Tourist-Knights

Algoritmos e Estruturas de Dados

João Almeida - 90117
https://github.com/Umelhor99

Miguel Fazenda - 90146
https://github.com/miguel71

Docentes:
Carlos Filipe Gomes Bispo
António Manuel Morgado Brandão Leal
Luís Miguel Teixeira D'Avila Pinto da Silveira
Nuno Filipe Neves

Grupo 11
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Instituto Superior Técnico
December 12, 2018

Contents

1 Apresentação do Enunciado

Fénix: *https://bit.ly/2Pp0L3b*

GitHub: *https://bit.ly/2G7HZyh*

1.1 Descrição do Problema

Este trabalho , descrito sucintamente, visa a análise de puzzles estáticos (2D) tendo em consideração o custo de cada célula e célula inicial, que dado um ficheiro com vários puzzles, se consiga calcular o melhor caminho.

Cada ficheiro de entrada pode conter ou não vários puzzles estáticos constituídos, idealmente, por um cabeçalho, uma lista de coordenadas de pontos turísticos de células e por uma cidade. O cabeçalho dispõe de 4 informações:

- Dimensões da matriz (cidades), (altura e largura);
- Tipo de variante, ou objetivo do agente;
- Número de células necessárias que o agente tem de visitar.

A lista de coordenadas de pontos turísticos dispõe das coordenadas do género (x,y), próprias de uma célula necessária ao caminho do agente. O tamanho da lista de células diz respeito à última informação no cabeçalho. A primeira linha da lista de coordenadas de pontos turísticos diz respeito às coordenadas iniciais do problema.

A cidade pode ser representada por uma matriz de inteiros, contendo duas informações distintas:

- Inteiro nulo - Representa que a célula é inacessível;
- Inteiro não nulo - Custo de entrada / passagem na célula.

O melhor caminho calcula-se tendo em conta um ponto inicial que é sempre conhecido, bem como o custo de entrada em cada célula, assim, este refere-se ao caminho menos dispendioso em termos de custo total (soma do custo de entrada de todas as células pertencentes ao caminho), ao invés de ser o caminho de menor distância. Cada célula além do seu custo também tem coordenadas próprias.

O problema apresenta três variantes, que embora o objetivo seja comum, dá a possibilidade ao agente de escolher completar objetivos diferentes:

(A) Cálculo do melhor caminho, entre duas células da matriz dadas no cabeçalho;

(B) Cálculo do melhor caminho, entre várias células pela ordem definida no ficheiro;

(C) Cálculo do melhor caminho, tendo como começo a célula inicial, passando pelas outras células presentes na lista de coordenadas de pontos turísticos pela melhor ordem.

1.2 Abordagem ao Problema

Em primeiro lugar , começou-se por guardar os dados relativos a cada puzzle estático numa estrutura, permitindo assim organizar a informação relativa a cada puzzle mais facilmente e ser de rápido acesso.

De seguida, após analisar algumas hipóteses de como lidar com o ficheiro de entrada, definiu-se que a possível solução mais equilibrada, balanceando memória e custo, seria escrever no ficheiro de saída à medida que analisavamos cada um dos puzzles. Esta solução foi a escolhida tendo em conta que a hipótese paralela, armazenar toda a informação e respetiva análise do ficheiro de entrada e posterior escrita no ficheiro de saída, ser desequilibrada no que diz respeito à quantidade de memória envolvida.

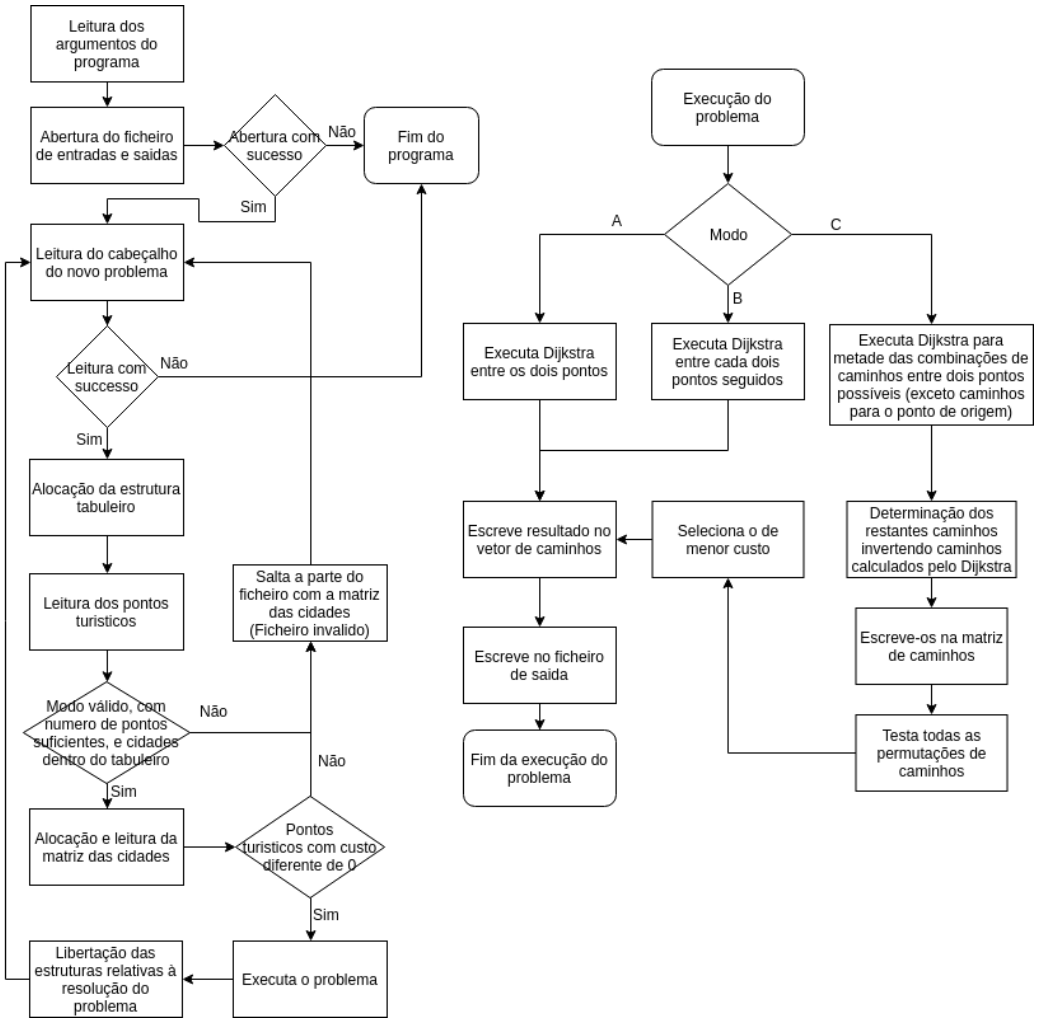
A solução escolhida, foi implementada de tal forma que trata cada puzzle como um isolado, visto que quando um novo puzzle é lido são criadas novas estruturas (alocação de memória). Por conseguinte após um puzzle ser solucionado, liberta-se a memória alocada.

Em terceiro lugar, no momento da decisão do algoritmo a usar para o cálculo, preferiu-se o algoritmo *Dijkstra*.Apesar de este algoritmo pecar da sua complexidade, dá certezas que se existe caminho válido entre duas células, obtido seria sempre o de menor custo.

Finalmente, visando a necessidade de implementar uma fila prioritária de suporte ao algoritmo *Dijkstra*, preferiu-se a implentar de um *Heap* ou acervo (estrutura de dados organizada como árvore de dados binária balanceada, ao invés de uma tabela não ordenada ou ainda de uma tabela ordenada).

2 Arquitetura do Programa

Fluxograma programa tuktuk:



3 Descrição das Estruturas de Dados

Durante a realização do projeto decidiu-se guardar os dados relativos à leitura de cada puzzle em estruturas , bem como em relação aos métodos usados para a resolução do problema. Assim trabalha-se com uma melhor forma de armazenamento de informação, não só do aspeto da organização, mas também da rapidez de acesso à mesma.

3.1 struct tabuleiro_t

A struct tabuleiro_t é a estrutura central do código, visto que é nesta estrutura que é guardada toda a informação relativa ao cabeçalho, à lista de coordenadas de pontos turísticos e à cidade.

Nesta estrutura também são guardadas duas matrizes (wt & st) relacionadas com a realização do algoritmo *Dijkstra*.

É também na estrutura tabuleiro_t que se guarda um apontador para acervo_t. Foi também relevante ter guardado o número de caminhos que se encontram no vetor de caminhos do tipo path_t, bem como o respetivo vetor de caminhos.

3.2 struct acervo_t

A struct acervo_t é uma estrutura auxiliar à função *Dijkstra*, onde se encontra guardados parâmetros como: o heap ou acervo, o respetivo tamanho (memória alocada) e o número de elementos presentes no heap. Também se encontra nesta estrutura uma matriz de índices, que nos permite um acesso mais rápido ao heap.

O *Heap* (ou acervo em português) é uma forma bastante eficiente de implementar uma fila de prioridades. É uma árvore binária balançada, (em que ambos os filhos de um elemento tem prioridade menor que a do pai. Foi usado um vetor dinâmico (com possibilidade de aumentar de tamanho com recurso à função *Realloc*) para representar a árvore.

3.3 struct path_t

A struct path_t é uma estrutura que representa um caminho. Esta contêm tanto o custo do caminho bem como o comprimento do caminho,

e possuí também dois Pares de coordenadas dos pontos inicial e final. Por fim nesta função também se encontra o vetor de pontos ou caminho.

Visto ser uma estrutura simples e de grande uso durante todo o código, deixou-se definida no ficheiro path.h .

3.4 struct vector2_t

A struct vector2_t é uma estrutura básica que permite representar um par de coordenadas (x,y). Foi adotada uma ordem diferente da ordem no enunciado, visto que o mesmo usa a notação (altura, largura) e no trabalho usou-se (largura, altura) devido ao paralelismo (x,y).

Visto ser uma estrutura de grande uso durante a todo o código e ser relativamente simples, deixou-se definida no ficheiro vector2.h .

4 Algoritmos

4.1 *Dijkstra*

O algoritmo *Dijkstra* permite determinar o melhor caminho de uma célula a vários destinos num grafo, tendo em conta o peso de cada aresta. Neste projeto, a matriz de custos das cidades constituiu um grafo implícito, existindo uma ligação entre cada célula e as células acessíveis a partir de um salto de cavalo (no máximo 8 movimentos possíveis por célula).

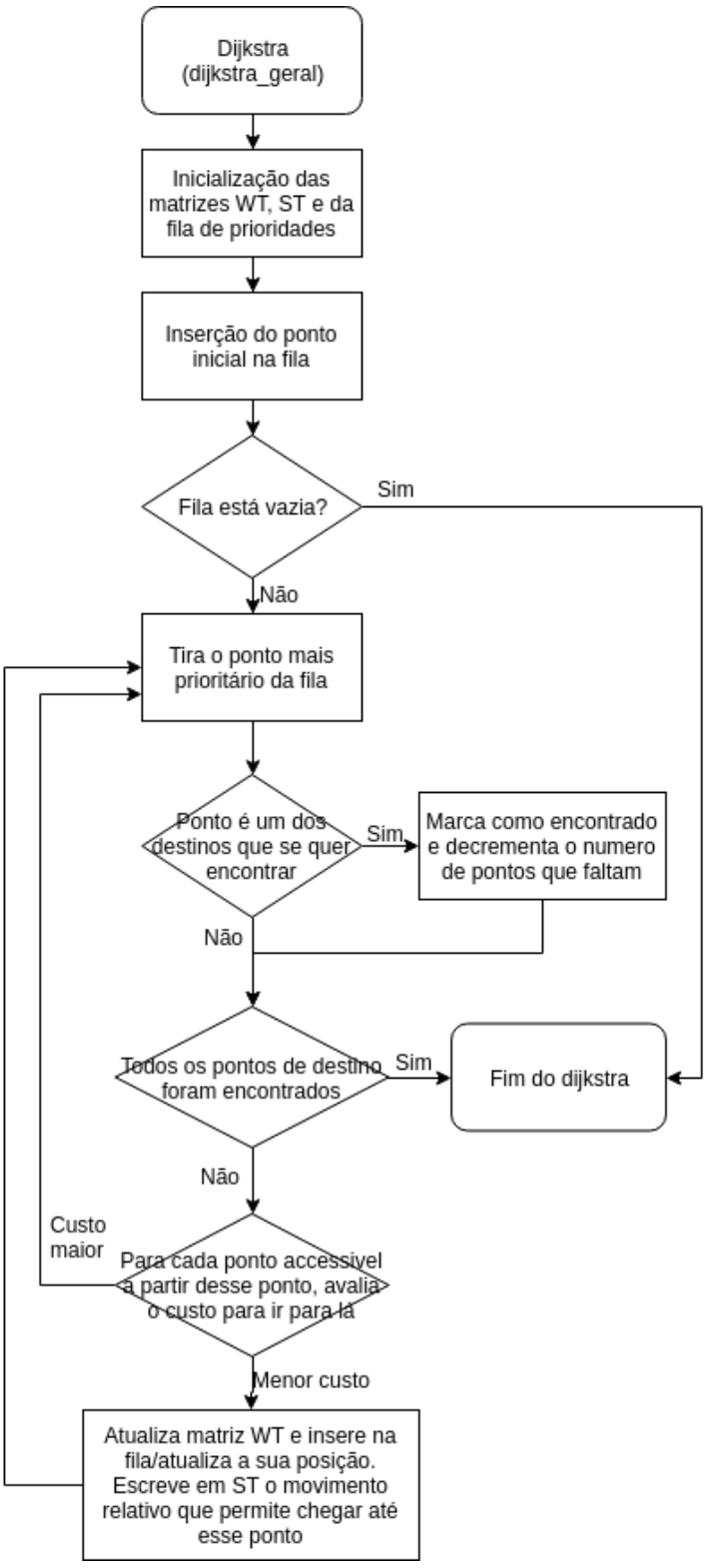
Todos as células exceto a célula inicial são marcados como custo até ao ponto inicial infinito (-1), porque ainda não foram visitadas.

As condições de paragem do algoritmo, são : Quando o heap se encontra vazio ou quando o célula final é removida do *Heap*.

Quando o elemento mais prioritário é retirado do heap, são avaliados os movimentos possíveis que o agente pode realizar.

Compara-se o custo acumulado da nova célula, com a soma do custo da nova célula e do custo acumulado da célula onde o agente se encontra, caso este seja menor, atualiza-se a posição da nova célula no *Heap* tendo em conta a atualização do novo custo acumulado atualizado na matriz wt. Se o movimento for válido guarda-se na matriz st o número do movimento (1-8) que deu origem ao filho a partir do seu pai.

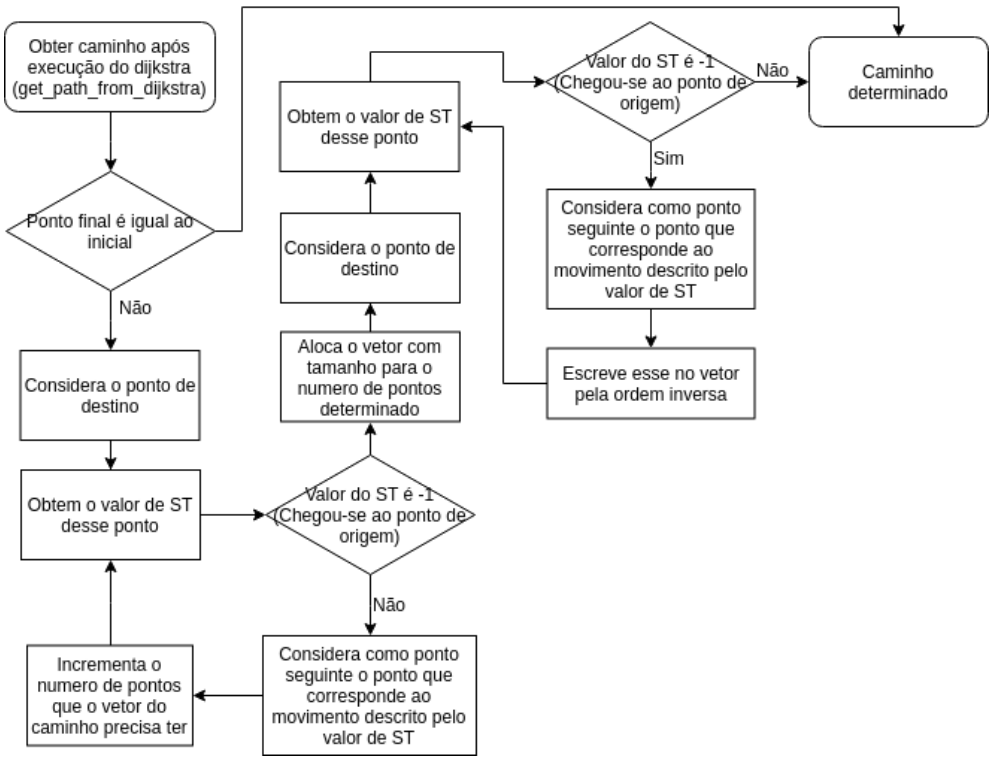
Na página a seguir, está um fluxograma mais explícito do funcionamento do algoritmo.


Dijkstra (dijkstra_geral)
Inicialização das matrizes WT, ST e da fila de prioridades
Inserção do ponto inicial na fila
Fila está vazia?
Sim
Não
Tira o ponto mais prioritário da fila
Ponto é um dos destinos que se quer encontrar
Sim
Marca como encontrado e decrementa o numero de pontos que faltam
Não
Todos os pontos de destino foram encontrados
Sim
Fim do dijkstra
Não
Custo maior
Para cada ponto accessivel a partir desse ponto, avalia o custo para ir para lá
Menor custo
Atualiza matriz WT e insere na fila/atualiza a sua posição. Escreve em ST o movimento relativo que permite chegar até esse ponto

4.2 *movimentos_find_path_to_city*

Para encontrar um caminho entre dois pontos começa por se verificar se esse mesmo caminho já foi calculado anteriormente (se está presente no vetor de caminhos a serem escritos no ficheiro de saida $tabuleiro- > paths$). Verifica-se também se o ponto está à distancia de um salto de cavalo ou se o ponto de origem é igual ao de destino. Em caso afirmativo, facilmente se determina o caminho, e não é necessário executar o *Dijkstra*. Sendo necessário calcular o caminho começa-se por executar o *Dijkstra*. De seguida executa-se a função get_path_from_dijkstra. Esta cria o caminho a partir da matriz st já determinada. O seu funcionamento está descrito no fluxograma.

Existe também a função movimentos_find_path_to_multiple_cities que funciona de forma semelhante, mas que calcula o caminho para vários destinos em vez de um apenas.



4.3 *calcula_matriz_custo_caminhos*

Esta função calcula, para a variante C, uma matriz de adjacências que representa os custos de ir de um ponto turístico a outro. A matriz é uma matriz de caminhos, por isso para além do custo de ir de um ponto ao outro guarda também os pontos de passagem. Tem como largura o numero de pontos turísticos e altura o numero de pontos turísticos - 1. A coluna representa o ponto de origem e a linha o ponto de destino, e como o ponto inicial é fixo, não pode ser um destino, e por isso há menos uma linha do que colunas. Assim, o caminho entre dois pontos obtém-se através de [i][j-1], sendo i o índice no vetor dos pontos turísticos da célula de origem e j o da cidade de destino.

4.4 *calcula_melhor_caminho*

Esta função permite obter a ordem de passagem pelos pontos turísticos que minimiza o custo. A partir de um vetor com os índices dos pontos no vetor de pontos turísticos a função é chamada com o primeiro elemento fixo, já que o primeiro ponto é fixo. A função é recursiva, e vai fixando elementos que ainda estavam livres no vetor. Com todos os elementos fixos, compara-se o custo com o mínimo até determinado até ao momento, e se for melhor copia-se o vetor para um outro que guardará a melhor ordem. Para otimizar, o custo vai sendo calculado progressivamente e assim que esse ultrapassa ou iguala o mínimo anteriormente determinado, não continua e prossegue não fixando mais elementos nas posições seguintes.

5 Subsistemas funcionais

Neste programa foram usados 6 subsistemas: acervo, tabuleiro, movimentos, util, path e vector2.

Os ficheiros tabuleiros.h e acervo.h incluem as declarações das funções usadas para o manuseamento da struct tabuleiro_t e acervo_t, respetivamente, e seus typedefs. Os ficheiros util.h e movimentos.h, incluem apenas as declarações de funções que dizem respeito aos respetivos *.c . Nos ficheiros path.h e vector2.h, além das definições das funções usadas nos respetivos *.c, também é possível encontrar a declaração das funções e tipo de estruturas auxiliares criadas.

Os ficheiros tabuleiros.c e acervo.c dizem respeito a todas as funções que envolvem o manuseamento de ambas as respetivas estruturas, tais como alocação de memória, inicializações, manuseamento, execução do código dependente de cada variante. Também nestes ficheiros temos funções auxiliares de busca de informação de um dado elemento da matriz *acervo_get_top* e de inserção de novos valores *tabuleiro_set_element*(ex. tabuleiro_set_height, tabuleiro_set_cost, tabuleiro_get_fila, ...), pois está estrutura não está visível para os outros subsistemas do código, visto que está declarada no respetivo ficheiro.c.

O ficheiro util.c inclui funções auxiliares como por exemplo a alocação de memória com a respetiva verificação de uma correta alocação. O ficheiro path.c incluiu uma função de criação de um *path* e respetiva inicialização, e de criação de um *path* com um ponto. O subsistema vector2.c engloba todas as operações que envolvem a soma, a subtração, a leitura, comparação e a criação de um novo ponto. Por fim o ficheiro movimentos.c engloba todo o código relacionado com o movimento do agente no tabuleiro e cálculo do melhor caminho da cidade.

Em alguns ficheiros *.c incluímos a declaração na função com o modo 'static' visto que são funções apenas usadas nos respetivos *.c .

6 Análise dos requisitos computacionais

Considerando um puzzle de tamanho $C \times L$, com T pontos turísticos, e designando $C \times L = N$:

- Inicialização: A complexidade da inicialização do programa não depende do tamanho do problema, logo é $\mathcal{O}(1)$
- Leitura de um puzzle: O puzzle para além da matriz dos custos contem o cabeçalho e os pontos de passagem por isso a sua complexidade será $\mathcal{O}(\mathrm{N} + \mathrm{T} + 1)$, que para puzzles grandes pode ser aproximado por $\mathcal{O}(\mathrm{N})$
- Dijkstra: A complexidade do algoritmo *Dijkstra* (função dijkstra_geral) implementado com um acervo é dada por $\mathcal{O}(E \lg V)$, sendo E o numero de arestas e V o numero de vértices. Assim para este problema, E vai ser aproximadamente 8 (porque cada célula tem no máximo 8 vizinhos), e V vai ser N, logo a complexidade será $\mathcal{O}(8 \lg N)$.
- Procura de caminhos: A complexidade da procura de caminhos (funções: movimentos_find_path_to_city e movimentos_find_path_to_multiple_cities) a partir de um ponto de origem, será a complexidade da função dijkstra_geral, e da função get_path_from_dijkstra, que será executada tantas vezes quanto o numero de destinos. A complexidade desta ultima função será $\mathcal{O}(2K)$, sendo K o numero de células que formam o caminho. Assim a complexidade da procura do caminho será $\mathcal{O}(8 \lg N \times +(2K \times X))$, sendo X o numero de destinos.
- Execução de um puzzle estático: A complexidade da execução depende do tipo de puzzle:
  - Modo A: Executa-se uma procura com um destino, logo a complexidade será $\mathcal{O}(8 \lg N \times 2K)$;
  - Modo B: Executa-se T-1 procuras com um destino(no pior caso, em que não é percorrido o mesmo caminho várias vezes), logo a complexidade será $\mathcal{O}((T-1) \times (8 \lg N \times 2K))$;
  - Modo C: No modo C executa-se uma procura por cada ponto turístico, para preencher a matriz de adjacências (exceto para o

ultimo), em que cada uma será com menos pontos de destino. Isto porque basta preencher a matriz triangular inferior, sendo que a a parte superior é preenchida invertendo os caminhos já calculados. O primeiro ponto terá T-1 destinos, o segundo T-2, sendo que o ultimo terá 0 destinos, daí não se executar a procura para este. Sendo K o numero de pontos médio dentro de cada caminho, o custo de preencher a parte inferior da matriz será $\mathcal{O}((T-1) \times 8 \lg N + 2((T-1)K + (T-2)K + \ldots + 0))$, que se pode simplificar para $\mathcal{O}((T-1) \times 8 \lg N + 2K \times \frac{T(T-1)}{2})$. A inversão de um caminho tem complexidade $\mathcal{O}(K)$, logo a inversão das $\frac{T^2}{2} - T$ entradas da matriz preenchidas tem complexidade $\mathcal{O}((\frac{T^2}{2} - T) \times K)$. Por fim acrescenta-se a a execução, no pior caso de todas as permutações ($\mathcal{O}((T-1)!)$) que nos permitem o cálculo de melhor custo.

- Escrita do ficheiro: A complexidade da escrita do ficheiro será $\mathcal{O}(1 + T \times K)$ (com K o numero médio de pontos em cada caminho)
- Fim do programa: O(1)

Assim a complexidade total para cada modo será:

- Modo A: $\mathcal{O}((N+T) + (8 \lg N \times 2K) + (T \times K))$
- Modo B: $\mathcal{O}((N+2T) + ((T-1) \times (8 \lg N \times 2K)) + (T \times K))$
- Modo C: $\mathcal{O}((N+2T) + ((T-1) \times 8 \lg N + 2K \times \frac{T(T-1)}{2} + (\frac{T^2}{2} - T) \times K) + (T-1)! + (T \times K))$.

A complexidade de memória no do programa será:

- Para a matriz de custos dos pontos: $\mathcal{O}(N)$
- Para o vetor de pontos turísticos: $\mathcal{O}(T)$
- Para a matriz wt e st: $\mathcal{O}(N)$
- Para o acervo: O limite será $\mathcal{O}(N \times \frac{6}{8})$, porque entram 7 pontos e sai 1.
- Para o vetor de caminhos que serão escritos no ficheiro: $\mathcal{O}((T-1) \times K)$

7 Exemplo Prático

Nesta secção vai ser ilustrado um breve exemplo de como o programa funciona.

7.1 Variante A

Nota importante: neste exemplo, a notação de par de coordenadas adotada, foi em concordância com o enunciado para não causar qualquer confusão.

Considere-se o seguinte puzzle estático na variante A, lido a partir de um ficheiro ⟨nome_do_ficheiro⟩.cities:

```
7 7 A 2
0 0
5 4
1 2 1 3 1 1 8
4 1 3 5 6 1 1
2 2 3 7 0 5 6
9 1 5 7 2 0 3
1 1 7 5 1 0 1
1 3 8 1 7 9 1
4 1 9 8 1 1 0
```

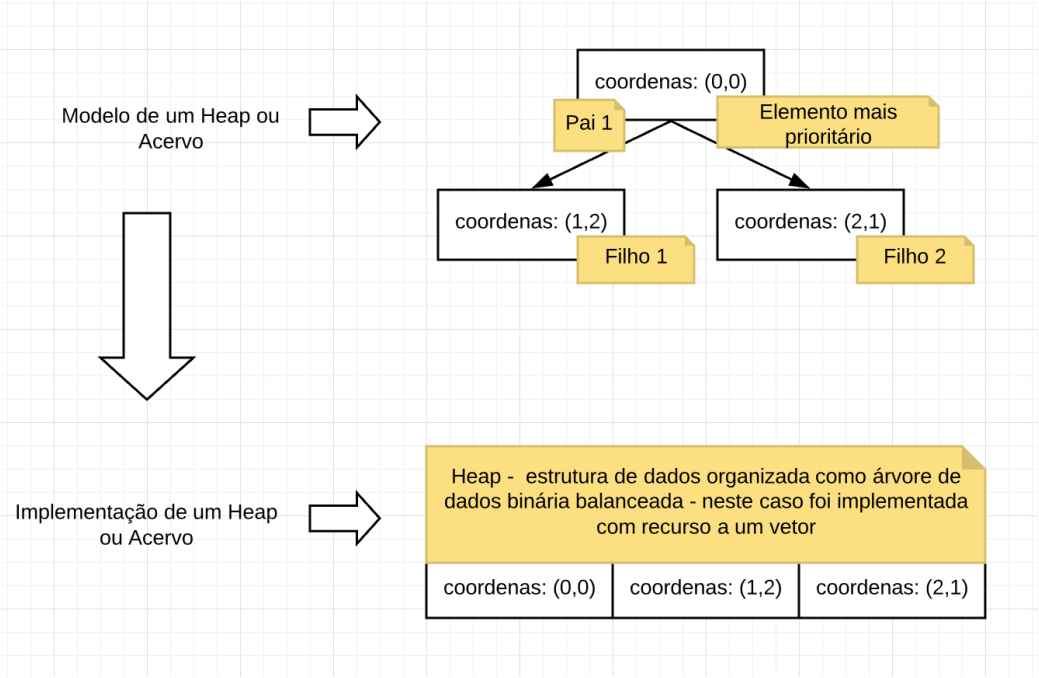
Para uma análise mais simples, rápida e eficaz considere-se três caminhos diferentes (possíveis soluções contidas no ficheiro ⟨nome_do_ficheiro⟩.walks, os quais tem potencial para ser o melhor caminho, sendo um deles o melhor caminho (x, y, custo):

caminho 1	caminho 2	caminho 3
1 2 1	2 1 2	2 1 2
0 4 1	4 2 7	4 0 1
1 6 1	5 4 7	6 1 1
-1		5 3 1
		6 5 1
		4 6 1
		5 4 7

Analisando o as informações que o cabeçalho, a lista de coordenadas de pontos turísticos e a cidade nos transmitem, conseguimos logo à partida validar a cidade visto que tanto a célula inicial como o célula final têm coordenadas válidas e são células acessíveis.

Visto se ter reunido as condições necessárias à execução da resolução do problema, vamos analisar se ambas as células estão à distância de um salto de cavalo. Como a condição é falsa, dá-se inicio ao algoritmo *Dijkstra*.

Após a iteração para a primeira célula o *Heap* vai ter na primeira posição a célula de coordenadas (0,0) de custo acumulado 0 (ou wt, sendo wt a matriz do custo acumulado) 0, na segunda posição a célula de coordenadas (1,2) com um custo acumulado 1, e na terceira posição a célula de coordenadas (2,1) com wt=2 .

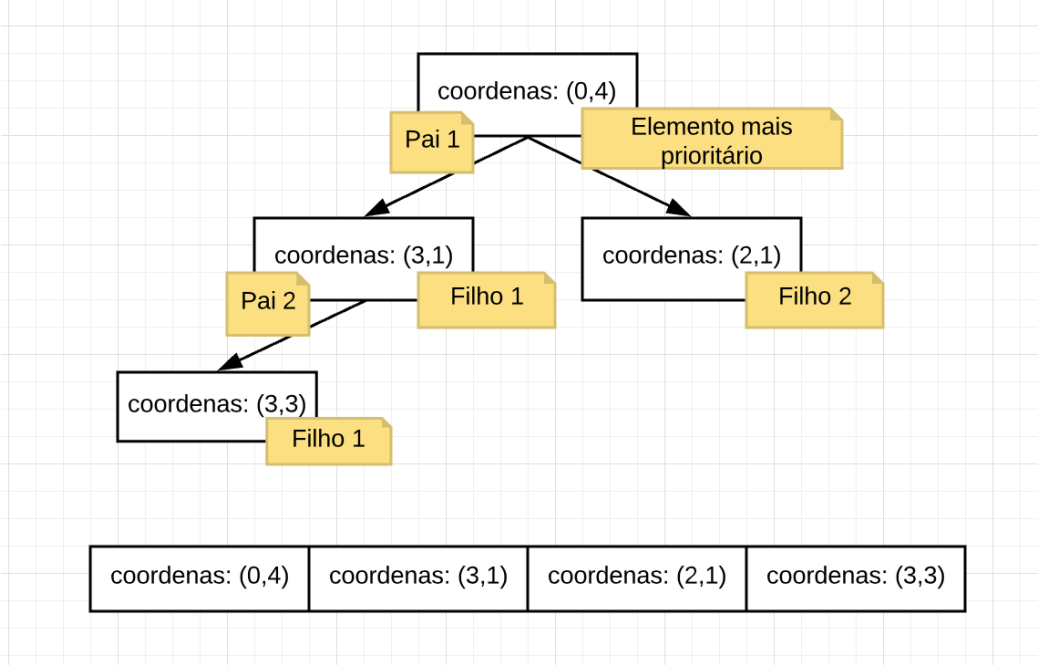


Desta maneira, a próxima execução do programa resulta com a remoção do pai e nomeação de um novo elemento mais prioritário. Visto que

$$wt[filho1] < wt[filho2]$$

, seja o filho 1 o novo elemento mais prioritário. Continuando, analisa-se os pontos adjacentes de (1,2) que são (0,4) com wt=1, (2,4) com wt=0 (célula inválida não vai para o acervo), (3,3) com wt = 7 e (3,1) com wt = 1. Assim após a inserção das novas células no acervo e a remoção do elemento mais

prioritário, o acervo e o vetor têm o seguinte aspeto:



Prosseguindo, o algoritmo irá desenhar o caminho 1 até à célula (1,6) visto ser a última célula possível para passar , pois das suas três células adjacentes, uma é o seu pai, e as outras duas são células inacessíveis.

Voltando assim atrás no algoritmo, temos a célula de coordenadas (2,1) como elemento mais prioritário. Se formos analisar bem a cidade, e o caminho 2, reparamos que a distância em linha reta entre as células inicial e final é menor seguindo o referido caminho, porém ao aplicar o algoritmo *Dijkstra* com auxílio de um *Heap*, a célula de coordenadas (4,2) ficará em *stand-by* no acervo devido ao seu wt=7, obtendo entretanto o melhor caminho, neste exemplo, o caminho 3.

Neste puzzle, o caminho 3 não é direto, no entanto o maior número de células que o agente visita em comparação com o caminho 2, é compensada por um menor custo do referido caminho numa nova comparação com o caminho 2.

Assim o cabeçalho do ficheiro ⟨nome_do_ficheiro⟩.walks produzido será:

7 7 A 2 14 7

(seguido do caminho 3)

7.2 Variante B

Considere-se agora a cidade do exemplo anterior, mas o seguinte cabeçalho e as duas listas de coordenadas de pontos turísticos:

```
7 7 B 4
0 0
2 1
3 3
5 4
```

Nesta variante vai ser calculado o melhor caminho entre as células inicial e final, visitando obrigatoriamente determinadas células, pela ordem em que aparecem no ficheiro ⟨nome_do_ficheiro⟩.cities. Assim estamos em condições de repetir a variante A, (n-1) vezes, sendo n o número de células presentes na listas de coordenadas.

Exemplificando, para o enunciado dado vamos aplicar o algoritmo *Dijkstra* n-1 = 4-1 = 3 vezes, para calcular o melhor custo entre a célula (0,0) e a célula (2,1), entre entre as células (2,1) e (3,3) e entre entre as células (3,3) e (5,4). De seguida somam-se os três custos calculados, e obtém-se o melhor custo do caminho total.

Assim o cabeçalho do ficheiro ⟨nome_do_ficheiro⟩.walks produzido será:

```
7 7 B 2 16 3
2 1 2
3 3 7
5 4 7
```

7.3 Variante C

Por fim considera-s,e na variante C, a cidade apresentada no exemplo da Variante A, e o seguinte cabeçalho e as duas listas de coordenadas de pontos turísticos: 7 7 C 4

caminho 1	caminho 2
2 1	1 2
5 4	5 4
3 3	3 3

Para esta variante calculou-se o custo entre cada caminho parcial, (de coluna para linha), conforme exemplificado nas tabelas 1 e 2 correspondentes ao caminho 1 e 2, respetivamente:

	(0,0)	(2,1)	(5,4)	(3,3)	(0,0)	(1,2)	(5,4)	(3,3)	
(2,1)	2	-	9	1	3	-	10	3	(1,2)
(5,4)	14	14	-	7	14	14	-	7	(5,4)
(3,3)	8	7	7	-	8	7	7	-	(3,3)

Nota importante 1: Nesta tabela, a coluna 1 e 10, e a linha 1 são apenas ilustrativas!

***Nota importante 2:** No código a matriz é de *paths*, sendo aqui representada apenas o custo do caminho, visto que a representação do caminho em cada célula iria tornar o exemplo mais complicado de se perceber.

Para o cálculo do melhor custo neste exemplo, foi calculado o melhor custo em cada caminho parcial e estudou-se de seguida todas as combinações (relativas à ordem de visita às células).

Tendo em conta a tabela* anterior, os caminhos 1 e 2 e a cidade, se se tratasse da variante B no caminho 1 obtinha-se que melhor custo seria 23 ((0,0) até (3,3)), no entanto na variante C, ao permutarmos os caminhos chegamos à conclusão que começando na célula (0,0) realizando o caminho 1 , o melhor custo será 2+7+7 = 16. Este custo diz respeito à seguinte ordem de células: ((0,0), (2,1), (3,3), (5,4)).

O caminho 2 não vai ser analisado, devido à sua semelhança com o caminho 1.

Caminho 2 ordenando: ((0,0), (1,2), (3,3), (5,4)) & custo total = 3+7+7 = 17. O melhor caminho nesta variante é o caminho 1.

Assim o cabeçalho do ficheiro ⟨nome_do_ficheiro⟩.walks produzido será:

```
7 7 C 2 16 3
2 1 2
3 3 7
5 4 7
```

8 Análise crítica

As primeiras submissões do projeto não foram concluídas com sucesso visto que o site estava a detetar erros básicos de compilação, de output produzido ou de saídas de função com valor diferente de 0, (ex: "exit(1)").

Posto isto o código ficou com marca equivalente entre a nota 16 e a 18, visto que se esteve em processo de melhoramento da eficácia e rapidez de código. Quando finalmente, o objetivo foi atingido, a nota máxima, obtou-se por fazer algumas alterações ao código e comentar mais explicitamente algumas partes do código. Por impulso, submeteu-se código ainda não concluído que baixou a nota, e piorou o resultado em termos de tempo. Após sermos avisados pelo professor que estava-mos a prejudicar os nossos colegas, limitam-mo-nos a submeter a última vez com os erros corrigidos de maneira a ter a nota máxima outra vez. Com isto, poderá haver partes do código menos explícitas.

O grupo tem noção que o código ainda poderia ser melhorado, como por exemplo pôr todas as estruturas ocultas ou usar abstração em relação ao *Heap*.

Comentando agora as partes que correram melhor no projeto, dizem respeito à pouca memória usada pelo programa, à divisão de assuntos e funções por subsistema, alguns raciocínios que se encontram com um excelente nível ou ainda à libertação total da memória alocada.

Em suma, considera-se que o projeto foi bem desenvolvido e que as decisões tomadas e concretizadas ao longo do projeto, quer seja na escolha das estruturas de dados, algoritmos e estratégias foram tomadas de modo a tornar o projeto o mais eficiente possível, com gestão de memória e reduzida complexidade.

9 Bibliografia

- Acetato 04 - "EstDados": Elementos genéricos;
- Acetato 05 - "Analise": Estudo da Complexidade;
- Acetato 09 - "GrafosC": Algoritmo Dijkstra;
- Acetato 11 - "Heaps": Acervo;
- Laboratório #2 - Manipulação de tabelas e listas;
- Laboratório #4 - Ordenação;
- Laboratório #5 - Grafos;
- Laboratório #6 - Acervos;
- Generic C Makefile - *https://gist.github.com/ork/11248510*: