Amazeme - PROJETO AED

Pedro Chanca, Rudolfo Félix

31 de Outubro de 2019

Conteúdo

1 Descrição do Problema

Este projeto, criado no âmbito da disciplina de Algoritmos e Estruturas de Dados, tem dois objetivos. O primeiro, é obter um caminho com 'k' número de passos de modo a que atinja a energia final pedida inicialmente. Enquanto, o segundo objetivo, é encontrar o caminho com 'k' número de passos no qual seja alcançada a maior energia final possível.

O programa criado (em linguagem C) deve conseguir ler os problemas de um ficheiro, descrito numa linha de sete inteiros com os parâmetros necessários à resolução do problema e numa matriz de inteiros que a segue.

Os primeiros sete algarismos descrevem o tamanho do tabuleiro(matriz) de cada problema, linhas(L) e colunas(C), o objetivo do jogo (variante 1(-2) ou variante 2(inteiro positivo)), o ponto de partida (l1 e c1), o número de 'k' passos, a energia inicial(ei) com o qual o utilizador. Após estes parâmetros, é lido o próprio tabuleiro de jogo.

O programa, deve criar um ficheiro de saída, no qual irá estar representado as resoluções do objetivo escolhido com o os sete primeiros algarismos, e com um oitavo inteiro, apresentado se o problema tiver os parâmetros válidos, que indica que há solução(energia final) ou se é impossível resolver(-1). Se o problema apresentar solução e se o número de passos for maior ou igual a um, é apresentado 'k' linhas de três inteiros, onde em cada qual, é um passo descrito pelas coordenadas(linha e coluna), mais o seu custo.

2 Abordagem

Para resolver o problema proposto há que criar inicialmente uma estrutura que permita guardar as informações presentes na primeira linha do ficheiro '.maps', a qual contem as informações essenciais para a realização do jogo. Uma vez que por vezes nao será necessário utilizar todas as casas da matriz, devido ao valor de 'k' ser inferior ao número de passos necessários para chegar a essas posições, como tal seleciona-se o valor máximo de linhas e colunas necessárias para a realização do jogo (matriz útil). No caso do ficheiro '.maps' apresentar mais do que um problema, a matriz útil que irá ser utilizada na resolução de todos os problemas do ficheiro corresponde à maior matriz útil dos problemas presentes no ficheiro.

Seguidamente, ocorre todo o processo evolvente na resolução dos problemas existentes de variante 1 ou 2. Este processo passa por 3 fazes distintas: verificação da validade dos problemas, cópia da matriz do ficheiro de entrada para a estrutura correspondente e finalmente a sua resolução.

Finalmente, há medida que os problemas são resolvidos a sua solução final é escrita no ficheiro de saída '.path'.

3 Arquitetura do Projeto

Na Figura 1 mostra-se a estrutura geral e básica do programa com as operações centrais para resolução do problema, que se divide nas seguintes fases:

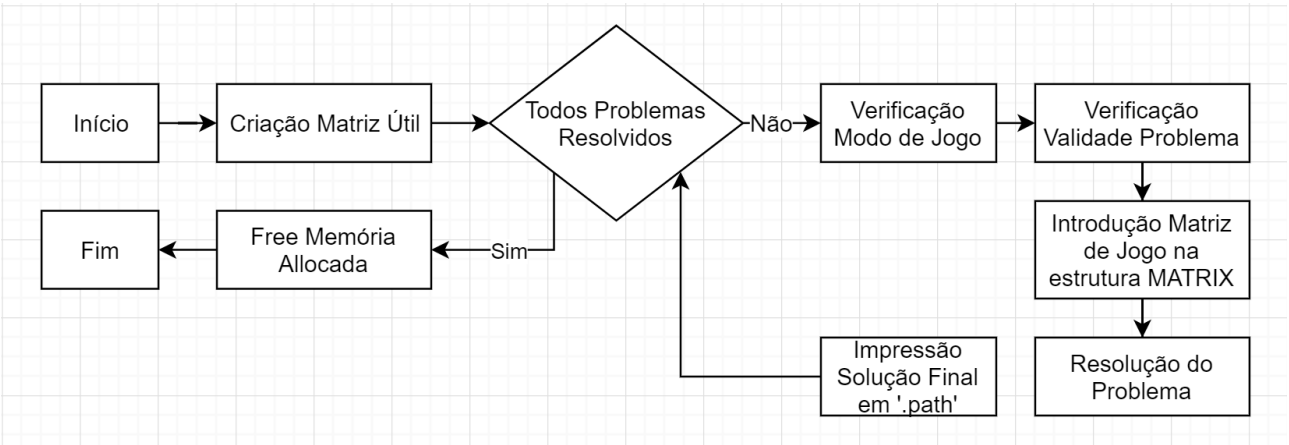

Figura 1: Main.c

Cada uma destas fases será analisada em maior detalhe na secção 5.

4 Estruturas e Tipos de Dados

Todas as estruturas têm a sua respetiva função de alocação e libertação de memória e os seus dados são manipulados por funções de abstrações de dados. As funções de libertação e abstração de memória não serão abordadas devido à sua pouco importância para a descrição do projeto.

4.1 Estruturas

4.1.1 MAZE

Descrita no ficheiro **structs.c** e definida no **structs.h**, a estrutura tipo MAZE tem como função principal guardar os dados fundamentais para a construção e resolução do problema. Contém, assim:

- MATRIX **matrix: Duplo ponteiro para a estrutura tipo MATRIX;
- int L: Inteiro que contém o número de linhas da matriz;
- int C: Inteiro que contém o número de colunas da matriz;
- int l1: Inteiro que contém o número da linha de partida da matriz;
- int c1: Inteiro que contém o número da coluna de partida da matriz;
- int m: Inteiro que contém o modo de jogo;
- int k: Inteiro que contém o número de passos a realizar;
- int k_r: Inteiro que contém o número de passos restantes, após cada passo na matriz;
- int ei: Inteiro que contém a energia inicial;
- int ef: Inteiro que contém a energia final;
- int eb: Inteiro que contém a melhor energia para a variante 2;
- int solved: Inteiro que funciona como flag (0 ou 1) para determinar se ocorreu a resolução do problema;
- int new-L: Inteiro que contém o número de linhas da matriz útil;
- int new-C: Inteiro que contém o número de colunas da matriz útil;
- int new-l1: Inteiro que contém o número da linha de partida da matriz útil;
- int new-c1: Inteiro que contém o número da coluna de partida da matriz útil.

Funções utilizadas para a criação e introdução de dados na estrutura:

1. **MAZE *createMaze(int max_line, int max_cols)**

   - Função do tipo MAZE* que executa a alocação de memória para a estrutura e inicialização das variáveis. Dentro desta função, também é inicializada a matrix através da sua igualdade à função 'createMatrix'. No final, ocorre o return do ponteiro tipo MAZE.

2. **MAZE *setupMaze(MAZE *maze, int L, int C, int l1, int c1, int k, int m, int e)**

   -Função do tipo MAZE* que efetua a inserção de dados em cada uma das variáveis da estrutura. No final, ocorre o return do ponteiro tipo MAZE, atualizando as variáveis.

4.1.2 MATRIX

Descrita no ficheiro structs.c e definida no structs.h, a estrutura tipo MATRIX, a qual corresponde à matriz útil (matriz que tem um tamanho máximo de [2k_max+1 x 2k_max+1], k_max é o maior k entre os problemas do ficheiro, onde contém a posição de partida e todos os blocos que se pode atingir com k passos, "o diamante"), tem a função de guardar os dados referentes a cada uma das posições da matriz útil. Sendo assim, contém:

- int orig-line: Inteiro que contém o número da linha da matriz original;
- int orig-col: Inteiro que contém o número da coluna da matriz original;

- int new-c1: Inteiro que contém o prémio da posição da matriz selecionada;
- int new-c1: Inteiro que funciona como flag (0 ou 1) para determinar se a posição ja foi visitada anteriormente.

Função utilizada para a criação da estrutura:

1. **MATRIX **createMatrix(int max_lines, int max_cols)**

   - Função do tipo MATRIX** que executa a alocação de memória para a estrutura matrix e inicialização das variáveis que a constituem a '0'. As variáveis de entrada 'int max_lines', número de linhas da maior matriz útil, e 'int max_cols', número de colunas da maior matriz útil, são utilizadas para a alocação da matriz.

4.2 Tipo de Dados

4.2.1 Array

Descrito no ficheiro **structs.c** e definido no **structs.c**, o tipo de dado foi o array devido à sua facilidade de alocação de memória, inserção e remoção de dados em qualquer posição ou até mesmo acesso a um dado independentemente da sua posição no array.

Relativamente ao conteúdo armazenado, dentro do array cada uma das posições guarda a informação referente a uma certa coordenada da matriz útil do problema. Por outras palavras, cada posição do array aponta para a estrutura *tipo MATRIX*.

Para a variante 1 é criado apenas um array, ao qual se deu o nome de 'path', uma vez que não é preciso fazer qualquer tipo de backup/cópia do array, enquanto que na variante 2 é criado dois array's, sendo que um tem o nome de 'path' e o segundo de 'b_path'. O 'path' é o array responsável pelo armazenamento dos passos válidos que podem resultar num caminho para atingir o objetivo da variante 1/2. O 'b_path', existente apenas na variante 2, é responsável pelo armazenamento do melhor caminho alcançado pelo array 'path'.

Para que o acesso ao array seja possível existem várias funções para a sua manipulação, tais como:

1. **MATRIX* newPath(int k_elementos)**

   - Função do *tipo MATRIX** responsável pela alocação de memoria do array, sendo que o tamanho do array irá ser igual ao 'k' (número de passos) + 1 (?????). Depois sua alocação, através de um ciclo 'for' é iniciado a '0' cada variável de cada uma das posições do array.

2. **void increasePath(MATRIX *path, int i, int line, int col, int prize)**

   - Função *void* que tem como finalidade a inserção de informação em cada uma das variáveis da posição 'i' (variável de entrada) do array.

3. **void reduzePath(MATRIX *path, int i)**

   - Função *void* que tem o objetivo de igualar a '0' cada uma das variáveis da posição 'i' (variável de entrada) do array.

4. **void subMAX(MATRIX *path, MATRIX *b_path, int i)**

   - Função *void* utilizada apenas na variante 2, sendo que esta irá fazer uma cópia das variáveis da posição 'i' (variável de entrada) do array 'path' para o array 'b_path'.

5 Subsistemas e Algoritmos

Nesta secção apresentam-se os subsistemas do programa, juntamente com a descrição das funções, e os algoritmos mais importantes para a resolução dos vários problemas. Uma vez que a maioria das pequenas descrições já estão no código, apresentam-se apenas explicações de algumas funções que se consideram de mais relevantes.

5.1 Subsistemas

5.1.1 Funcs.c

O ficheiro **'Funcs.c'** incorpora a maioria das funções necessárias para a resolução do problema do inicio ao fim. Os **ficheiros .h** utilizados neste ficheiro são os seguintes: **Structs.h** e **file.h**; sendo que a declaração das funções utilizadas no ficheiro **'Funcs.c'** foi feita em **'Funcs.h'**. Uma vez que o ficheiro apresenta grande parte das funções do projeto, é possivel dividir este em três partes distintas: avaliação de dados (1), criação do tabuleiro(2) e resolução do problema(3).

(1) Avaliação de dados:

- int first_line(FILE *fp, int *L, int *C, int *l1, int *c1, int *k, int *m, int *ei)
  - Função *int* que lê a primeira linha do ficheiro de entrada de cada um dos problemas que esse contém. Dá return de '1' caso o numero de variáveis diferentes seja igual a 7, caso contrário dá return de '0'.
- int game_mode(int L, int C, int l1, int c1, int k, int m, int ei)
  - Função *int* que faz return do modo de jogo, MODO_MIN ou MODO_MAX, caso o problema seja válido. No caso de este não o ser, dá return de MODE_INVLD.
- int mode_invalid(FILE* fp_in, FILE* fp_out, int L, int C, int L1, int c1, int k, int m, int ei)
  - Função *int* que inicialmente verifica se todos os espaços da matriz apresentam um inteiro. De seguida, imprime no ficheiro de saída que o problema é inválido caso o número de passos, 'k', seja maior ou igual que L*C (número de linhas*número de colunas). Caso contrário, imprime na primeira linha do ficheiro com a informação da primeira linha do ficheiro de entrada.
- void only_invalids(char *fileIn)
  - Função *void* chamada apenas na função getMaxSize caso o ficheiro seja todo ele composto por problemas inválidos. Dentro da mesma é realizado a abertura do ficheiro de entrada e saída, atribuição do nome do ficheiro de saída, impressão da primeira linha dos problemas inválidos e finalmente o fecho do ficheiro de entrada e de saída tal como a libertação de memória alocada para a atribuição do nome do ficheiro de saída.

(2) Criação do tabuleiro:

- void getMaxSize(FILE *fpIn, char *fileIn, int *line, int *cols)
  - Função *void* que cálcula a maior matriz útil através do auxílio da primeira linha de cada um dos problemas que se encontram no ficheiro de entrada '.maps'. Se o ficheiro só tiver problemas for mal definidos, é chamada a função *only_invalids* que escreve no ficheiro de saída a primeira linha de cada problema e o programa termina, não havendo alocação de memória para qualquer matriz.
  - Após o cálculo ter sido executado, a função irá retornar os valores do número máximo de linhas(line) e colunas(cols) da matriz útil.
- void min_max (MAZE* maze, int *min_orig_line, int *min_orig_col, int *max_orig_line, int *max_orig_col, int n)
  - Função *void* cujo objetivo é definir os limites originais da matriz útil na matriz original de cada problema, calculado entre a diferença da menor e maior linha(min_orig_line/max_orig_line) e coluna(min_orig_col/max_orig_col) original.
  - É de ter em conta que o a menor de linha e coluna nunca poderá ser menor que 1 e para a maior linha e coluna da matriz útil não poderá ser maior que L e C, respetivamente.
- void readFromFileToMatrix (FILE *fp, MAZE *maze, int mode)
  - Função *void* tem como objetivo copiar a matriz que se encontra no ficheiro de entrada '.maps' para a estrutura MATRIX.
  - Inicialmente, é cálculado o número de linhas e colunas da matriz útil do problema através do recurso da função min_max, explicada anteriormente. Recorrendo a dois ciclos for's, a matriz útil é preenchida com os seguintes dados: energia da posição, a qual é copiada do ficheiro através do comando 'fscanf', a linha e coluna da matriz original de cada uma das posições e inicialização da variável 'lamp' a 0, sendo que a posição inicial inicia com a variável 'lamp' a 1.

(3) Resolução do problema:

- int play_game(FILE *fp_out, MAZE *maze, int mode)
  - Função *int* que chama as funções necessárias para a resolução do tabuleiro consoante o modo de jogo, ou seja, inicialmente esta função verifica qual a variante através de um 'if' que compara a variável de entrada 'int mode' com a variável global 'MODO_MIN' e 'MODO_MAX'. De seguida, ocorre a resolução do problema através da função mode_min ou mode_max e para finalizar ocorre a confirmação da resolução do problema.

- Caso este tenha sido resolvido corretamente, a função solved é chamada de modo a imprimir os resultados no ficheiro de saída '.paths', seguido do free da memória das pilha(s) criadas até ao momento. Se o programa não foi resolvido corretamente, a função unsolved é chamada para ser impresso o resultado em como não houve solução final no ficheiro de saída '.paths', seguido do free da memória dos vetores criados até ao momento.

- int mode_min(MAZE *maze, MATRIX *path, int l_atual, int c_atual)
  - Função *int* que efetua os cálculos da variante 1, ou seja, procura o primeiro caminho de 'k' passos que atinge a energia final pedida inicialmente. Dentro desta função são chamadas várias funções auxiliares, tais como: upper_bond, sub_mode_min, sendo que dentro deste se encontram as funções positive_update e negative_update. *Na secção 5.2.2 está presente uma explicação mais pormenorizada desta função tal como do seu algoritmo.*

- int mode_max(MAZE *maze, MATRIX *path, MATRIX *b_path, int l_atual, int c_atual)
  - Função *int* que efetua os cálculos da variante 2, ou seja, procura o caminho de 'k' passos com o qual a energia final é máxima. Dentro desta função são chamadas várias funções auxiliares, tais como: upper_bond, sub_mode_max, sendo que dentro deste se encontram as funções positive_update e negative_update. *Na secção 5.2.3 está presente uma explicação mais pormenorizada desta função tal como do seu algoritmo.*

- int upper_bond(MAZE *maze, int l_atual, int c_tual)
  - Função *int* calcula o maior ganho posível no diamante da coordenada recebida. Este ganho é a variavél majorante que é a soma dos k_atual(número de passos restantes à coordenada recebida, k_) maiores prémios positivos e ainda não visitados. Retornando assim o valor de majorante.

- void positive_update(MAZE *maze, MATRIX *path, int l_atual, int c_atual)
  - Função *void* que tem como propósito a atualização de variáveis após a ocorrência de um passo válido na variante 1 ou 2.
  - Contém assim as seguintes atualizações:
    1. Energia final (ef) e diminuição do número de passos restantes(k_r) na estrutura tipo MAZE;
    2. Adiciona uma nova posição no array 'path', através da função increasePath, com os valores da coordenadas da matriz original, energia da posição e ativação da flag 'lamp' (variáveis pertencentes à estrutura tipo MATRIX).

    Para que ocorra as atualizações acima, o número de passos dados até ao momento têm de ser menores que o número de passos máximos e maiores ou iguais a '0'.
  - Para além de atualização de dados, também ocorre a verificação se a variante 1 ou 2 já alcançou a solução final desejada.

- void negative_update(MAZE *maze, MATRIX *path, int l_atual, int c_atual)
  - Função *void* que tem como propósito a atualização de variáveis, tal como a função positive_update, mas nesta as atualizações ocorrem quando há um passo atrás na matriz, sendo válida a sua utilização tanto na variante 1 como na 2.
  - Contém assim as seguintes atualizações:
    1. Energia final (ef) e aumento do número de passos restantes(k_r) na estrutura tipo MAZE;
    2. Coloca na última posição do array 'path', através da função reduzePath, os valores da coordenadas da matriz original e energia da posição a '0' e é desativada a flag 'lamp' (variáveis pertencentes à estrutura tipo MATRIX).
  - Para além de atualização de dados, também ocorre a verificação se a variante 1 ou 2 já alcançou a solução final desejada.

NOTA: Tanto o sub_mode_max, como o sub_mode_min são funções de continuidade da função mode_max e mode_min, respetivamente, não funcionando como funções independentes.

5.1.2 File.c

O ficheiro **'file.c'** incorpora funções necessárias para a validação, abertura, alocação e atribuição do nome de ficheiros. Todas as funções encontram-se devidamente desclaradas no ficheiro **'file.h'**.

- void check_file_name(int argc, char* argv_1)
  - Função *void* que verifica se o ficheiro de entrada é válido.
- void getOutputName(char* fileIn, char ** fileOut)
  - Função *void* que aloca memória para o nome do ficheiro de saída e escreve o nome do ficheiro de saida na variável 'fileOut'
- FILE *AbreFicheiro(char *nome, char*w_or_r)
  - Função void de abertura de ficheiro.

5.2 Algoritmos

As Variantes 1 e 2 foram resolvidas pelas funções recursivas mode_min e mode_max, respetivamente, pela procura em DFS e pela ordem Cima, Baixo, Esquerda, Direita.

As direções de procura estão representadas por condições que em cada qual é verificado se o bloco adjacente na direção x ao bloco atual, tem parâmetros válidos. As setas de retorno à avaliação deste mesmo blocos, estão associadas ao "backtrack".

Para resolver a variante 1 e 2, é preciso avaliar se na posição atual, temos passos por dar e se a sua energia atual é maior ou igual que o objetivo a alcançar, energia mínima atingir ou melhor energia do labirinto. Passando esta condição, usamos a função upper_bond para calcular o majorante e à qual somamos a energia atual e, este resultado será a nossa energia máxima atingivel.

De seguida, avaliamos se a o bloco adjacente Cima é válido da posição atual, se não for passamos para o bloco adjacente Baixo, e seguimos esta lógica para os restantes. Se algum bloco for válido, avaliamos as condições nesse bloco como foi feito na posição inicial e vai ser guardado no caminho atual ou não. Se as coordenadas seguintes ao bloco válido anteriormente forem inválidas, voltamos atrás, backtrack, e avaliamos a próxima direção.

Atravês desta recursividade, se preenchermos o vetor que guarda o caminho atual, significa que a Variante 1 está resolvida e que a Variante 2 já tem um melhor caminho, mas nesta temos de procurar todos os outros caminhos válidos e compárar-mos no final de cada um para avaliar qual é melhor do que o caminho guardado. Após avaliação de todos os caminhos válidos, a Variante 2 termina.

Se o vetor atual não for pelo menos uma vez preenchido, significa que não existe solução para qualquer uma das variantes.

5.2.1 Variante 1

Na figura 2, encontra-se representado um fluxograma a partir do qual é possível perceber o método utilizado para a resolução dos problemas de variante 1.

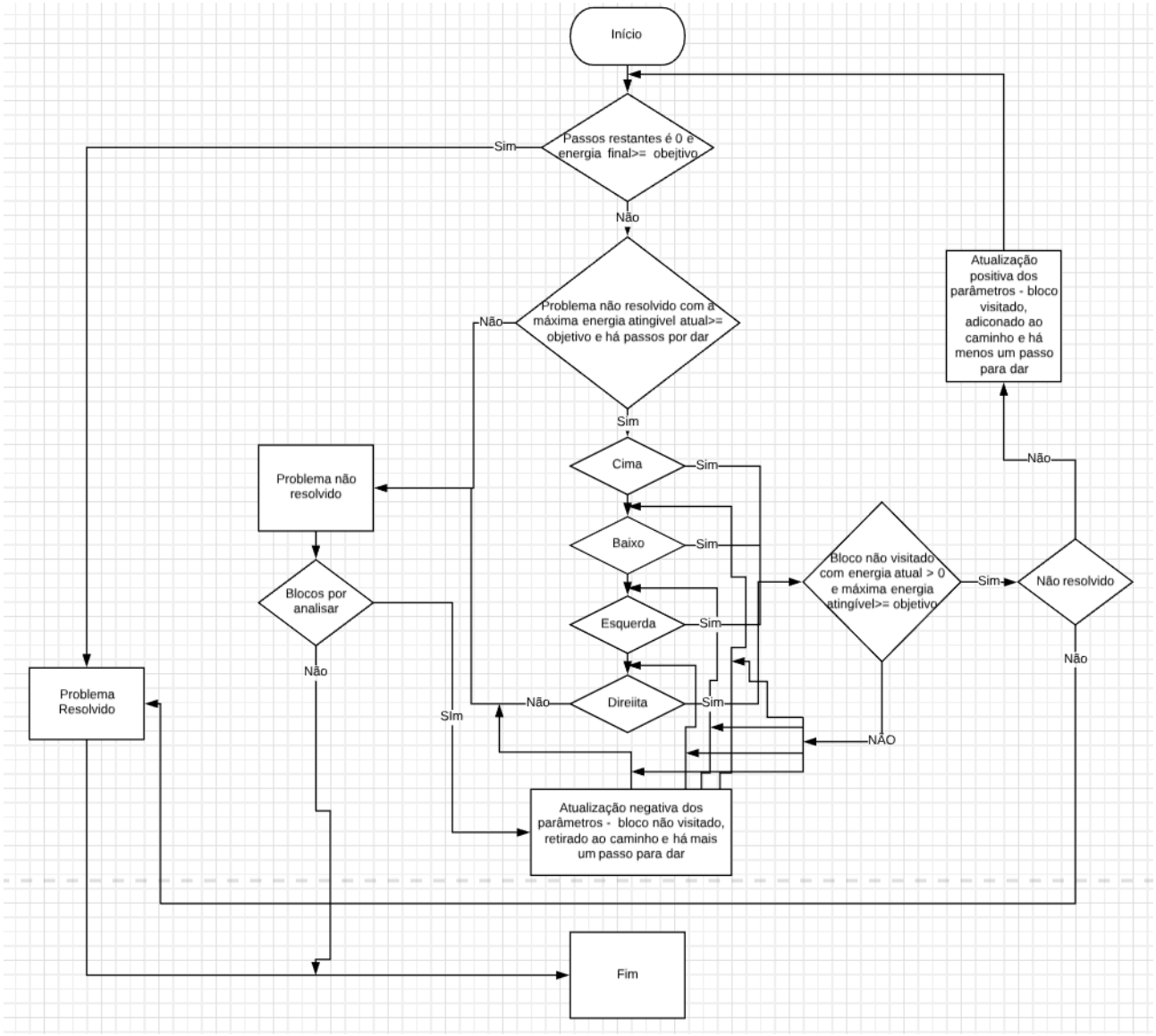

Figura 2: Fluxograma da função mode_min.

5.2.2 Variante 2

Na figura 3, encontra-se representado um fluxograma a partir do qual é possível perceber o método utilizado para a resolução dos problemas de variante 2.

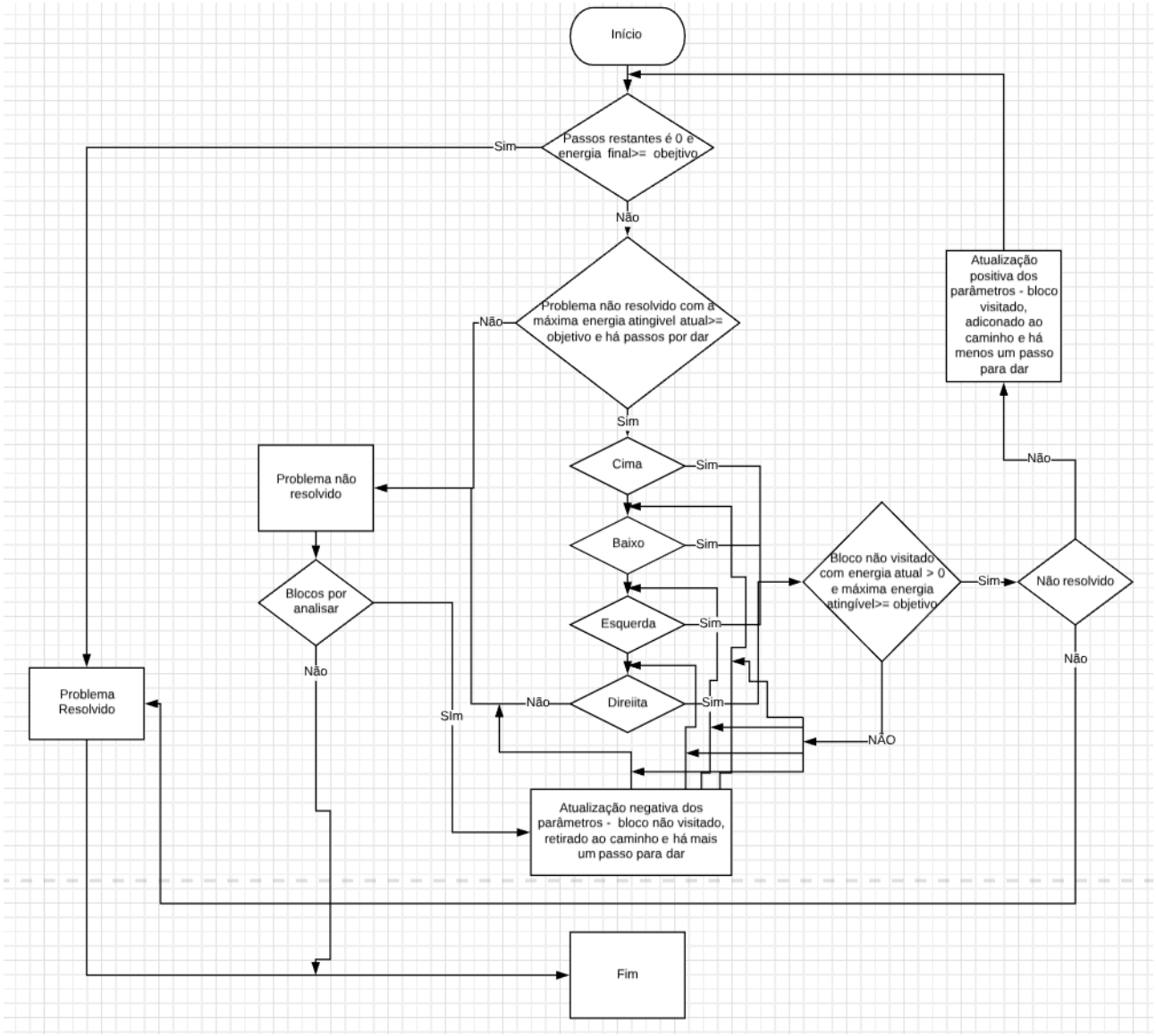

Figura 3: Fluxograma da função mode_max.

6 Análise dos Requisitos Computacionais

6.1 Espacial

Devido ao cálculo da maior matriz útil, maior k, K, do ficheiro, só é alocada uma vez esta matriz do tipo MATRIX de dimensão [2K+1 x 2k+1]. Sendo N = 2K+1, vamos ter uma complexidade de O(N^2) para a matriz.

Para a variante 1, precisamos de um vetor do tipo MATRIX de de k passos de cada problema e para a variante 2 preciso do mesmo vetor, mas duas vezes, por isso para D problemas todos em variante 2, temos uma complexidade de O($(2k)$.

Com isto, a complexidade da alocação de memória para cada problema, num ficheiro, tem complexidade O(N^2+$(2k)$.

6.2 Temporal

Em termos temporais temos de analisar os acessos e escritas de memória. Neste projeto, a pior complexidade de memória é na variante 2 como também em termos temporais.

Devido à imposição de ser produzido um caminho com k passos, onde k é menor que L*C, nunca vamos aceder a todas as posições da matriz útil, só ao limite diamente(losango apresentado na secção Exemplo) que tem $(2k+1)^2$-(2k+1)/2 blocos. Logo, sendo N = 2k+1, a complexidade de acessos é O(N^2-N/2). Mas nesta variante, vamos usar dois vetores para guardar os blocos e no pior dos casos é que achava L*C/k-1 caminhos diferentes

e que o melhor deles seja o último analisar, por isso vamos estar sempre alocar e a libertar memória destes vetores. Os dois vetor ao longo da resolução do problema alocam e libertam 2*L*C/k-1. Logo, a complexidade de um vetor em termos temporais será O(2*L*C/k-1).

Logo, a complexidade temporal total é O(N^2-N/2+2*L*C/k-1).

7 Análise crítica

Considera-se que este projeto foi um sucesso, visto que passou nos 20 testes do corpo docente, ou seja, realizou-os abaixo dos limites de tempo e memória estabelecidos. No entanto, foram necessárias algumas tentativas até se obter este resultado.

Inicialmente, o grupo esteve alocar a matriz por completo, o que consumia muita memória e em termos de resolução, os problemas eram resolvidos com um intervalo temporal muito elevado. Após, testarmos mais ficheiros que apresentado maiores matrizes, concluimos que não precisamos de alocar a matriz original do problema e com isso, calculamos a matriz util para alocar a matriz. Mas em termos temporais, o programa ainda estava muito lento, porque percorria todo o diamante à procura de solução e a o uso do majorante veio trazer novas restrições à procura e com isso foram eliminados enumeros casos inválidos, o que aumentou o processamento de procura e também menos custo de memória.

8 Exemplo

As funções para resolver as variantes usam o algoritmo de procura DFS, como já mencionado, pela seguinte ordem: cima(linha anterior, coluna atual), baixo(linha seguinte, coluna atual), esquerda(linha atual, coluna anterior), direita(linha atual, coluna seguinte).

Para as demonstrações da Variante 1 e 2, é utilizada a mesma matriz de tamanho [7x4], posição inicial (3,3), 4 passos e energia inicial de 5.

Para matrizes substancialmente grandes e através do determinação da matriz útil, os problemas seriam resolvidos de forma igual em cada variante, porque temos todas os blocos necessários conforme o enunciado para a sua resolução.

8.1 Variante 1

O objetivo deste problema é alcançar um caminho de 4 passos de energia final maior ou igual que o objetivo de 12.

Nota: Sendo a posição inicial (3,3), vamos substituir o seu valor por 5, assinalizando amarelo, para uma melhor perceção da resolução do problema. No programa, o seu valor mantém-se a -1.

Para esta variante, precisamos de um vetor para guardar os passos válidos, que aqui chamamos ATUAL, tendo o nome original no programa de *path*.

Iniciamos a função com a posição inicial que avalia, primeiramente, se a soma da energia inicial com o majorante atual(1+10+10+0=21), calculado na função *upper_bond*, representado em 1 pelo losango amarelo, é maior ou igual que a objetivo e se os passos restantes que são iguais aos iniciais, 4, são maiores que zero. Neste caso, é válido e agora iremos analisar os blocos adjacente pela ordem já mencionada, cima, baixo, esquerda, direita.

Cada bloco adjacente(no programa tem coordenadas (l_direction, c_direction)) à nossa coordenada atual, neste primeiro passo (3,3) que é a posição inicial, tem de seguir alguns parâmetros para ser válido e com isso adicionado a um possível passo para o nosso caminho representado no vetor ATUAL:

1. Bloco não visitando;
2. A energia atual mais a energia do bloco adjacente é maior que zero;
3. A energia atual mais o majorante do bloco adjacente é maior ou igual que o objetivo.

Se estas condições forem válidas e o problema ainda não estiver resolvido, o bloco é adicionado ao caminho. Coordenada atual

- Cima (2,3), majorante calculado dentro do losango cinzento - não passa à 2 condição, logo não é um passo válido. Avançamos.
- Baixo(4,3), majorante calculado dentro do losango verde - passa a todos os teste, logo é um passo válido e atualizamos o caminho e a energial final.

Quando um passo é válido, o a sua coordenada torna-se a atual e processa-se do mesmo modo que como na posição inicial.

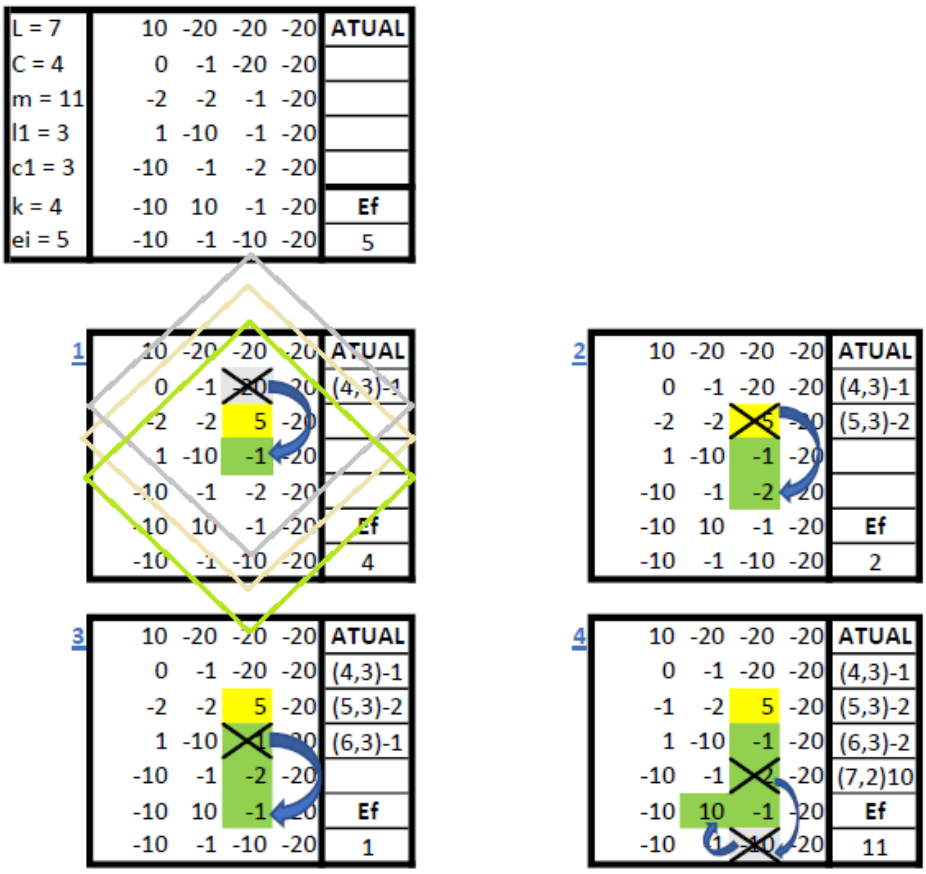

Figura 4: Exemplo da função mode_min.

8.2 Variante 2

O objetivo deste problema é alcançar o caminho 4 passos de com maior energia final.

A variante 2 é um modo mais complexo que a variante 1. Aqui o objetivo não é fixo, é váriável à medida em que encontramos caminhos com k, neste caso 4, passos válidos(cumpre as três condições descritas na variante 1) que são guardados no vetor ATUAL(no programa chamado de path) e substituídos no vetor BEST(no programa chamado de b_]path), se a energia final do caminho em ATUAL é maior que a do BEST, passando a maior energia a do vetor ATUAL que é guarda na variavél Eb, energia do caminho BEST.

Após, o vetor ATUAL encontrar um caminho válido, este retorna à coordenada anterior, passo k-1 em relação ao último, para analisar outros blocos adjacentes ao bloco k-1. Se estas outras opções forem inválidas, ele retorna ao anterior e repete o mesmo processo. Sempre que voltamos atrás no caminho para analisar outras opções, é retirado do vetor o ultimo passo(4 e 5).

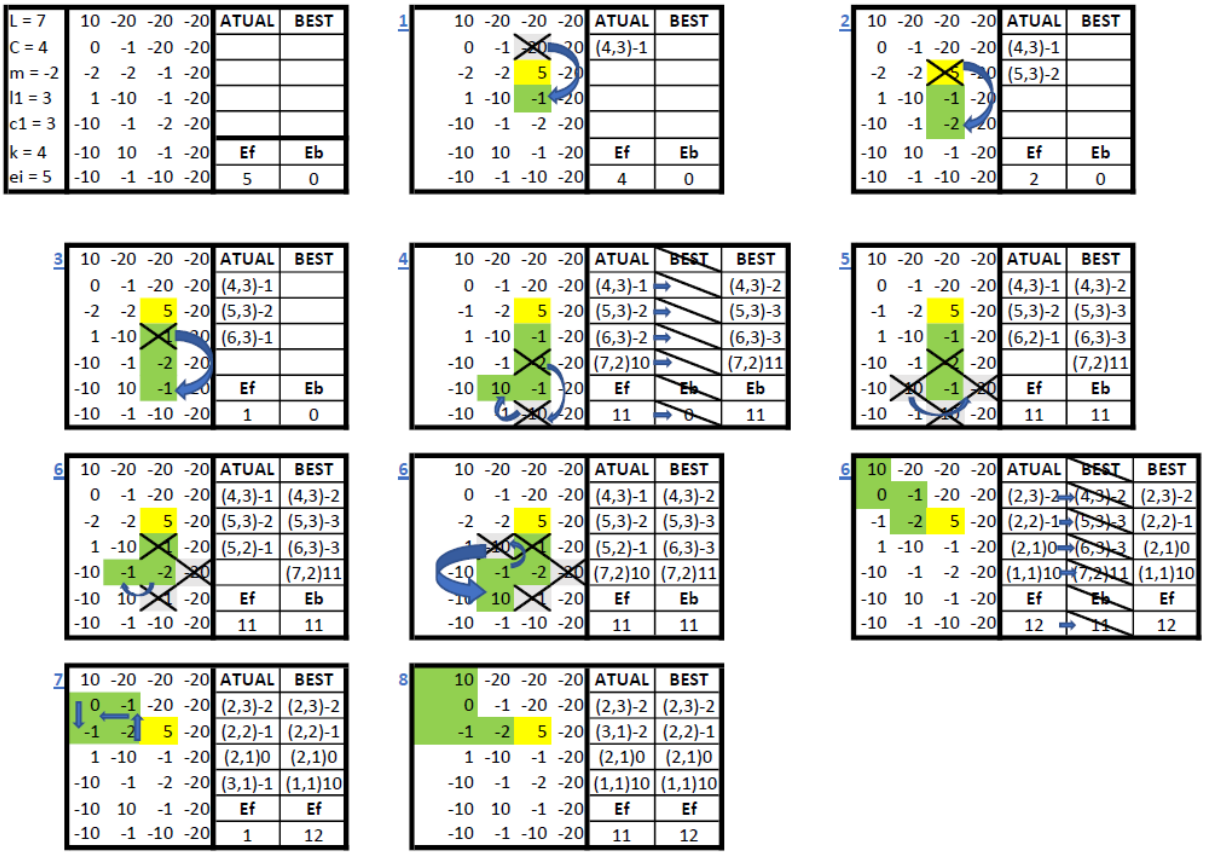

Figura 5: Exemplo da função mode_max.

9 Bibliografia

Slides das aulas teóricas da disciplina de Algoritmos e Estruturas de Dados, 2ºsemestre de 2017/2018, MEEC, da autoria do Professor Carlos Bispo.