Instituto Superior Técnico

Mestrado em Engenharia Eletrotécnica e de Computadores

Algoritmos e Estruturas de Dados

2018/2019 – 2º Ano, 1º Semestre

Relatório do Projeto

Tourist Knights

Grupo Nº 49

Francisco Miguel Velez dos Santos

Nº 90077 e-mail: velez.fmvs@hotmail.com

Joel Neves Pastilha

Nº 88035 e-mail: joelnpastilha@hotmail.com

Docente: **Carlos Bispo**

INDICE

Descrição do problema

Em traços gerais de modo a resumir a informação do enunciado do projeto foi-nos pedido um programa capaz de resolver puzzles estáticos representando cidades, de dimensões retangulares, em que os únicos movimentos admissíveis são deslocamentos em salto de cavalo. Em cada célula das cidades está representado o custo pelo seu acesso e algumas delas são inacessíveis e representadas pelo valor 0.

Como objetivo, o programa tem de elaborar um passeio na cidade, que pode conter vários passos e o custo desse passeio é a soma do custo da entrada em cada célula.

Existem 3 variantes que o programa terá de implementar:

A. Que aborda apenas 2 pontos e é necessário encontrar o passeio de menor custo.
B. Que aborda vários pontos, sendo necessário respeitar a sua ordem de passagem e encontrar o passeio de menor custo passando por todos eles.
C. Também aborda vários pontos, mas sem que seja necessário respeitar a sua ordem, otimizando a dimensão do passeio e sendo de menor custo.

Abordagem ao problema

Inicialmente começamos por utilizar uma matriz complementar há matriz cidade para obtermos índices de cada célula válida, assim como vetor de estrutura com coordenadas e os custos. Afim de testarmos o programa, mudámos a forma como guardávamos estes dados pois usava muita memória, indo além do permitido para validar este programa.

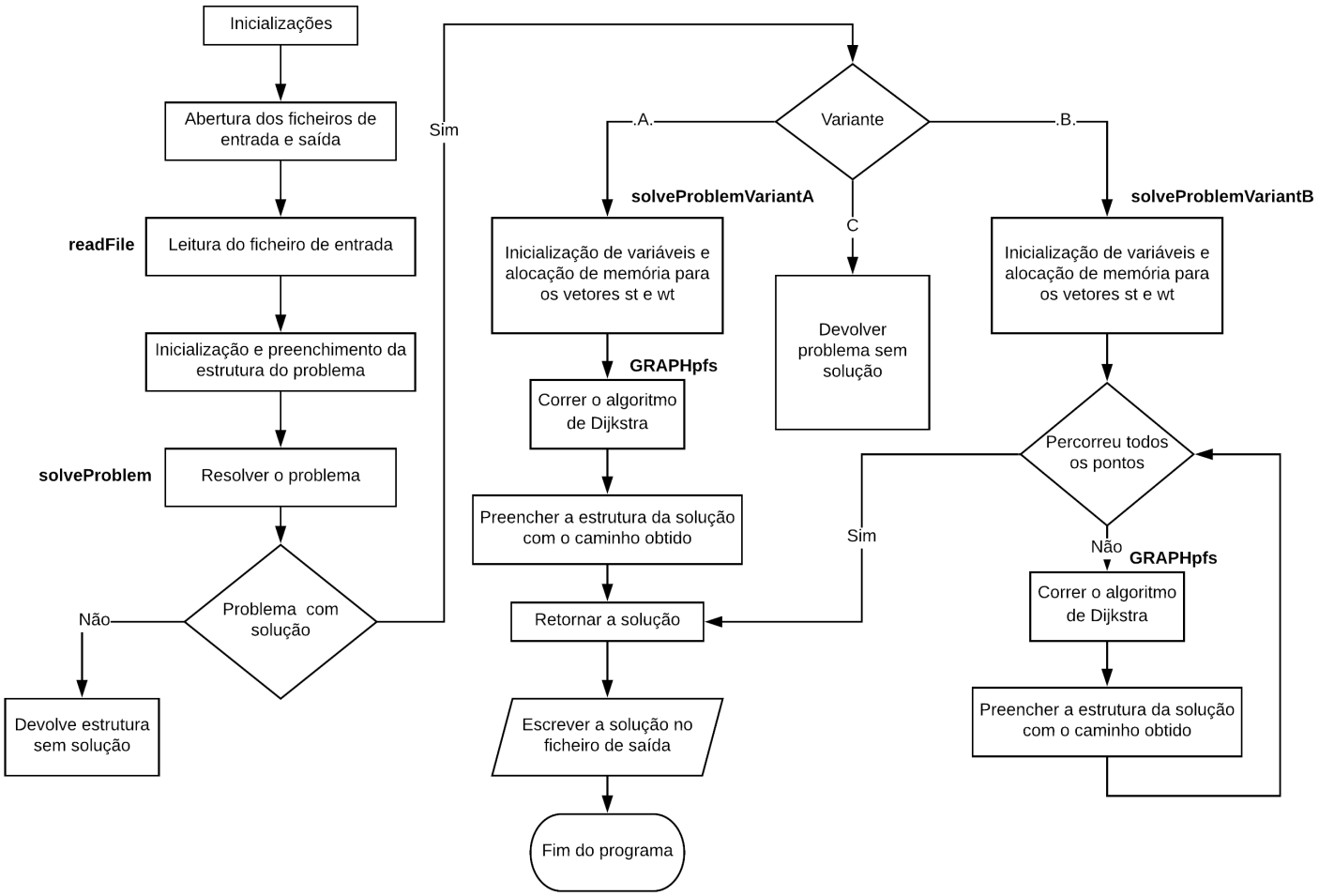
Acabámos por apenas utilizar uma matriz da cidade, como a introduzida no ficheiro. Desta vez implementamos os índices através da fórmula *c*X + Y*, em que c representa o número de colunas e X e Y as coordenadas.

Para depois obtermos as coordenadas utilizamos o processo contrário, *I/c* e *I-(c*(I/c))*, para obter X e Y respetivamente.

Depois pensamos no algoritmo mais eficiente para o programa e acabamos por implementar o algoritmo Dijkstra usando acervos implementados através de uma fila de prioridades por tabela.

Arquitetura do programa

Fluxograma do programa Tourist Knights:



O Fluxograma anterior mostra a arquitetura do programa e condensa a informação sobre as principais funções e os seus objetivos. As funções **solveProblemVariantA** e **solveProblemVariantB** invocam o algoritmo de acordo com a variante do problema.

Descrição das estruturas de dados

point - Foi criada uma estrutura para representar um ponto através das suas coordenadas x, y.

problem - Optou-se por usar uma estrutura para guardar toda a informação de um problema lido do ficheiro. A estrutura problem utiliza uma matriz de inteiros (unsigned short) para guardar a matriz da cidade com os custos de cada célula e um vetor de estruturas (point) para guardar os pontos a ser visitados.

vertice - A estrutura vertice foi usada para transportar o peso e uma identificação de cada vértice. Esta estrutura é útil no uso do algoritmo de Dijkstra onde as únicas informações necessárias de um vértice são o seu peso para a ordenação por prioridade e a sua identificação

cell - Estrutura usada como estrutura auxiliar na solução para guardar as coordenadas e o peso de um determinado vértice.

solution - Optou-se por usar uma estrutura para guardar toda a informação relativa à solução de um problema para isso esta faz uso de um vetor de estruturas auxiliar do tipo cell de modo a guardar todos os pontos do caminho que constitui a solução do problema.

Descrição dos algoritmos

Os principais algoritmos presentes no programa encontram-se nas funções: **GRAPHpfs**, **getAdjacentPoints**, **getAdjIndex** e as funções usadas para calcular um número que identifica o vértice e vice-versa (**getVertexFromCoord**, **getYCoord**, **getXCoord**, **getWtCoord**).

• Converter as coordenadas num vértice e vice-versa (**getVertexFromCoord**, **getYCoord**, **getXCoord**, **getWtCoord**):

Foram implementadas as seguintes equações de modo a poder obter um número único a cada vértice partindo das suas coordenadas (x, y).

Sendo x, y as coordenadas de um determinado vértice, c o número de colunas na matriz da cidade do problema e v o id do vértice, então o número identificador desse vértice é: **v = c * x + y.**

De modo a poder reverter este processo e obter de novo as coordenadas de um vértice foram implementadas equações para obter a coordenada x e a coordenada y de um vértice a partir do seu identificador: **x = v / c, y = v - (c * x).**

• Função **getAdjacentPoints** e **getAdjIndex:**

O objetivo destas funções é de preencher um vetor de inteiros de tamanho máximo 8 (pois este é o número máximo de posições acessíveis através de um único salto de cavalo) com os identificadores dos vértices adjacentes a um vértice v passado como parâmetro de entrada.

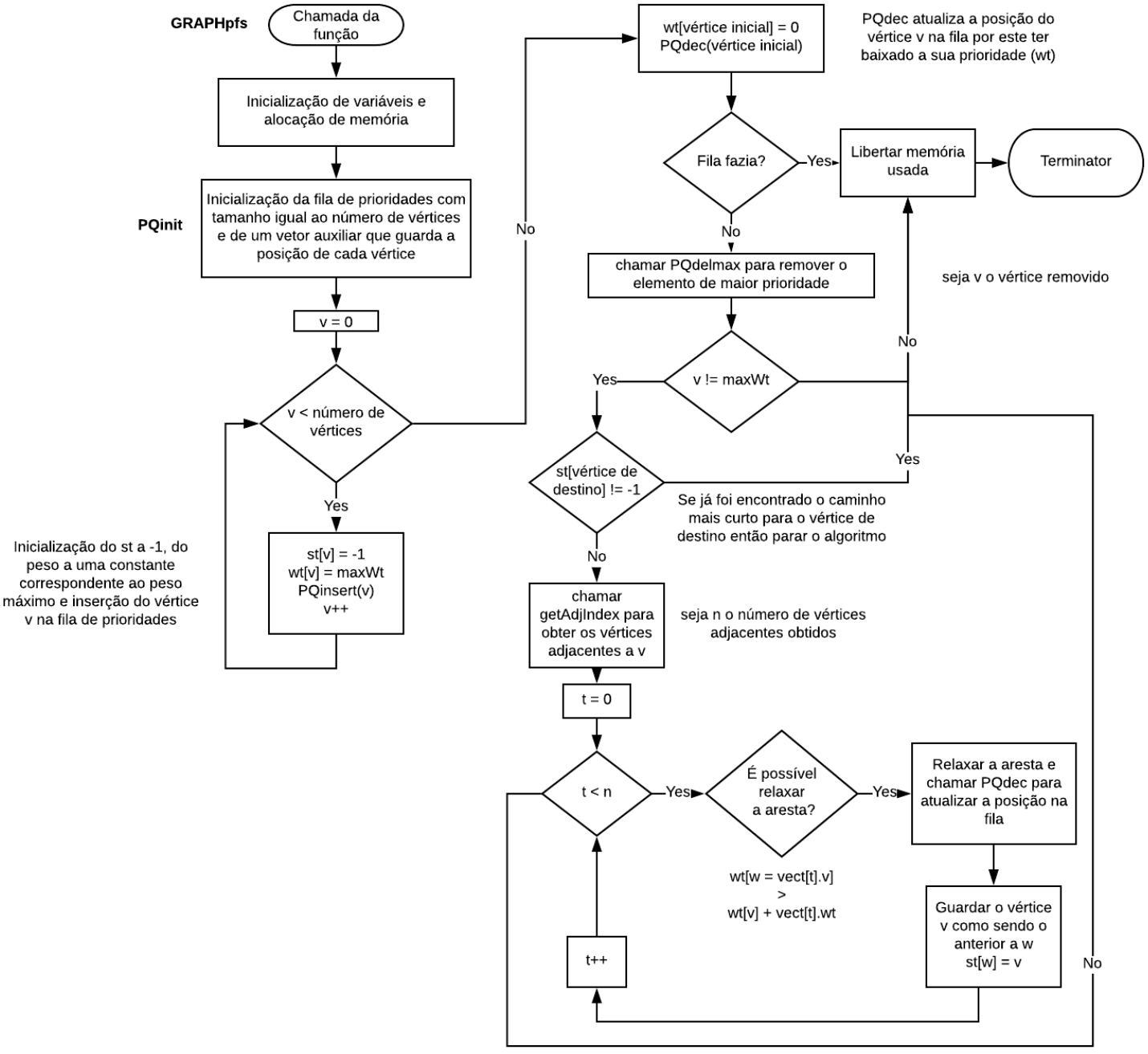
Para atingir este objetivo o algoritmo faz uso de dois vetores auxiliares x, y que contêm as constantes a serem somadas ou subtraídas às coordenadas de v de modo a obter os seus pontos adjacentes. Estas constantes foram obtidas tendo em conta o movimento em l do cavalo.

O funcionamento do algoritmo é o seguinte: Num loop que corre 8 vezes, são somadas as constantes dos vetores x, y às coordenadas do vértice v, verificam-se as novas coordenadas testando se estão dentro do tabuleiro e se o seu custo é diferente de 0 e se sim guarda-se esse vértice num vetor. No final fica preenchido um vetor de estruturas do tipo vertice que guarda todos os vértices válidos atingíveis através de apenas um passo de cavalo.

• Função **GRAPHpfs:**

Esta função consiste na implementação do algoritmo de Dijkstra para o cálculo de caminhos mais curtos. Para tal faz uso de um acervo implementado através de uma tabela. O algoritmo faz também uso de dois vetores auxiliares indexados pelos vértices st e wt que guardam o identificador do vértice anterior e o custo total para um determinado vértice respetivamente. Este constrói uma SPT colocando o vértice fonte na raiz da árvore e construindo-a uma aresta de cada vez.

Fluxograma do algoritmo de Dijkstra:



Análise do funcionamento do algoritmo de Dijkstra

O algoritmo faz uso de uma fila de prioridades de modo a ordenar os vértices com base nos seus pesos e distâncias. Assim foram utilizadas funções auxiliares para manipular esta fila que foi implementada com um acervo. O algoritmo começa for inicializar variáveis auxiliares aos cálculos necessários e alocar memória para o vetor que posteriormente irá ter as informações dos vértices adjacentes a um determinado vértice. De seguida é feita a inicialização da fila e de um vetor auxiliar que guarda a posição de cada vértice, isto é, é feita a sua alocação de memória. Agora são inicializados os vetores st a -1 e wt com uma constante maxWt e são inseridos todos os vértices na fila. Depois o peso do vértice de origem é posto a zero e é atualizada a fila de prioridades devido a esta mudança (o vértice de origem sobe para o topo da fila). Enquanto a fila não estiver vazia retira-se o elemento de maior prioridade e se o seu peso for diferente de maxWt então testa-se o vértice retirado para saber se este corresponde ao vértice de destino e caso este teste passe o algoritmo para pois o caminho está completo, caso não passe, o algoritmo continua. O vetor alocado anteriormente é agora preenchido com os vértices adjacentes ao vértice removido. De seguida, para cada vértice adjacente é testado se a aresta composta pelo vértice removido e pelo vértice adjacente define um novo caminho mais curto do vértice fonte ao vértice adjacente. Para isso basta que (sendo v o vértice removido e w o vértice adjacente) basta testar se wt[w] é maior que wt[v] + v.wt. Se for então guarda-se o novo caminho mais curto no vetor wt[w] e devido a esta alteração é ajustada a posição de w na fila de prioridades. Por último é guardado no vetor st[w] o vértice v, ou seja, para chegar a v é o vértice anterior a w. Quando o algoritmo termina este vetor pode ser usado para recuperar todos os passos do caminho. De modo a obter os passos do primeiro para o último, visto que st requer que se comece no vértice de destino, foi usada uma função que é chamada recursivamente, permitindo escrever no ficheiro de saída todos os pontos de um caminho pela ordem correta. Para obter o custo total do passeio basta aceder à posição do vetor wt indexada pelo vértice de destino. Obtêm-se assim todos os dados necessários à resolução dos problemas propostos no que toca ao cálculo do caminho mais curto de um único vértice de partida a um único vértice de destino. Todos os modos A, B e C têm como base este algoritmo.

Análise dos requisitos computacionais

A fim de reduzir ao máximo a complexidade do programa foram tomadas várias decisões no que toca à escolha das várias estruturas de dados usadas. A seguir é mostrado em detalhe o porquê dessas decisões.

Nas funções responsáveis pela conversão das coordenadas de um vértice no seu id e vice-versa a complexidade é semelhante e igual a O(1), isto pois cada função utiliza apenas uma fórmula com um número limitado de operações não dependente do número de vértices ou da sua posição na matriz.

Inicialmente tinha sido pensado em utilizar uma matriz e um vetor auxiliares para guardarem o id de cada vértice e as suas coordenadas respetivamente, no entanto esta abordagem era bastante exigente no consumo de memória, assim optou-se pela abordagem acima descrita que resolve ambos os problemas.

Matriz de custos da cidade

Uma das operações mais repetidas no programa era de acesso ao custo de um determinado ponto da cidade, assim optou-se por guardar toda a informação do custo dos pontos numa matriz indexada pelas suas coordenadas x, y. Pois, assim este acesso era feito de forma direta com complexidade O(1). Aqui, a memória usada é proporcional ao número de pontos da cidade.

Vetor de pontos adjacentes

Semelhante à matriz de custo da cidade, este é um vetor bastante utilizado pelo algoritmo de Dijkstra, logo é necessário que a sua complexidade seja baixa, assim foi utilizado um vetor para permitir o rápido acesso aos valores dos pontos adjacentes. Mantendo assim a complexidade O(1).

Grafo

Devido ao modo como o problema está descrito e, por ser possível para qualquer ponto da cidade calcular matematicamente todos os seus pontos adjacentes, então foi definido um grafo implícito pelo que não foi necessário implementar quaisquer funções auxiliares para gratos e alocar memória para uma eventual lista de adjacências. Isto permitiu reduzir tanto a complexidade geral do programa como o seu uso de memória.

Acervo

Tendo em conta a aplicação do acervo no algoritmo de Dijkstra em que as principais operações são as de inserção, remoção do elemento máximo e modificação e tendo também em conta as alternativas ao acervo, sendo estas uma tabela ordenada e uma tabela não ordenada conclui-se que a opção que mais permite reduzir a complexidade é efetivamente o acervo. Assim foi usado um acervo implementado por uma tabela para a implementação da fila de prioridades no algoritmo supra descrito.

Conclui-se então que a complexidade do acervo é de O(lg (n)) para as operações de inserção, remoção do elemento máximo e para a modificação da prioridade de um determinado elemento. Quanto ao uso de memória, o tamanho do acervo é proporcional ao número de pontos da cidade (número de colunas da matriz multiplicado pelo número de linhas da matriz). Para conseguir implementar o acervo mantendo esta complexidade foi criado um vetor auxiliar com o mesmo tamanho que o acervo com o objetivo de guardar a posição no acervo de cada vértice, assim este vetor indexado pelo valor do vértice permite obter reduzir a complexidade quando é preciso realizar operações sobre um vértice do qual só se sabe o seu valor como é o caso do algoritmo de Dijkstra.

Algoritmo de Dijkstra

Este algoritmo foi escolhido para este problema pois calcula SPT's, ou seja, caminhos mais curtos a partir de um determinado vértice. Como dito anteriormente o algoritmo faz uso de um acervo para a sua implementação. Tendo implementado o algoritmo na sua forma mais eficiente utilizando acervos a sua complexidade é de O(E lg V) em que E é o número de arestas e V o número de vértices.

Descrição dos subsistemas

O programa por nós implementado tem o subsistema **problem,** contituido por ***problem.c*** e ***problem.h*** onde estão guardadas as funções que:

- Guardam os dados dos ficheiros nas estruturas e alocam a sua memória
- Permitem a realização dos problemas em todas as variantes
- Efetuam cálculos de variáveis e coordenadas
- Libertam memória
- Preenchem as soluções.

As estruturas de dados estão definidas no ficheiro ***problem.h***. Existe outro subsistema **file** constituído por ***file.c*** e ***file.h*** que contém as funções diretamente relacionadas com a leitura ou manipulação de dados em ficheiros de texto. Existem ainda os ficheiros ***utils.c utils.h*** que contêm funções auxiliares para fazer verificações.

problem.c

problem **allocateProblem**(problem pb)

Função para alocar memória para o número de pontos propostos pelo problema e da matriz do mapa das cidades.

void **freeProblem**(problem pb)

Função para libertar o problema do tipo problem

int **checkIfProblemHasSolution**(problem pb)

Função para verificar se o problema tem solução através das regras da quantidade de pontos de cada variante, se os pontos pertencem ao mapa de cidades e se os pontos são cidades válidas.

int **getWtCoord**(int v, problem pb)

Função para obter o custo através do índice.

int **getXCoord**(int v, problem pb)

Função para obter a coordenada x através do índice.

int **getYCoord**(int v, problem pb)

Função para obter a coordenada y através do índice.

void **getAdjacentPoints**(int p, int *points, problem pb, int *n)

Função para obter os pontos adjacentes de um índice.

Retorna um vetor com os índices válidos.

int **getVertexFromCoord**(point pt, problem pb)

Função para obter o índice através das coordenadas.

void **getAdjIndex**(problem pb, int v, int *v_adj, vertice* vect)

Função para preencher um vetor do tipo vértice com índice e custo.

void **fillSolution**(int *st, int a, solution *sol)

Função recursiva para obter os todos os índices do passeio.

void **fillSolutionPoints**(problem pb, int *st, int a, solution *sol, int *aux)

Função recursiva para preencher a estrutura solution com coordenadas e custos.

solution **solveProblemVariantA**(problem pb)

Função para resolver o problema de variante A.

solution **solveProblemVariantB**(problem pb)

Função para resolver o problema de variante B.

solution **solveProblemVariantC**(problem pb)

Função para resolver o problema de variante C.

solution **solveProblem**(problem pb)

Função para verificar qual a variante do problema e invocar as funções para o resolver.

void **readFile**(FILE* fp, FILE* fSol)

Função para ler o ficheiro, alocar memória e preencher as estruturas de dados.

File.c

Funções de abertura de ficheiros

void **saveSolution**(solution sol, FILE* fSol)

Função para escrever a solução no ficheiro de saída.

FILE* **openFile**(char* fileName, char mode, char* extension)

Função para abrir ficheiros para ler ou escrever dependendo da extension.

heap.c

void **GRAPHpfs**(int s, int st[], unsigned short wt[], int vd, problem pb)

Função onde se desenvolve o algoritmo da procura do passeio.

Funções do algoritmo

- BOOL **PQempty**()
- void **PQdec**(int s)
- void **PQinit**(unsigned Size)
- void **PQinsert**(int I)
- int **PQdelmax**()
- void **FixUp**(unsigned int Heap[], int Idx)

- void **FixDown**(unsigned int Heap[], int Idx, int N)
- void **cleanUp**(vertice* vect)

utils.c

void **checkMalloc**(void* pointer)

Função para verificar as alocações de memória.

void **usage**()

Função para verificar o formato do ficheiro introduzido.

int **getVectMin**(int* vect, int n)

Função para devolver o valor mínimo de um vetor.

Funcionamento do programa

Inicialmente o programa passava em apenas nos testes da variante A, mas depois de identificado o problema e fazendo ajustes no código do algoritmo o programa passou também nos testes da variante B.

Ainda assim havia problemas com a aplicação da extensão ".walks" e após identificar o problema e alterar o código da aproveitação do nome do ficheiro sem a extensão ".cities", conseguimos passar em 14 testes dos 20 usados na avaliação através do verificador Mooshak.

Afim da última submissão, havia um teste em que foi ultrapassado o limite de tempo e não nos foi possível realizar a parte da variante C, por falta de tempo devido a erros contínuos nas outras variantes.

Exemplo

Ficheiro de entrada:

```
10 11 A 2
0 0
3 6
1 0 2 3 2 3 2 3 2 3 1
3 2 1 2 3 3 1 2 3 2 1
4 3 2 1 2 3 4 5 0 3 1
1 6 2 3 2 3 2 3 2 3 1
3 2 1 2 3 0 1 2 3 2 1
4 3 2 1 2 3 4 5 3 3 1
1 2 0 3 2 3 2 3 2 3 1
3 2 1 2 3 3 1 2 3 2 1
4 3 2 1 2 3 4 5 3 0 1
1 7 2 3 2 3 2 3 2 3 1
```

1 – Ler o número de linhas e colunas, a variante e o número de pontos.

2 – Alocação de memória para a matriz e armazenamento dos dados lidos do ficheiro através da função **readFile**.

3 – É verificada a variante e invocado o **solveProblemVariantA.**
4 – São alocados os vetores st e wt, que indicam se a célula foi visitada e o custo do passeio total até ao ponto v, respetivamente.

1 0 2 3 **2** 3 2 3 2 3 1
3 2 **1** 2 3 3 1 2 3 2 1
4 3 2 1 **2** 3 4 5 0 3 1
1 **6** 2 **3** 2 3 2 3 2 3 1
3 2 1 2 3 0 1 2 3 2 1
4 3 2 1 2 3 4 5 3 3 1
1 2 0 3 2 3 2 3 2 3 1
3 2 1 2 3 3 1 2 3 2 1
4 3 2 1 2 3 4 5 3 0 1
1 7 2 3 2 3 2 3 2 3 1

5 – O algoritmo inicialmente encontra apenas um ponto válido (1,2).
6 – De seguida são encontrados 5 pontos adjacentes e, neste caso, os pontos com custo menor vão ser testados.

1 0 2 **3** ~~2~~ **3** 2 3 2 3 1
3 2 1 2 3 3 **1** 2 3 2 1
4 3 2 1 **2** 3 4 5 0 3 1
1 6 **2** 3 2 3 **2** 3 2 3 1
3 2 1 ***2*** 3 0 1 2 3 2 1
4 3 2 1 2 3 4 5 3 3 1
1 2 0 3 2 3 2 3 2 3 1
3 2 1 2 3 3 1 2 3 2 1
4 3 2 1 2 3 4 5 3 0 1
1 7 2 3 2 3 2 3 2 3 1

7 – Depois de testar ambos os valores de menor custo, é encontrado o ponto de destino como filho do ponto (2,4).
8 – Uma vez que o ponto a ser testado é o ponto de destino, o algoritmo é interrompido.
9 – É agora preenchida a solução com os pontos presentes em st representados por índices pelas funções recursivas **fillSolution** e **fillSolutionPoints**.
10 – Por último a função **saveSolution** escreve no ficheiro de saída e a memória é libertada.

10 11 A 2 5 3
1 2 1
2 4 2
3 6 2