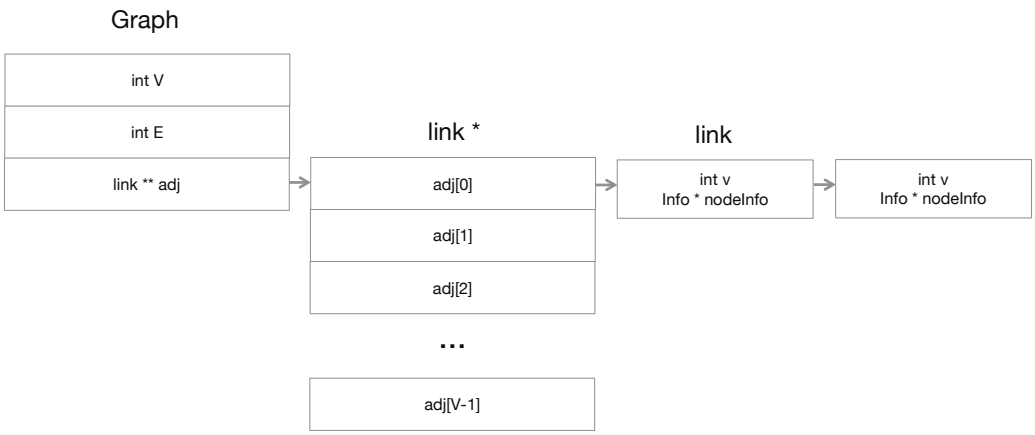


Figure 2: Representação esquemática da interligação entre estruturas de dados

#### 4.1.4 $Info$ - **struct** $info$

```
struct info {
   transporte meio;
   int duracao;
   int custo;
   int t1;
   int tf;
   int p;
};
```

$Info$ constitui uma estrutura de dados que representa a aresta do grafo, contendo vários pesos associados ao percorrê-la. Neste problema especificamente, $Info$ possui um meio de transporte, *meio*, a duração do transporte, o seu custo, o instante da partida do primeiro transporte ($t1$), do último ($tf$) e a periodicidade com que estes ocorrem ($p$).

#### 4.1.5 *Heap* - struct *_heap*

```
struct _heap {
  int (*less) (Item, Item);
  int n_elements;
  int size;
  Item *heapdata;
};
```

A estrutura de dados *heap* possui o número de elementos do heap, *n_elements*, a sua capacidade, *size*, um ponteiro para uma função para comparar dois *Item* e garantir que se mantém a condição de acervo, *less*, e ainda uma tabela de *Item*, *heapdata*.

#### 4.1.6 *Item*

```
typedef data * Item;
```

Um *Item* corresponde apenas a um ponteiro para *data* (ver seguinte). Esta dupla definição existe de forma a que seja mais simples alterar o tipo de dados que o *heap* armazena sem necessitar de alterar um grande número de ficheiros (e ainda abstração de dados).

#### 4.1.7 *data* - struct dataHeap

```
typedef struct dataHeap
{
  int vertice;
  int peso;
} data;
```

Uma estrutura do tipo *data*, utilizada no *heap* através de ponteiros, corresponde a um inteiro *vertice* e outro *peso*, de forma a permitir que a condição de acervo se verifique, comparando os parâmetros *peso*.

### 4.2 Algoritmos

#### 4.2.1 Ler mapa e armazenar no *Graph*

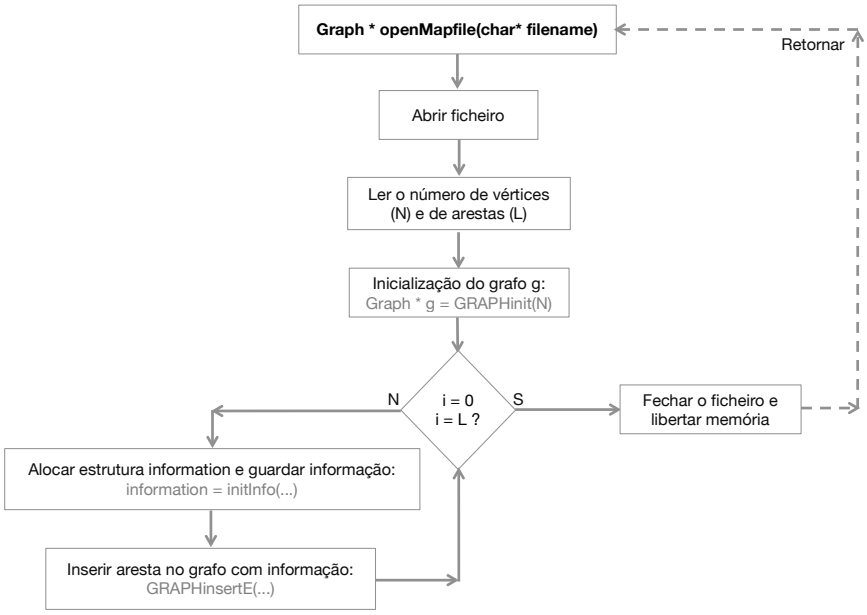


Figure 3: Fluxograma detalhado do algoritmo de processamento do mapa

### 4.2.2 Algoritmo de *Dijkstra*

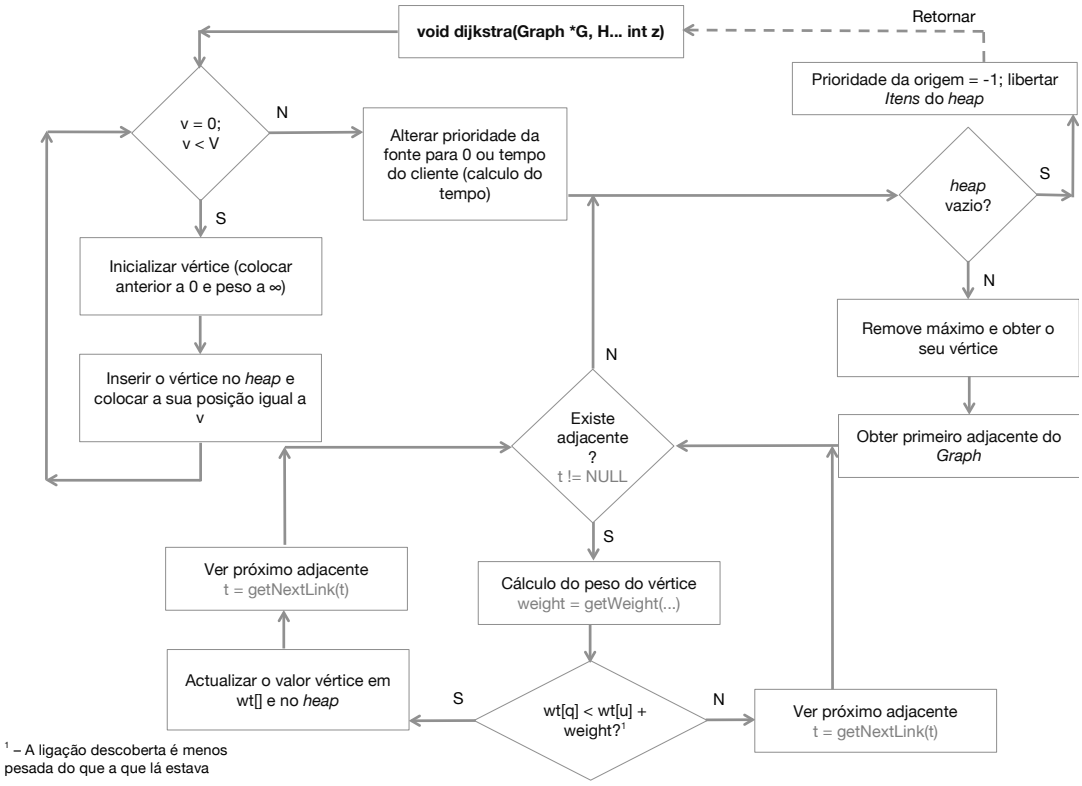


Figure 4: Fluxograma detalhado do algoritmo de Dijkstra

### 4.2.3 Ler informação cliente e determinar solução

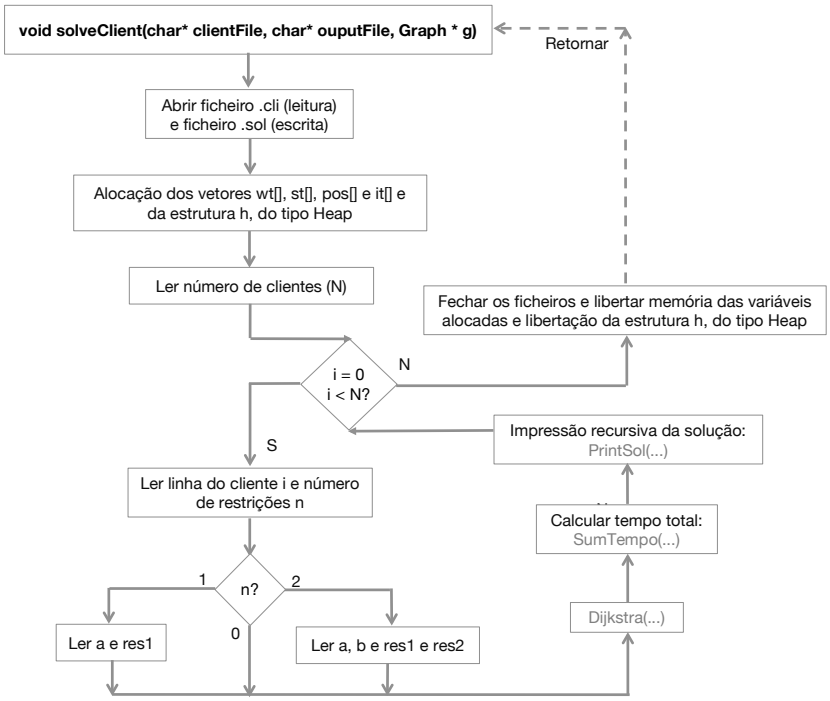


Figure 5: Fluxograma detalhado do algoritmo de determinação da solução

## 4.3 Subsistemas

O programa principal divide-se em 4 subsistemas diferentes, de acordo com os grupos de funções requeridas pelo projeto.

### 4.3.1 *GraphList.c* e *GraphList.h*

O objetivo deste subsistema é criar e desenvolver a estrutura *Graph*, através da implementação de uma lista, com o objetivo de guardar e tratar da informação relativa às ligações no mapa, constituindo um papel fundamental na determinação da solução pretendida.

**Funções relativas à estrutura** *Graph*

```
Graph * GRAPHinit(int v);
```

Aloca espaço para um novo grafo e inicializa-o, com o número de vértices passado à função como argumento.

```
void GRAPHinsertE(Graph * G, int v, int w, Info * toAddInfo);
```

Insere uma nova aresta no grafo G, que liga os vértices $v$ e $w$, com a informação contida na estrutura $toAddInfo$, do tipo $Info$.

```
void GRAPHdestroy(Graph * G, void (*freeInfo) (Info *));
```

Função que liberta todas as estruturas *link* do grafo (cuja informação é libertada pela função $freeInfo$). Liberta ainda o vetor de adjacências e o próprio grafo.

```
int getGraphV(Graph * G);
```

Devolve o número de vértices de um grafo G, passado como argumento à função.

```
int getGraphE(Graph * G);
```

Devolve o número de arestas de um grafo G, passado como argumento à função.

**Funções relativas à estrutura** *link*

```
link *NEW(int v, Info * information, link *next);
```

Aloca espaço para uma nova estrutura do tipo *link*, atribuindo-lhe o índice do vértice de origem, a sua informação (numa estrutura $information$, do tipo $Info$) e um ponteiro para o *link* seguinte. A função retorna o ponteiro para a nova estrutura *link* criada.

```
link * getFirstLink(Graph * G, int vertice);
```

Devolve um ponteiro para o *link* correspondente ao primeiro vértice adjacente ao vértice passado como argumento para a função, no grafo dado.

```
link * getNextLink(link * l);
```

Devolve um ponteiro para o *link* seguinte ao *link* passado como argumento para a função, no grafo dado.

```
int getLinkV(link * l);
```

Devolve o índice do vértice correspondente ao *link* passado como argumento à função.

```
Info * getLinkInfo(link * l);
```

Devolve um ponteiro para uma estrutura $Info$, que contém a informação correspondente ao *link* passado à função como argumento.

**Funções relativas à estrutura** $Info$

```
Info * initInfo(char * transportName, int duracao, int custo, int t1, int tf,
    int p);
```

Aloca espaço para uma nova estrutura $Info$, guardando a informação necessária (argumentos passados à função) nos campos correspondentes e devolve um ponteiro para a nova estrutura criada.

```
int getCusto(Info * information);
```

Devolve o custo correspondente a uma determinada estrutura do tipo $Info$, $information$, passada como argumento à função.

```
int getDuracao(Info * information);
```

Devolve a duração correspondente a uma determinada estrutura do tipo $Info$, $information$, passada como argumento à função.

```
int getMeio(Info * information);
```

Devolve o índice correspondente ao meio de transporte de uma determinada estrutura do tipo $Info$, $information$, passada como argumento à função.

```
int getTempo(Info * information, int tc);
```

Devolve o tempo de espera + o tempo de duração de $information$, uma estrutura do tipo $Info$, passada como argumento à função. O tempo de espera é calculado com base no inteiro tc (tempo em que o cliente chega à cidade), passado também como argumento à função.

```
int getWeight(link * ramo, int criterio, int instante, int A, int z);
```

Devolve o peso (custo ou tempo) da estrutura $ramo$, do tipo $link$, consoante o critério escolhido e as restrições pedidas. Se o peso não respeitar os limites impostos pelas restrições, devolve infinito.

```
char * meioName(Info * information);
```

Converte o índice do meio de transporte de $information$ para o próprio nome do meio, devolvendo a string correspondente.

```
void freeInfo(Info * information);
```

Função que liberta a estrutura $information$, do tipo $Info$.

### 4.3.2 $funcoes.c$ e $funcoes.h$

Neste módulo estão descritas funções auxiliares, que lidam com ficheiros, erros, entre outras. O objetivo deste subsistema é ser a base das funções necessárias para a execução das tarefas básicas do programa.

**Funções tratamento de dados**

```
FILE * openFile(char * filename, char * mode);
```

Abre um ficheiro e devolve um ponteiro para o mesmo.

```
Graph * openMapfile(char* filename);
```

Recebe o nome do ficheiro, abre-o e lê a informação da rede de ligações, guardando-a num grafo, que é devolvido pela função.

```
void solveClient(char* clientFile, char* ouputFile, Graph * g);
```

Função que abre o ficheiro de cliente *clientFile* e gera o resultado de acordo com o mapa passado em *g*, escrevendo-o em *outputFile*, recorrendo à função *dijkstra*, de *heap*.

```
void PrintSol(FILE * f, int c, int wt[], int st[], Info * it[], int * sumCusto,
    int resMax);
```

Imprime a solução dada por *st*[], de acordo com a restrição de custo implicada (*resMax*).

```
void SumTempo(int c, int wt[], int st[], Info * it[], int * sumTempo);
```

Calcula, recursivamente, a soma do tempo para o caminho dado por *st*[].

**Função de erro**

```
void error(char * message);
```

Imprime uma mensagem de erro dada por *message* e sai do programa com o código 2.

### 4.3.3 *heap.c* e *heap.h*

As funções descritas neste módulo relacionam-se com a implementação de uma estrutura de dados do tipo *heap*, utilizada no algoritmo *dijkstra*, também incluído neste ficheiro devido à interligação entre ambas.

### Funções Alocação/Libertação do *heap*

```
Heap *NewHeap(int size, int (*less) (Item, Item));
```

Aloca um *heap* de tamanho *size*, cuja função de comparação entre dois *Item* é *less*, devolvendo o ponteiro para o mesmo.

```
void CleanHeap(Heap * h);
```

Liberta os valores do *heap*, libertando todos os *Item* alojados, não libertando a estrutura $h$.

```
void FreeHeap(Heap *h);
```

Chama a função $CleanHeap(h)$ e liberta a estrutura de dados $h$.

### Função de operação num *heap*

```
void FixUp(Heap * h, int k, int pos[]);
```

Realiza um $FixUp$ a um vértice de $h$, especificado por $k$.

```
void FixDown(Heap * h, int k, int pos[]);
```

Realiza um $FixDown$ a um vértice de $h$, especificado por $k$.

```
int Direct_Insert(Heap * h, Item element);
```

Insere *element* em $h$, não realizando $FixUp$ posteriormente.

```
Item RemoveMax(Heap * h, int pos[]);
```

Remove de $h$ o *Item* com maior prioridade, devolvendo-o (não liberta);

```
int IsHeapEmpty(Heap * h);
```

Devolve 1 se $h$ estiver vazio, 0 caso contrário.

```
void changePriority(Heap * h, int pos, int priority, int positions[]);
```

Modifica a prioridade de um elemento de $h$, na posição *pos*, mudando-a para *priority*.

**Algoritmo de** *Dijkstra*

```
void dijkstra(Graph *G, Heap *h, int s, int st[], int wt[], int pos[], Info *
    it[], int tc, int criterio, int A, int z);
```

Aplica o algoritmo de *dijkstra* a $G$, tendo como origem $s$, guardando os vértices anteriores (caminho) em $st[]$, os pesos em $wt[]$ e as $Info$ respetivas em $it[]$.

### 4.3.4 *Item.c* e *Item.h*

As funções descritas neste módulo relacionam-se com a utilização de uma estrutura abstracta, definida com um *Item*, de forma a manter ao máximo a abstração da implementação do *heap*.

**Funções Alocação/Libertação de** *Item*

```
Item newItem(int vertice, int peso);
```

Aloca um *Item*, dando-lhe como parâmetros *vertice* e *peso*.

```
void freeItem(Item e);
```

Liberta um *Item*.

```
int getVertice(Item e);
```

Retorna o vértice de $e$.

```
int getItemWeight(Item e);
```

Retorna o peso de $e$.

```
int LessNum(Item a, Item b);
```

Função que devolve 1 se $a$ for maior do que $b$; 0 caso contrário.

# 5 Análise Computacional

Antes de conseguir realizar a análise temporal e em termos de memória de partes do projeto, é necessário apresentar as tabelas temporais e de memória das várias estruturas utilizadas nas várias partes.

## 5.1 Estruturas

### 5.1.1 *Graph*

| Processo | Custo Temporal | Custo Memória |
|---|---|---|
| Criação do grafo | O(V) | O(V) |
| Libertação do grafo | O(E) | - |
| Adicionar aresta ao grafo | O(1) | O(1) |
| Acesso à lista de adjacentes | O(1) | - |
| Acesso ao próximo adjacente | O(1) | - |

**Tabela 1.** *Custo temporal e de memória na implementação do Graph*

Onde $V$ corresponde ao número de vértices do grafo e $E$ o número de arestas do grafo.

### 5.1.2 *Info*

| Processo | Custo Temporal | Custo Memória |
|---|---|---|
| Criação da estrutura | O(1) | O(1) |
| Libertação da estrutura | O(1) | - |
| Obter campo da estrutura | O(1) | - |

**Tabela 2.** *Custo temporal e de memória na implementação do Info*

### 5.1.3 *Heap*

| Processo | Custo Temporal | Custo Memória |
|---|---|---|
| Criação do heap | O(1) | O(N) |
| Libertação do heap | O(N) | - |
| Inserir no final do heap | O(1) | O(1) |
| Remover máximo | O($log(n)$) | - |
| Alterar prioridade de um vértice[1] | O($log(n)$) | - |
| Verificar se o heap está vazio | O(1) | - |

**Tabela 3.** *Custo temporal e de memória na implementação do Heap*

Onde $N$ corresponde à capacidade do *heap*, e $n$ corresponde ao número de elementos[1].

## 5.2 Algoritmos

### 5.2.1 Ler mapa e armazenar no *Graph*

Na função **openMapfile()** abre-se um ficheiro, uma operação O(1) por natureza, cria-se um *Graph*, uma operação O(V), e preenche-se com a informação lida.

A ação de preencher o *Graph* corresponde a inserir, sucessivamente, na estrutura as várias arestas dadas pelo ficheiro de entrada. Ora, como foi visto anteriormente (ver 6.1.1 e 6.1.2), alocar uma aresta e adicioná-a a um *Graph* corresponde a uma operação O(1). Dado que se inserem $E$ arestas ao grafo, a complexidade total temporal e de memória é **O(E)** ou **O(V)**, de acordo com a relação entre o número de arestas e vértices.

### 5.2.2 *Dijkstra*

O algoritmo *Dijkstra* implementado constitui uma variante do leccionado nas aulas, já que utiliza uma estrutura do tipo *heap* associado a um **vetor de posições**, *pos*[], que guarda a posição dos vértices no *heap*. Esta associação de fatores faz com que a alteração da prioridade de um vértice adjacente deixe de ser uma operação O(N), passando a ser **O(log(N))**.

No entanto, o algoritmo é constituído por um conjunto de passos sequenciais que influenciam a sua complexidade. Primeiramente, ocorre a inicialização de estruturas e vetores (posição, pesos, entre outros), uma operação O(V), tanto em termos temporais como de memória.

Seguidamente vem a alteração da prioridade da origem, uma operação temporal O(log(V)).

Por fim, entramos no ciclo para determinar os vértices adjacentes ao máximo (neste caso, com o peso mínimo). Todas as funções dentro desta secção, à excepção do **changePriority()**, correspondem a funções cuja complexidade temporal é O(1). No pior caso, todos os vértices são adjacentes uns aos outros, e a ordem pelo qual é percorrido o caminho implica

[1]Custos calculados considerando que é conhecida a posição do vértice no *heap*

que o valor do peso dos mesmos seja atualizado a cada iteração do *while* exterior. Neste caso, a complexidade temporal do *Dijkstra* é **O(E*log(V))**.

Em termos de memória, o algortimo de *Dijkstra* implementado requer apenas $V$ *Itens*, o que corresponde a **O(V)**.

#### 5.2.3 Ler o ficheiro cliente e escrever a solução

Todas as operações necessárias para esta operação encontram-se na função **solveClient()**. A leitura de uma linha do ficheiro corresponde a um processo O(1) temporalmente, sendo que, seguidamente, é aplicado o algoritmo de *Dijkstra*, cuja complexidade é O(E*log(V)), sendo finalmente impresso recursivamente o caminho solução, uma operação cuja ordem de grandeza é O(V) (no pior dos casos, o caminho passa por todos os vértices). Devido ao facto de todas estas operações serem sequenciais, e definindo $C$ como a complexidade final, obtem-se:

$$C = n_c \cdot (O(1) + O(E \cdot log(V)) + O(V)) = n_c \cdot (O(E \cdot log(V)) + O(V)) \quad (1)$$

Em que $n_c$ corresponde ao número de clientes. A complexidade final temporal será definida de acordo com a relação entre $E$ e $V$, podendo ser **O(E *log(V))** ou **O(V)**.

Em termos de memória, a leitura do ficheiro requer 4 tabelas de tamanho $V$ mais um *heap*, pelo que a complexidade final é **O(V)**.

## 6 Exemplo de Funcionamento

Como exemplo ilustrativo do funcionamento do nosso programa, recorremos, em seguida, ao caso particular *enunciado*, com ficheiros de entrada **enunciado.map** e **enunciado.cli**.

### 6.1 Argumentos do main

Para correr o programa, é necessário passar como argumentos do main ambos os ficheiros para leitura, neste caso, **enunciado.map** e **enunciado.cli**.

Na linha de comandos deve aparecer:

```
$./istravel [path]/enunciado.map [path]/enunciado.cli
```

### 6.2 Leitura do ficheiro .map e transferência de informação para o grafo

É utilizada a função **openMapfile**, para ler o ficheiro de entrada que contem o mapa com as ligações possíveis e guardar a sua informação num grafo.

O ficheiro **enunciado.map** contém a seguinte informação:

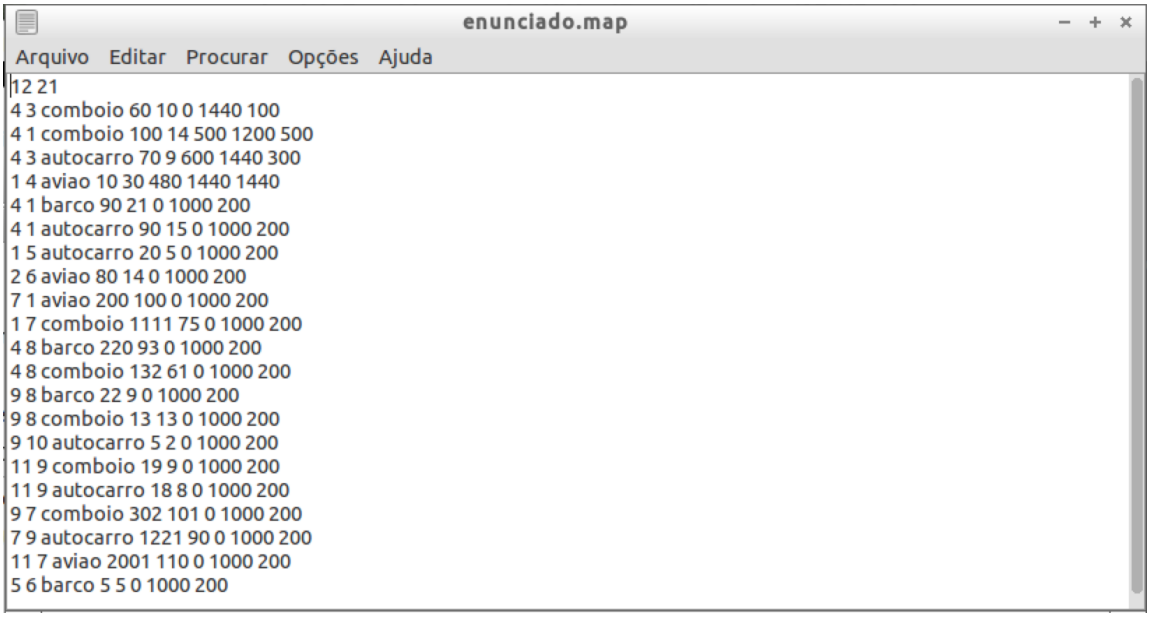


Figure 6: Informação e formato do ficheiro enunciado.map

Em seguida, é inicializado o grafo, criando uma estrutura do tipo *Graph*, recorrendo à função **GRAPHinit**.

Agora, é inicializada uma outra estrutura do tipo $Info$, com a função **initInfo**, onde são armazenadas as informações de cada ligação da rede.

Depois, é inserida a respetiva informação no grafo, aplicando a função **GRAPHinsertE**, que insere, primeiro, uma aresta no grafo por cada ligação lida do ficheiro de entrada e cria, depois, uma estrutura do tipo *link*, onde são guardados os dados da ligação considerada.

Quando é lida a primeira linha do ficheiro, por exemplo, é inserida no grafo uma aresta que liga os vértices 4 e 3. Na realidade, estes vértices têm o valor de 3 e 2, devido a um $shift$ necessário, de modo a evitar a alocação de uma posição de memória adicional (cidades no mapa numeradas de 1 a V). A essa aresta é associado um *link* que contém o vértice de origem, um ponteiro para a estrutura $Info$ com a informação e um ponteiro para a ligação seguinte.

Após a alocação do *Graph* (figura 7), é fechado o ficheiro **enunciado.map**.

### 6.3 Leitura do ficheiro .cli e tratamento de informação

É chamada a função *solveClient*, onde começa por ser lido o ficheiro **enunciado.cli**, recorrendo a um $fscanf$ de todos os campos do mesmo, guardando-os em variáveis.

É feita uma análise do número de restrições requeridas pelo cliente, através de um

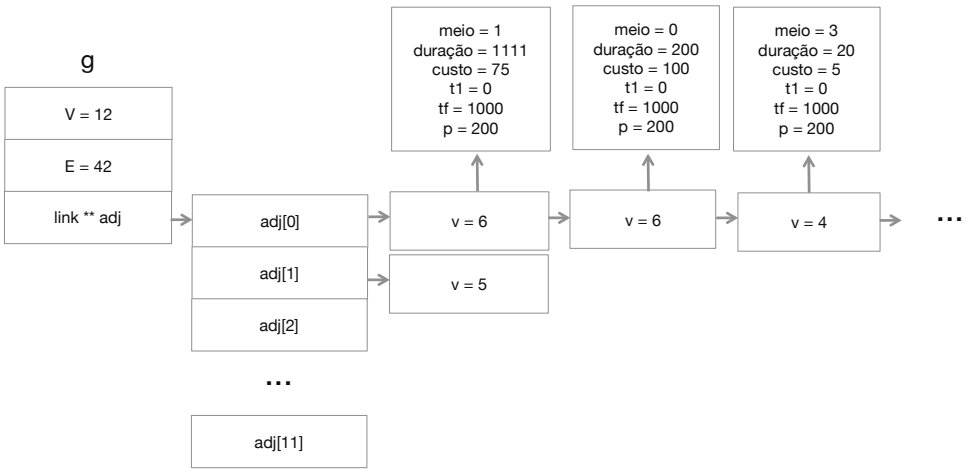


Figure 7: Representação do grafo para o ficheiro enunciado.map

*switch*. Aqui é logo guardado o tipo de restrições, em uma ou duas variáveis (consoante o número), no caso de estas existirem.

No caso do primeiro cliente, aparece:

```
1 1 4 0 tempo 1 A1 aviao
```

, o que mostra que o critério a minimizar é o tempo e que existe uma restrição, cujo objetivo é a exclusão do avião como transporte em qualquer das ligações.

## 6.4 *Dijkstra*

Aqui recorre-se à estrutura *Graph*, com o objetivo de encontrar o caminho mais curto entre dois vértices do grafo.

Para o caso do primeiro cliente, por exemplo, procura-se uma ligação entre os vértices 1 e 4, com o menor tempo possível (o critério a minimizar é o tempo), sem a utilização do

avião em nenhuma das viagens.

Depois de inicializar o vetor de posições a -1, o vetor de peso a infinito e inserir todos os vértices no *heap*, altera-se a prioridade do vértice 1 para 0 (dado que este é a origem e o cliente está disponível a partir do instante 0).

Em seguida, retira-se do *heap* a origem (por ter prioridade máxima) e inicia-se a procura dos seus adjacentes. Para todos os adjacentes ao vértice 1, neste caso, 7 (comboio), 7 (avião), 4 (autocarro), 4 (barco), 4(avião) e 4 (comboio), tenta-se atualizar o peso da ligação. Todos, exceto o 7 (avião) e o 4(avião) vêm o seu peso reduzido, obtendo-se o seguinte *heap*: (considerando a contagem dos vértices a 0)

```
v: 3 w: 90 (autocarro)
v: 11 w: INT_MAX (NULL)
v: 6 w: 1111 (comboio)
v: 1 w: INT_MAX (NULL)
v: 4 w: INT_MAX (NULL)
v: 5 w: INT_MAX (NULL)
v: 2 w: INT_MAX (NULL)
v: 7 w: INT_MAX (NULL)
v: 8 w: INT_MAX (NULL)
v: 9 w: INT_MAX (NULL)
v: 10 w: INT_MAX (NULL)
```

Retira-se, depois, o vértice 4 (3, no *heap*) e repete-se o mesmo processo para os adjacentes.

Finalmente, limpa-se o *heap*, chamando a função **CleanHeap**.

## 6.5 Impressão recursiva da solução

Neste momento, o caminho final já está guardado, embora na ordem inversa, no vetor $st[]$, estando as informações das arestas no vetor $it[]$.

Em primeiro lugar, é chamada a função $SumTempo$, que calcula o tempo total do itinerário calculado, com base nos vetores $st[]$ e $it[]$, devolvidos do algoritmo $Dijkstra$.

Em seguida, é chamada a função $PrintSol$, onde é calculado o custo total da rota e é impressa recursivamente a solução obtida, no ficheiro de saída, para cada cliente.

Obtém-se, neste caso, o ficheiro **enunciado.sol**, com a seguinte forma:

```
1 1 autocarro 4 90 15
2 2 aviao 6 barco 5 autocarro 1 comboio 4 autocarro 3 1020 47
3 -1
4 9 autocarro 10 5 2
5 5 autocarro 1 autocarro 4 comboio 8 comboio 9 853 94
6 2 aviao 6 barco 5 autocarro 1 aviao 4 autocarro 3 1860 63
7 -1
```

```
8 1 comboio 4 600 14
9 -1
```

# 7 Bibliografia


1. "University Assignment." *LaTeX Templates*. N.p., n.d. Web. 21 Nov. 2014. <http://www.latextemplates.com/template/university-assignment-title-page>

2. Damas, Luis. *Linguagem C*. Lisboa: FCA - Editora Informática, 1999.

3. Sedgewick, Robert. *Algorithms in C*. 3rd ed. Boston, Mass.: Addison Wesley, 1998.