TÉCNICO LISBOA

Algoritmos e Estrutura de Dados

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

2018-2019 – 2º Ano - 1º Semestre

Relatório do Projeto

TOURIST KNIGHTS

Grupo nº 10:

Rodrigo Muchagato Figueiredo

nº 90185 e-mail: rodrigofigueiredo@tecnico.ulisboat.pt

Rafael André Alves Cordeiro

Nº 90171 e-mail: rafael.andre.alves@tecnico.ulisboa.pt

Docente: Carlos Bispo

ÍNDICE

DESCRIÇÃO DO PROGRAMA

Numa abordagem mais geral do enunciado, de modo a perspetivar o problema com maior facilidade, foi proposto a realizar um programa de resolução de puzzles estáticos, representando cidades, tendo como movimentos possíveis os saltos de cavalo da peça de xadrez. Objetivo principal consiste em calcular o caminho de custo mínimo de um itinerário proposto pelo utilizador, este custo varia com o custo das células que percorre.

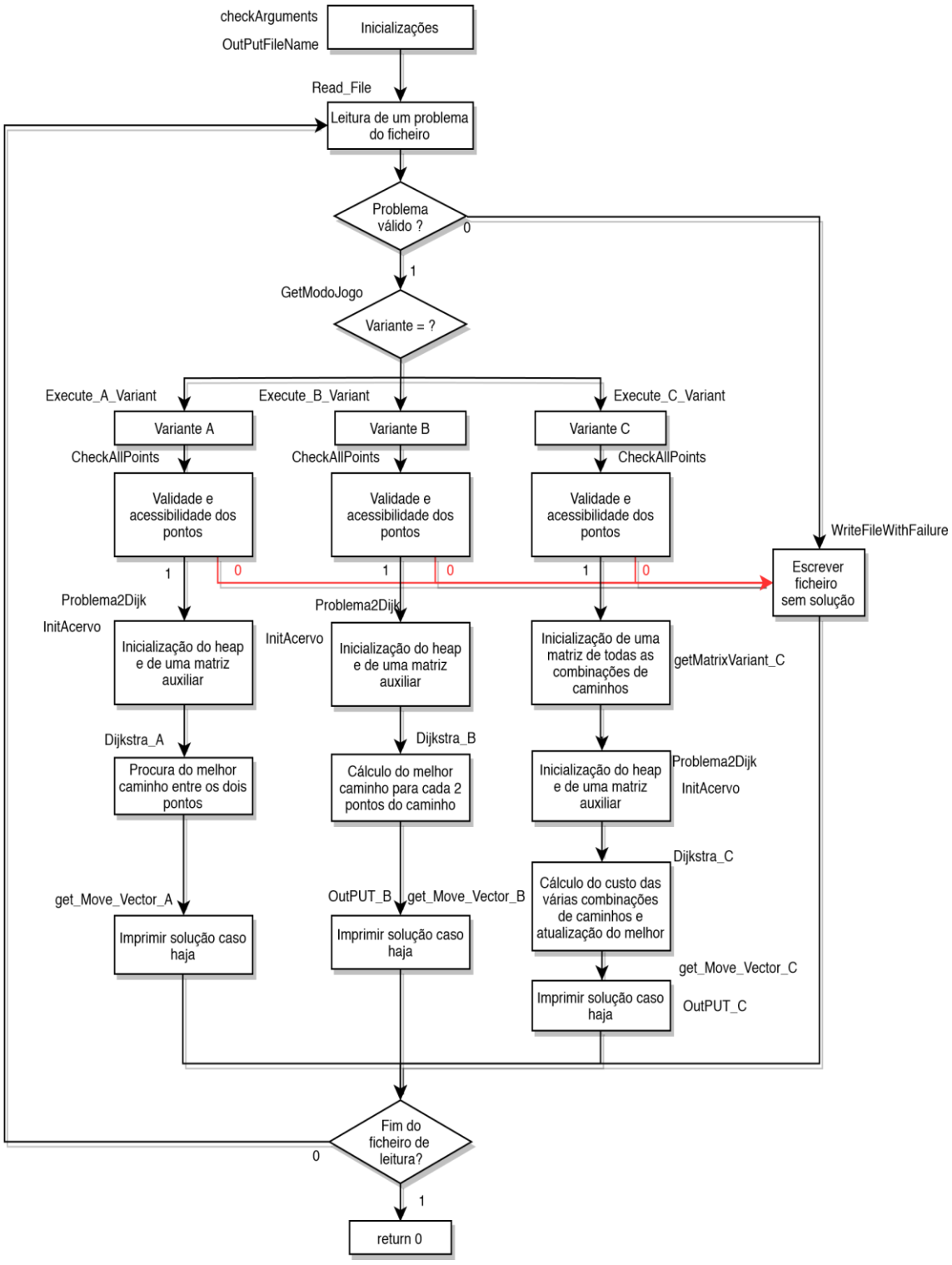
O utilizador poderá escolher uma de três variantes, que foram classificadas como modo A, B e C. A abordagem A do problema analisa exclusivamente 2 pontos, determinando, consequentemente, o caminho de menor custo que une os pontos propostos para análise. Na abordagem B, o utilizador escolhe um determinado conjunto de pontos, com o objetivo de obter o caminho de custo total mínimo que une todos os pontos, na ordem colocada inicialmente. Por último, a abordagem C do problema consiste numa variante da abordagem B, contudo a ordem que une todos os pontos do caminho é aquela que oferece um custo mínimo, sendo, por isso, irrelevante a ordem dos pontos que o utilizador introduz.

ABORDAGEM AO PROBLEMA

O problema principal a resolver consiste em calcular o caminho mais curto, de uma das abordagens mencionadas anteriormente, num tempo bastante reduzido. Assim, começou-se por copiar os dados do ficheiro proposto para uma matriz, apenas se os pontos forem todos válidos e se todas as condições necessárias se verificarem, para prevenir a alocação indevida de memória. A partir daqui, foi primeiramente abordado a variante A, visto que é a base das restantes, tendo como base o Algoritmo de Dijkstra (onde através da operação de relaxação do caminho, verifica se existe um caminho mais curto do que algum dos já conhecidos) associado a um acervo de pontos, tendo como chave o menor custo. Para a variante B, foi apenas necessário correr a variante A para vários pontos, em vez de serem apenas 2, bem como mais algumas especificações relacionadas com a apresentação do ficheiro de saída.

Uma descrição mais pormenorizada deste algoritmo encontra-se especificada mais à frente.

ARQUITETURA DO PROGRAMA



DESCRIÇÃO DAS ESTRUTURAS DE DADOS

Estrutura	Descrição
Struct Problema	Estrutura principal do programa, aqui é armazenada a grande maioria da informação relevante ao problema proposta, tal como uma struct caminho (que tem os pontos turísticos do ficheiro), um tabuleiro (matriz proposta pelo ficheiro), bem como o modo de jogo. Assim, como é natural, só existirá uma struct Problema no programa.
Struct caminho	Geralmente é utilizada sozinha, para descrever um caminho entre 2 ou mais pontos, ou, mais frequentemente, como uma matriz de estruturas, com tamanho $N \times N$ (N corresponde ao número de pontos introduzidos), para poder armazenar os caminhos que unem 2 pontos, evitando, deste modo, a execução do Dijkstra, que poderia demorar bastante tempo. A estrutura armazena o número de pontos de um dado caminho; um vetor de pontos, que guarda todos os pontos/células de um dado percurso e o custo total do caminho. A struct caminho é usada várias vezes, tal como foi mencionado, mas de um modo geral, o tamanho é variável, podem ser utilizadas individualmente para descrever um caminho entre A e B ou podem ter tamanho Num_pontos_turisticos × Num_pontos_turísticos, se for utilizada uma matriz.
Struct Dijk_Struct	Tal como o nome indica o nome, esta estrutura é apenas utilizada para obter uma matriz de estruturas, sendo, por isso, única no programa todo, com o tamanho da matriz proposta pelo ficheiro, com o objetivo de ligar dos os pontos com relação

	Pai-Filho, com maior facilidade no algoritmo de Dijkstra. A estrutura é constituída por uma ligação ao Pai e por um custo acumulado até esse ponto.
Struct acervoStruct	Estrutura que contém todas as informações essenciais para o funcionamento do acervo, visto que contém um vetor de pontos (para o armazenamento do acervo de pontos), um inteiro free que determina quantos pontos já foram inseridos na heap e uma matriz de indexação do acervo, com tamanho idêntico à matriz proposta no enunciado, que permite o seu acesso rápido e de forma eficaz. O acervo vai ter o tamanho da matriz proposta pelo ficheiro.
Struct tabuleiro_t	Estrutura que armazena a matriz equivalente do tabuleiro proposto pelo ficheiro, bem como as suas dimensões, assim só existe uma estrutura destas no programa inteiro.

A estruturas acima mencionadas descrevem, de um modo geral, as estruturas utilizadas, bem como o modo e tamanho que eram utilizadas. Para além destas, utilizámos as seguintes macros:

Função	Descrição
lessPri (A, B) (A > B)	Recebe dois inteiros e analisa qual é o maior. Geralmente é utilizada no FixUp e FixDown, para comparar a chave/custo do Pai e Filho.
exch (A, B) {Item t = A; A = B; B = t;}	Recebe duas posições e troca os seus valores. Geralmente utilizada no FixUp, FixDown e no HeapDeleteMaxPoint.

DESCRIÇÃO DOS ALGORITMOS

Os principais algoritmos implementados que foram lecionados nas aulas foram o **Dijkstra** e as **Priority Queues**, mais concretamente associadas a **Acervos.**

Neste capítulo, é feita uma explicação geral sobre os algoritmos utilizados e como foram utilizadas, mais à frente é explicado, com maior detalhe como foram implementados e usados para cada variante do programa.

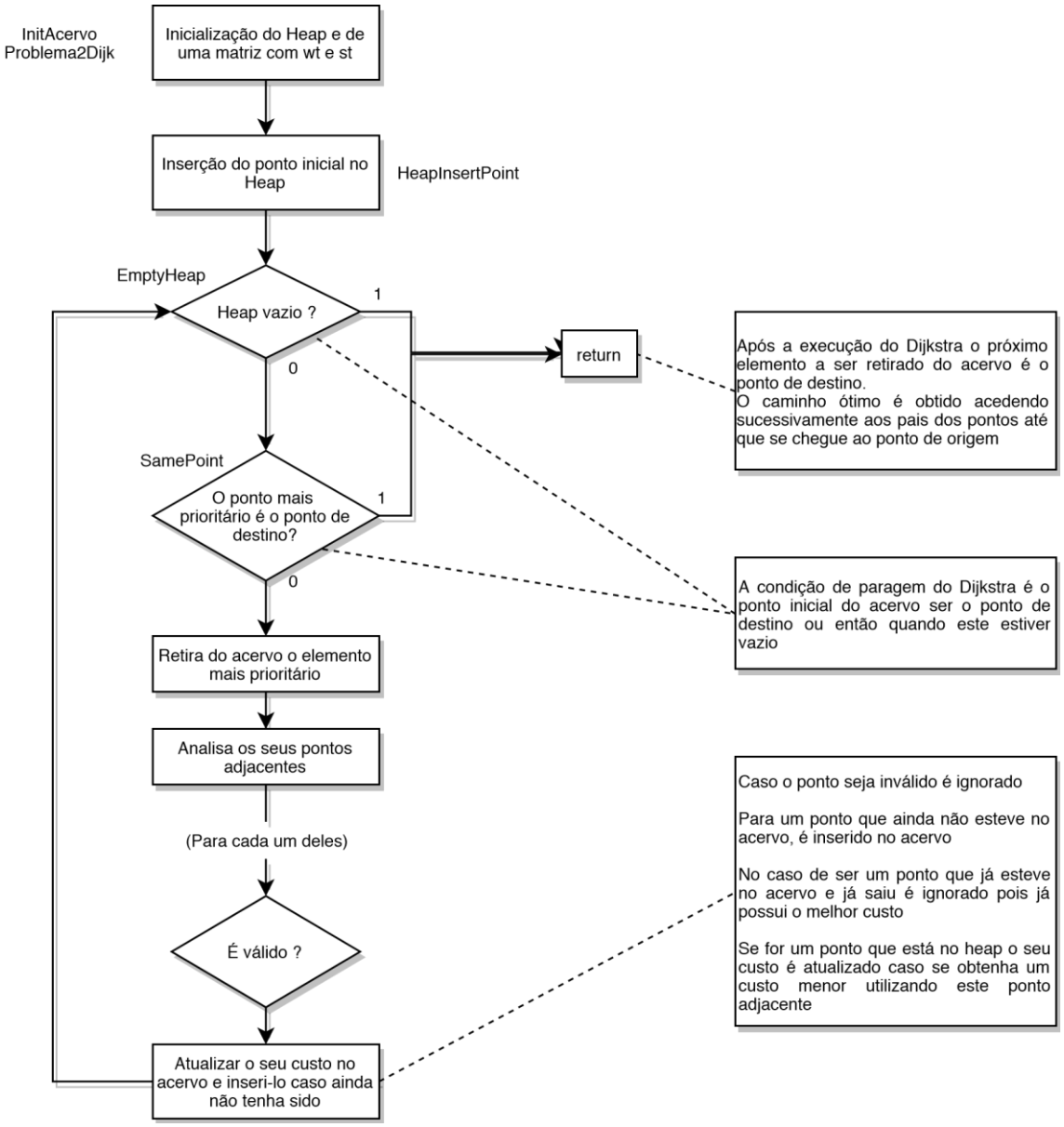
DIJKSTRA

O algoritmo de Dijkstra foi utilizado para poder obter o caminho de menor custo acumulado de um caminho de 2 ou mais pontos.

A execução deste algoritmo foi sempre efetuada com base em gastar o menor tempo possível, assim utilizou-se em combinação com um acervo indexado por uma matriz, para permitir uma procura mais rápida, por exemplo, no caso da prioridade ser alterada. A escolha da utilização foi óbvia, pois as operações de remoção e de inserção têm uma eficiência logarítmica.

Para executar este algoritmo utilizámos uma matriz de indexação do acervo, que ia analisando 3 casos possíveis, associados ao valor contido na matriz na posição desejada: -1 se o ponto não estiver no acervo, logo a inserção é direta; -2 se o ponto já foi retirado do acervo e um valor superior a -1 que indica a sua posição. Para além disto, é usada uma matriz de tamanho idêntico com a matriz original que serve para estabelecer as relações de pai/filho e armazenar o custo acumulado do caminho até esse ponto. Por conseguinte, esta matriz é inicializada com um custo inicial infinito e uma ligação pai/filho ao ponto inicial, sendo constantemente atualizada à medida que são inseridos pontos no acervo e no caso de a prioridade ser alterada.

O algoritmo vai assim realizando um ciclo até o acervo estar vazio (não existe caminho) ou até encontrar o ponto de destino, permanecendo constantemente a analisar os pontos de Salto de Cavalo possíveis e a remover o ponto mais prioritário até ao momento, tal como é indicado no fluxograma apresentado de seguida.



ACERVO:

Face ao programa pedido a utilização de acervos como forma de organização de pontos fez todo o sentido, pois a eficiência dos acervos é logarítmica e reduziu-se bastante em memória, assim foi possível reduzir substancialmente o tempo gasto na gestão de dados/informação. Tal como foi dito anteriormente, foi utilizado um acervo indexado, para permitir uma procura rápido, por exemplo, no caso de existirem pontos repetidos, esta indexação foi

implementada com o auxílio de uma matriz que contém -2(ponto retirado do acervo), -1(ponto ainda não inserido),>-1(posição no acervo).

O acervo foi implementado tendo como chave o menor custo total acumulado até esse ponto, estando contido na estrutura **acervoStruct** que contém o acervo (representado em tabela, como é habitual).

DESCRIÇÃO DAS VARIANTES:

VARIANTE A:

A função **EXECUTE_A_VARIANT** é responsável pela execução da Variante A, onde vai executar o algoritmo de Dijkstra para 2 pontos apenas. Tal como foi mencionado anteriormente, na explicação do Dijkstra, a variante A consiste na execução do Dijkstra associado a uma acervo para encontrar o caminho com menor custo, assim, se ambos os pontos turísticos foram válidos, é executada a função **DijkstraAlgoritm_A** que vai consecutivamente, removendo o elemento mais prioritário(menor custo acumulado) e analisando os seus pontos adjacentes, adicionando-os ao acervo, se ainda não estiverem lá ou alterando a sua prioridade caso o custo acumulado de um ponto adjacente seja superior ao custo até esse ponto + o custo da célula, isto é, ocorre uma relaxação do caminho. O ciclo termina quando o acervo está vazio ou quando se chega ao ponto de destino.

VARIANTE B:

A função encarregue de executar a variante B denomina-se de EXECUTE_B_VARIANT, que tem como estrutura principal a variante A, pois realiza a variante A para calcular os caminhos entre os pontos turísticos propostos.

Por exemplo, para um caminho ABCD, a nossa implementação vai usar o Dijkstra da seguinte forma (cada caixa representa a utilização do Dijkstra para calcular um caminho).

VARIANTE C:

A função encarregue de executar a variante C denomina-se de EXECUTE_C_VARIANT, que se os pontos turísticos forem todos válidos vai inicializar um vetor de combinações(se tiver existirem 4 pontos turísticos, é inicializado com 1-2-3) que vai sendo constantemente alterada a partir de uma função recursiva que vai continuamente criando combinações e chamando a função C_for_one_path. A função anteriormente mencionada implementa uma matriz de caminhos, que vai guardando os caminhos de AB, BC, etc…, bem como os seu simétricos para evitar a execução do algoritmo de Dijkstra, poupando tempo de execução. Para além disso, implementa uma variante de uma Branch and Bound, que antes de fazer qualquer operação analisa se o caminho já é mais caro que o melhor até ao momento, por outras palavras, se tiver o caminho ABCD e se depois de calcular AB e BC o caminho já custo mais que o meu melhor caminho calculado, o programa deixa de executar operações para essa combinação, não calculando o CD. Assim, a execução do algoritmo C baseia-se na figura em baixo, onde caso uma das condições não se verifique, não vê as restantes:

DESCRIÇÃO DOS SUBSISTEMAS:

Main	Local onde é lida a matriz apresentada pelo ficheiro e análise das condições do programa. Para além disso, é também aqui onde são chamadas as funções que dão início a uma determinada variante.
Dijkstra	Subsistema que contém todas as componentes necessárias para a execução do algoritmo de Dijkstra mencionado e explicado anteriormente, tal como o Dijkstra das várias variantes, as funções de análise de pontos adjacentes, etc…
Tabuleiro	Subsistema que contém a estrutura tabuleiro oculta, onde armazena uma matriz idêntica à matriz original. Para além disso, neste subsistema estão também situadas todas as operações intrínsecas à utilização desta estrutura.
Points	Tal como o nome sugere, o ficheiro contém todas as operações relacionadas com pontos, como para obter coordenadas na matriz, ou para analisar se dois pontos são iguais.
Util	Contém funções úteis e abrangentes aos restantes subsistemas, como por exemplo, o Checked_Malloc para alocar memória, ou até a função OutPutFileName que altera o nome do ficheiro para o nome de output mais indicado.
Moves	Subsistema que contém como estruturas ocultas, os caminhos e a estruturas Problema, assim tudo o que envolva operações com caminhos, como as operações para inserir e remover na matriz de caminhos, para otimização da parte C, ou, por exemplo, as operações para gerar combinações recursivamente. Para além disto, é também neste subsistema onde são armazenadas as funções que executam as variantes A, B e C.
Acervo	Subsistema que contém todas as funções relacionadas com as operações com acervos, sejam elas de FixUp, FixDown, etc…

EXEMPLO DE FUNCIONAMENTO DO PROGRAMA

VARIANTE A

Tomando o seguinte ficheiro de leitura:

```
3 17 A 2
2 0
2 8
0 0 0 0  0  0  0 0 2 3 4 4 2 6 5 77 3
6 3 2 1  7 10 20 4 8 2 1 1 2 3 4 11 3
2 9 6 17 3 23 12 6 7 6 0 0 0 0 0  0 0
```

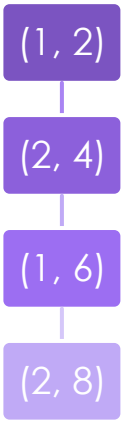
A função Read_File é responsável pela alocação de memória para a estrutura Problema que guarda no campo *modo_jogo* o caracter '*A*', um ponteiro para uma estrutura designada *tabuleiro* e contem uma estrutura que é um *caminho*.

Dentro da estrutura *tabuleiro*, no campo *tab* está um apontador para a cidade lida sob a forma de matriz, nos campos *size_x* e *size_y* estão as dimensões em *x* e em *y* que são respetivamente *17* e *3*. Ainda dentro da estrutura principal existe outro campo que contem um *caminho*. Dentro deste, no campo *num_pontos* vai ser guardado o número de pontos do caminho lido, que neste caso é 2 e um vetor de pontos, *points*, que contém (2,0) na posição 0 e (2,8) na posição 1.

Com o problema lido o programa verificar se os pontos são ambos acessíveis e se estão dentro dos limites do tabuleiro. O ponto inicial tem custo 2 e o ponto final tem custo 7 e ambos estão dentro dos limites por isso o problema tem potencial para ter solução.

O próximo passo a ser executado é o algoritmo Dijkstra que já foi explicado anteriormente. Após a execução deste algoritmo o ponto a ser retirado do acervo é o (2,8), pois neste caso o problema tem solução.

Acedendo aos respetivos pais o caminho é construído e guardado na estrutura *caminho*, vetor de pontos *points*. O número de pontos é incrementado na variável *num_pontos* dentro de uma estrutura caminho, bem como o custo total do caminho ao qual vai ser somado o custo de cada pai para cada ponto. Este custo também se encontra guardado na estrutura *caminho* no variável *custo_total*.



Começando no ponto (2,8) o seu pai é o (1,6) e este tem como pai o (2,4) que por fim tem o (1,2) como pai. A variável de custo vai tomar o valor de $7+2+3+20$ que são os custos de cada ponto respetivamente. Deste modo está feito o percurso entre os dois pontos e procede-se à escrita do ficheiro que contém a linha inicial do problema, custo total, número de pontos da solução e nas linhas seguintes estão os pontos que constituem o caminho, um em cada linha como se pode ver de seguida:

```
3 17 A 2 32 4
1 2 2
2 4 3
1 6 20
2 8 7
```

VARIANTE B

Neste caso para a Variante B vai ser executada a variante A *(n-1)* vezes em que *n* é o número de pontos. O tabuleiro deste problema é igual ao do problema anterior e por isso foi omitido.

```
3 17 B 3
2 0
1 0
1 3
1 11
```

A leitura do ficheiro é igual ao exemplo anterior bem como a verificação dos pontos. A diferença reside no número de vezes que o Dijkstra vai ser executado bem como os pontos inicial e final, como mostra a seguinte tabela.

Execução do algoritmo	1	2
Pontos para a execução	(2,0) -> (1,0)	(1,0) -> (1,3)

O custo acumulado bem como o número de pontos vai ser acumulado numa variável do tipo *caminho* para que no final seja mais fácil a impressão dos valores.

VARIANTE C

Como foi anteriormente explicado a variante C corresponde à execução de várias variantes B's e por ser a mais complexa exigiu a implementação de algumas estruturas adicionais.

```
200 200 C 4
0 3
199 199
66 66
133 133
```

Tomando este exemplo com apenas 4 pontos em que se aplica a correspondência da tabela do lado, existem $(4-1)!$ Combinações de caminhos possíveis.

(0,3)	I (= 0)
(199,199)	A (= 1)
(66,66)	B (= 2)
(133,133)	C (= 3)

Nesta variável o ponto inicial é fixo e por isso os pontos que variam no caminho são apenas os pontos B, C e D, originado as seguintes possíveis combinações:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Como estrutura auxiliar vamos definir uma matriz de dimensão $n \times n$, com n número de pontos, que na posição i, j contém o caminho do ponto de índice i para o ponto de índice j e uma outra matriz de dimensão $L \times C$, ou seja, da dimensão da matriz do problema que em cada posição tem o custo acumulado, bem como o seu pai.

Exemplificando para a primeira combinação, *1 2 3*, os caminhos calculados vão ser $(0) \rightarrow (1); (1) \rightarrow (2); (2) \rightarrow (3)$ e armazenados na matriz nas respetivas posições $[0; 1]; [1; 2], [2; 3]$ para que mais tarde possam ser usados caso seja necessário.

Para cada combinação vai ser calculado o custo total e comparado com o custo da melhor combinação. Caso seja melhor então obteve-se uma combinação que tem um custo inferior. No final, sabe-se qual é a combinação com um custo menor e é essa a que vai ser impressa no ficheiro.

ANÁLISE DOS REQUISITOS COMPUTACIONAIS:

Ao longo do projeto foram tomadas decisões sempre tendo por base as suas implicações e custo, tanto em termos de memória, como de tempo. Assim, para uma análise mais concreta e pormenorizadas vamos explicar tendo por base cada variante do programa. Para uma melhor perceção do problema, considera-se uma matriz original $L \times C = N$.

INICIALIZAÇÃO:

A inicialização das variáveis, quer de leitura, quer de execução de algoritmos, não dependem do tamanho do problema inicial, logo têm complexidade $\mathcal{O}(1)$.

DIJKSTRA

O nosso algoritmo de Dijkstra no pior caso é $\mathcal{O}(E \log V)$, onde E representa o número de arestas e V o número de vértices. Em termos de tempo, como cada ponto tem no máximo 8 pontos adjacentes, a complexidade no pior caso não excederá $\mathcal{O}(8 \log V)$.

Em cada uma das variantes a complexidade varia consoante o número de vezes que o algoritmo é executado, como exemplo, o Dijkstra na variante A é executado apenas 1 vez. Por outro lado, na variante B, dado um caminho com T pontos turísticos, o Dijkstra é executado T – 1 vezes. Por último, na variante C, no pior dos casos o Dijkstra é chamado $(T-1)! \times (T-1)$.

ACERVOS

As funções dos acervos são executadas de forma idêntica para cada uma das variantes. Assim, a inserção e remoção são ambas, $\mathcal{O}(\log N)$, mas a procura foi reduzida substancialmente com o uso de uma matriz de indexação, com tamanho idêntico à da matriz original, sendo por isso $\mathcal{O}(1)$.

EXECUÇÃO DO PROGRAMA

VARIANTE A

A variante. A vai ter, uma complexidade $\mathcal{O}(8 \log V)$, visto que é executado o Djikstra apenas uma vez. Em termos de memória, foi gasta memória na criação de uma matriz Dijkstra e do Acervo.

VARIANTE B

A variante B, consiste na execução da variante A T – 1 vezes, tendo uma, por isso uma complexidade de $\mathcal{O}((T-1)8 \log V)$. Em termos de memória, aloca e liberta (T -1) vezes a mesma memória da Variante A.

VARIANTE C

A variante C, a nível de armazenamento é mais dispendiosa, pois alocamos memória para uma matriz de caminhos com tamanho igual à matriz original, para guardar os caminhos já calculados. A criação de 2 vetores de apontadores para caminhos, permite reduzir a complexidade temporal, pois na construção de um caminho não é necessário copiar todos os pontos, pois estes estão armazenadas na matriz dos caminhos e este vetor contém apontadores para as diversas posições da matriz. Assim, no pior caso a complexidade é $\mathcal{O}((T-1)!\,(T-1)8 \log V)$.

ANÁLISE CRÍTICA:

Nas primeiras submissões tivemos uns ligeiros percalços, relacionados com erros de compilação, mas depois de corrigidos, bastaram-nos relativamente poucas submissões para alcançar o número de testes máximos na parte A e B. Contudo, realizada a parte C, conseguimos o 18, a partir daqui tentámos incessantemente descobrir porque é que estávamos a falhar em 2 testes, foi a partir daqui que implementámos o Branch and Bound, alterámos o Dijkstra, alterámos a recursividade para ser mais rápida e implementámos a uma matriz de caminhos que armazena os caminhos já calculados, bem como o cálculo imediato dos seus simétricos, mas tudo sem sucesso. Depois de uma análise do gprof, analisámos que o tempo estava a ser consumido no FixDown e nas funções relacionadas com as estruturas ocultas. Assim, apercebemo-nos que tínhamos de reduzir o número de vezes que usávamos o Dijkstra, algo que depois fizemos passando a usar o Dijkstra para calcular não só o caminho AB da combinação ABCD, mas calcular A a C, A a D se estes tiverem inseridos no percurso de A a B, mas, com a falta de tempo, decidimos não submeter com esta alteração.

Em conclusão, pensamos ter um projeto bem conseguido, que nos permitiu desenvolver a nossa visão de perspetivar problemas de otimização, bem como aplicar os conhecimentos obtidos nas aulas teóricas e práticas de Algoritmos e Estruturas de Dados.

BIBLIOGRAFIA

- Acetato 05 – "Análise": algoritmo de procura binária
- Acetato 10 – "GrafosC": Dijkstra
- Acetato 11 – "Heaps": Acervo
- Acetato 12 – "Tree": Recursividade
- Laboratório 6 – "heap.c"