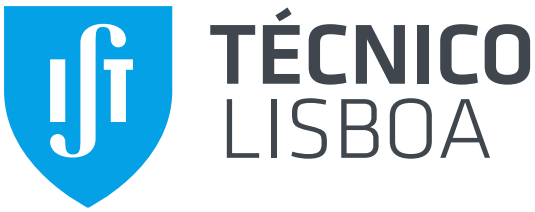

Engenharia Eletrotécnica e de Computadores

Algoritmos e Estruturas de Dados

2º Ano, 2º Semestre – 2017/2018

Relatório

aMazeMe

Identificação dos Alunos:

Grupo: 16

Nome Ana Isabel Ventura Teixeira **Número:** 83997

email: ana.isabel@tecnico.ulisboa.pt

Nome: Diana Martins de Oliveira **Número:** 84028

email: diana.m.oliveira@tecnico.ulisboa.pt

Docente: Carlos Bispo

Índice

Descrição do Programa

O problema enunciado consiste na resolução de puzzles estáticos que possam ser representados por mapas retangulares, ou seja, por matrizes de dimensões adequadas, constituídos por células associadas a um valor de energia (custo ou prémio de "entrar" nessa célula). Pretende-se percorrer o mapa, fornecido num ficheiro de texto de extensão ".maps", por um certo caminho a obter, de forma a cumprir um de dois objetivos: atingir ou superar o valor de energia específico dado ou atingir a maior energia possível para o mapa dado.

Estes objetivos devem ser alcançados de acordo com as seguintes restrições: o caminho produzido possui exatamente k passos, o agente não pode prosseguir caminho se a sua energia deixar de ser positiva e o caminho percorrido não pode usar uma qualquer célula mais do que uma vez, inclusive a célula de partida.

Desta forma, e tendo em consideração o objetivo pedido e as restrições todas, indicar-se-á num ficheiro de saída ".paths" se não existe solução (representado por -1) ou se esta existe (representado pela energia final do agente) e, por conseguinte, o percurso do agente juntamente com os respetivos custos/prémios de cada célula serão impressos no respetivo ficheiro.

Abordagem ao Problema

Em primeira instância, estabeleceu-se uma relação entre a resolução do puzzle e uma representação em árvore, cuja raíz é a posição inicial fornecida no ficheiro de entrada, seguindo-se-lhe um apelo ao modelo do algoritmo DFS ("depth-first-search") simples. Assim, é feita uma procura em profundidade que pode resultar quer numa solução válida quer num caminho errado, em cujo caso voltar-se-ia ao último momento decisivo em que as restrições impostas eram totalmente cumpridas, explorando-se um caminho alternativo, até então não analisado, e verificando-se, portanto, para este, a validade da sua solução.

Com a implementação deste algoritmo, inicia-se a resolução do puzzle na posição de partida, com a energia do agente, também fornecida no ficheiro de entrada, e começa-se por determinar a direção a seguir, dependendo do concluído da avaliação de cada uma das quatro (ou três, para todas as células que não a primeira) possíveis. Esta análise é feita comparando a energia total das quatro (ou três) diferentes possibilidades, decidindo-se apenas pelas que resultam numa energia total, associa dada ao upper bound, maior que a respetiva energia de target.

Após esta decisão, o procedimento a tomá-la é repetido consecutivamente até que se obtenha um caminho com k passos. Neste momento deverá verificar-se se a energia final do agente é a desejada, sendo no caso negativo ainda preciso concluir se tal se deve à impossibilidade de obter tal energia, para as condições iniciais dadas, ou se ainda existem caminhos por percorrer e potencialmente obter-se uma solução válida.

Arquitetura do Programa

a) Primeiros passos

O problema implementado foi dividido em dois subsistemas diferentes, um dedicado às estruturas de dados e às operações sobre elas realizadas, o ficheiro matrix.c, e outro com todas as funções que auxiliam à resolução do problema e que não se inserem no subsistema anterior, o ficheiro utils.c.

Por outro lado, e após alguma deliberação concluímos que o programa deveria ser divido em três partes lógicas, ou seja, no que diz respeito à resolução concreta do problema sem incluir as necessárias verificações de início do programa, criação do ficheiro de saída e, no fim, respetivo fecho dos ficheiros de saída e de entrada e libertação de memória alocada. Estas operações, ilustradas na Figura 1, são feitas na função main (ficheiro main.c), excepto parte da libertação da memória alocada que é realizada na função Solve, do ficheiro utils.c, representada na Figura 2.

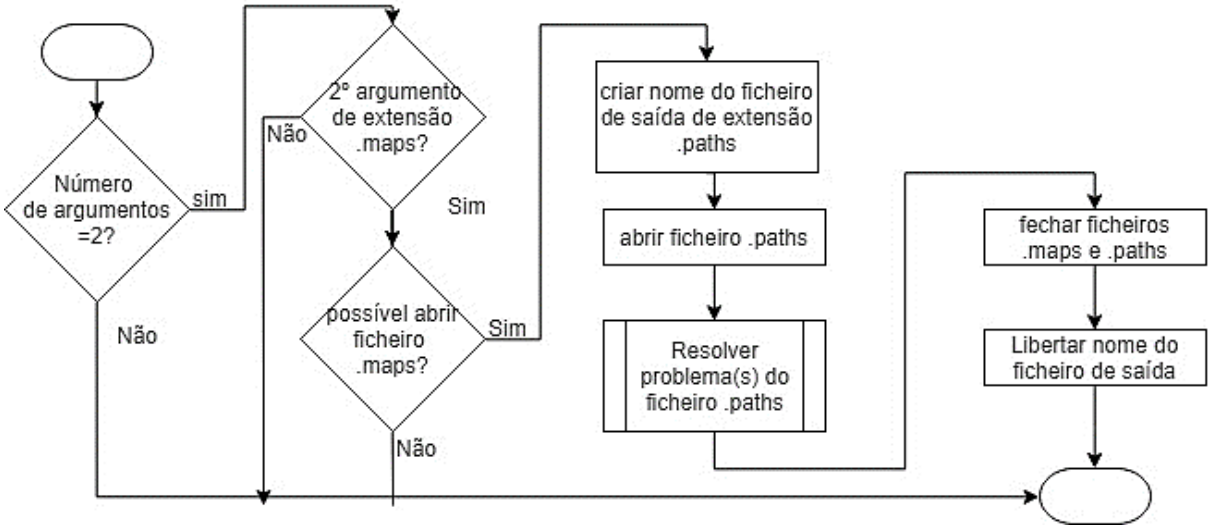

Figura 1 - Fluxograma da função main do programa.

Numa primeira parte, deverá proceder-se à leitura dos dados fornecidos na primeira linha no ficheiro de entrada e à respetiva alocação e preenchimento da memória estritamente

necessária à resolução do problema. Mais concretamente, verifica-se as coordenadas iniciais se encontram dentro dos limites da matriz e se os valores de k ($k \in [0, L \cdot C[$), do objetivo (obj $\in \{-2\} \cup]0, \infty[$) e a energia inicial do agente ($E \in \mathbb{N} \setminus \{0\}$) estão dentro dos valores esperados. Se este não for o caso, é dispensável a alocação de memória e salta-se para a reescrita da primeira linha do ficheiro de entrada no de saída, evidenciando a má definição do problema. Salientando-se que no caso em que $k \geq L \cdot C$ existem mais passos a dar que elementos na matriz, uma vez que a célula inicial não pode ser contabilizada para o caminho.

No entanto, se for o caso, passa-se à determinação da dimensão da matriz local, que compreende o losango de raio k, centrado na célula de coordenadas correspondentes às iniciais, a partir dos cantos do losango (as quatro células mais afastadas, na vertical e na horizontal). Este processo dá-se porque, independentemente da dimensão da matriz no ficheiro <.maps>, ela pode não ser útil na sua totalidade, visto que o número de passos no caminho é fixo e fornecido e pode não ser suficiente para alcançar todas as células na matriz dada.

A esta secção correspondem a sub-rotina “Resolver problema(s) do ficheiro .paths” da Figura 1 e está associada à função Solve do ficheiro utils.c, especificada na Figura 2. Nesta figura é evidenciado como esta sub-rotina contém mais duas sub-rotinas, correspondentes às duas partes seguintes da resolução do problema, dependendo do objetivo pretendido.

b) Resolução do Problema

Como já referido, as matrizes fornecidas nos ficheiros de entrada não se podem assumir como úteis na sua totalidade, uma vez que, dependendo do número de passos a dar, definido por k, é possível só parte delas serem usadas ao longo do programa. De forma a gastar apenas a memória estritamente necessária para a solução de cada problema, antes de se alocar a matriz respetiva, determinam-se as fronteiras da matriz útil/local e a sua dimensão.

Dependendo, então, da posição inicial, do raio k e das suas dimensões LxC, criam-se três possibilidades de “corte” da matriz universal, como de pode verificar pelos exemplos da Figura 3. A sua distinção é feita a partir da posição inicial (l,c), calculando-se as células mais afastadas desta, num raio de k, nos quatro sentidos. Fazendo l-k, l+k, c-k e c+k (cima, baixo, esquerda e direita, respetivamente), determinam-se as posições mais afastadas na vertical e na horizontal que, se alguma não se encontrar dentro das dimensões originais (LxC), deverá ser ajustada a 0, L, 0 e C, respetivamente.

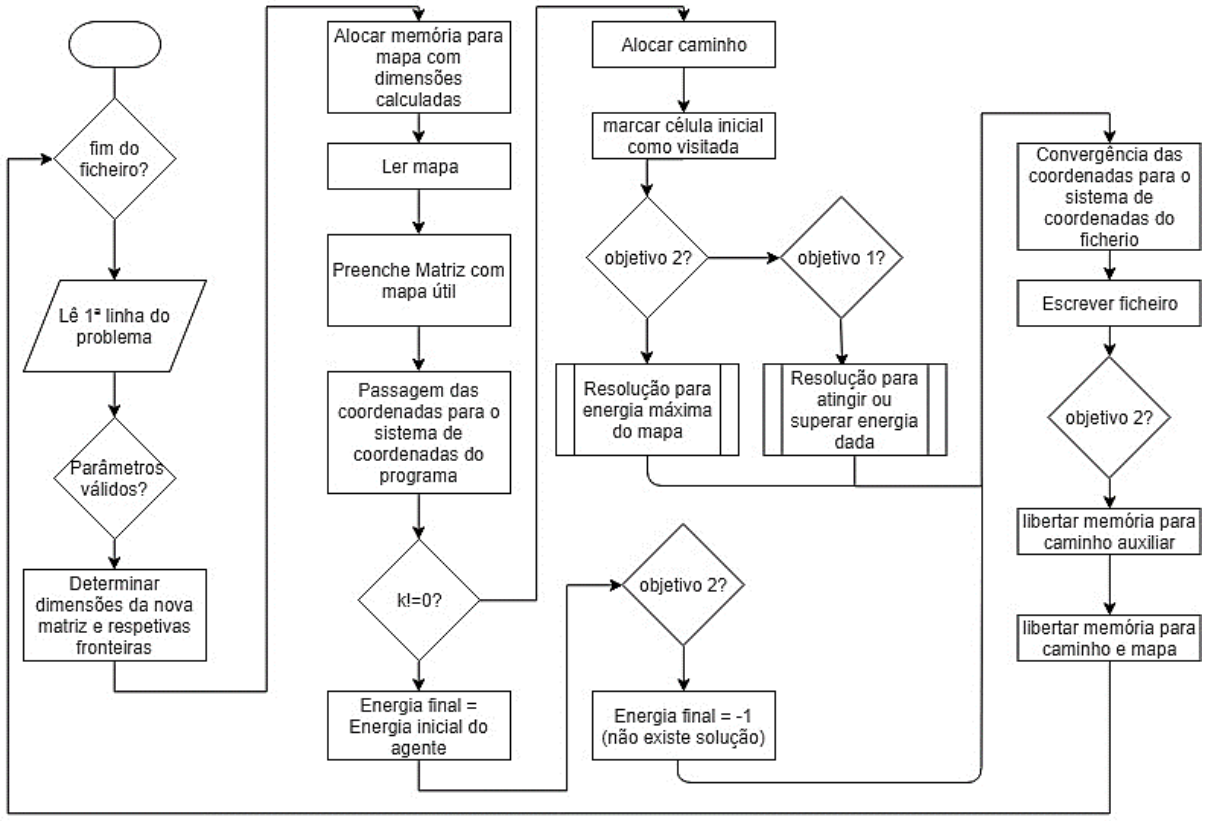

Figura 2 - Fluxograma da função Solve do programa.

De acordo com as novas coordenadas mínimas e máximas na vertical e na horizontal, torna-se possível determinar as dimensões da matriz local, subtraindo, de cada categoria, as mínimas das máximas, e obtendo-se L' e C', limitadoras do tamanho da nova estrutura onde se alocará a informação dita útil, e a conversão das suas coordenadas, evocando a transformação vetorial – evidenciada em cada exemplo da Figura 3 pelo vetor colorido - de coordenadas.

É, ainda, de salientar que para além da conversão da matriz fornecida para a matriz útil é necessário passar para o sistema de coordenadas do programa, isto é, transpor de $[1, L]$ e $[1, C]$ para $[0, L-1]$ e $[0, C-1]$ o que apenas envolve o decremento de uma unidade para cada coordenada.

Após o corte da matriz universal e da alocação (função NewMatrix) e do preenchimento da matriz local pode iniciar-se a procura de solução para o problema, referente a uma das duas últimas partes, a um dos dois diferentes objetivos: atingir uma energia mínima dada (objetivo 1) ou atingir a energia máxima possível (objetivo 2), em k passos. Caso k=0, não há passos a dar e a solução dependerá da relação entre a energia inicial e a energia target: caso $E \geq E_T$ a solução é igual a E, caso contrário não existe solução.

Para qualquer um dos objetivos é preciso alocar memória para um vetor que guarde o caminho, de dimensão k, (função CreatePath, matrix.c) sendo que no segundo objetivo é necessário um outro, de dimensão k+1, para servir como referência, cuja célula extra é a energia final do caminho guardado. Acrescente-se ainda que a célula inicial passa a ter a sua flag visited acionada (função SetCellUsage), de forma a que não seja utilizada na exploração do caminho.

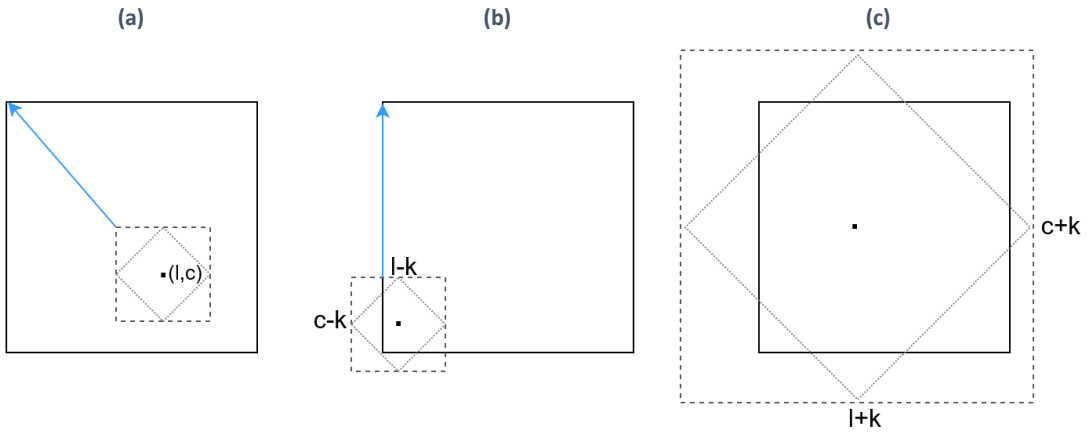

*Figura 3 - Exemplos de cortes realizados num mapa LxC para obter a matriz local que **(a)** seja um quadrado perfeito [(2k+1)x(2k+1)], por as suas fronteiras não ultrapassarem as originais; **(b)** tenha dimensões menores que [(2k+1)x(2k+1)], no caso contrário ou **(c)** tenha as dimensões originais, por o raio ser tal que inclui toda a matriz independentemente da posição inicial.*

i. Resolução do objetivo 1 – Energia Mínima

Para o caso do primeiro objetivo, tomou-se a decisão estratégica de criar uma função recursiva, a MinEnergy, que implementasse o algoritmo de procura em profundidade com branch and bound de forma mais global possível. Para atingir esta meta, optou-se por um switch case em que cada valor da variável está associado a uma direção (cima, baixo, esquerda, direita), como se pode verificar pela Figura 4.

Primeiramente, é necessário determinar se o passo em que nos encontramos, k0, está dentro dos limites de k, i.e., no intervalo [0,k[. Se não estiver, estamos perante uma de duas hipóteses: já não existem mais passos para dar e a energia total do agente é superior à energia target, pelo que o problema tem solução e já não é preciso efetuar mais nenhuma operação sobre o mapa, ou a energia total do agente é inferior ao target e deve regressar-se à última decisão cujas direções válidas não tenham sido todas exploradas. Nesta última ocorrência, deve retirar-se a energia da célula presente à energia total e remover esta célula do caminho, i.e., colocar a sua flag visitated a 0.

No entanto, estando dentro dos limites de k, passa-se à procura em profundidade na árvore. Para este fim, faz-se uso de um ciclo *for* para ir incrementando a variável que representa cada direção (0 equivale a ir para cima, 1 a ir para baixo, 2 corresponde a ir para a esquerda e 3 para a direita).

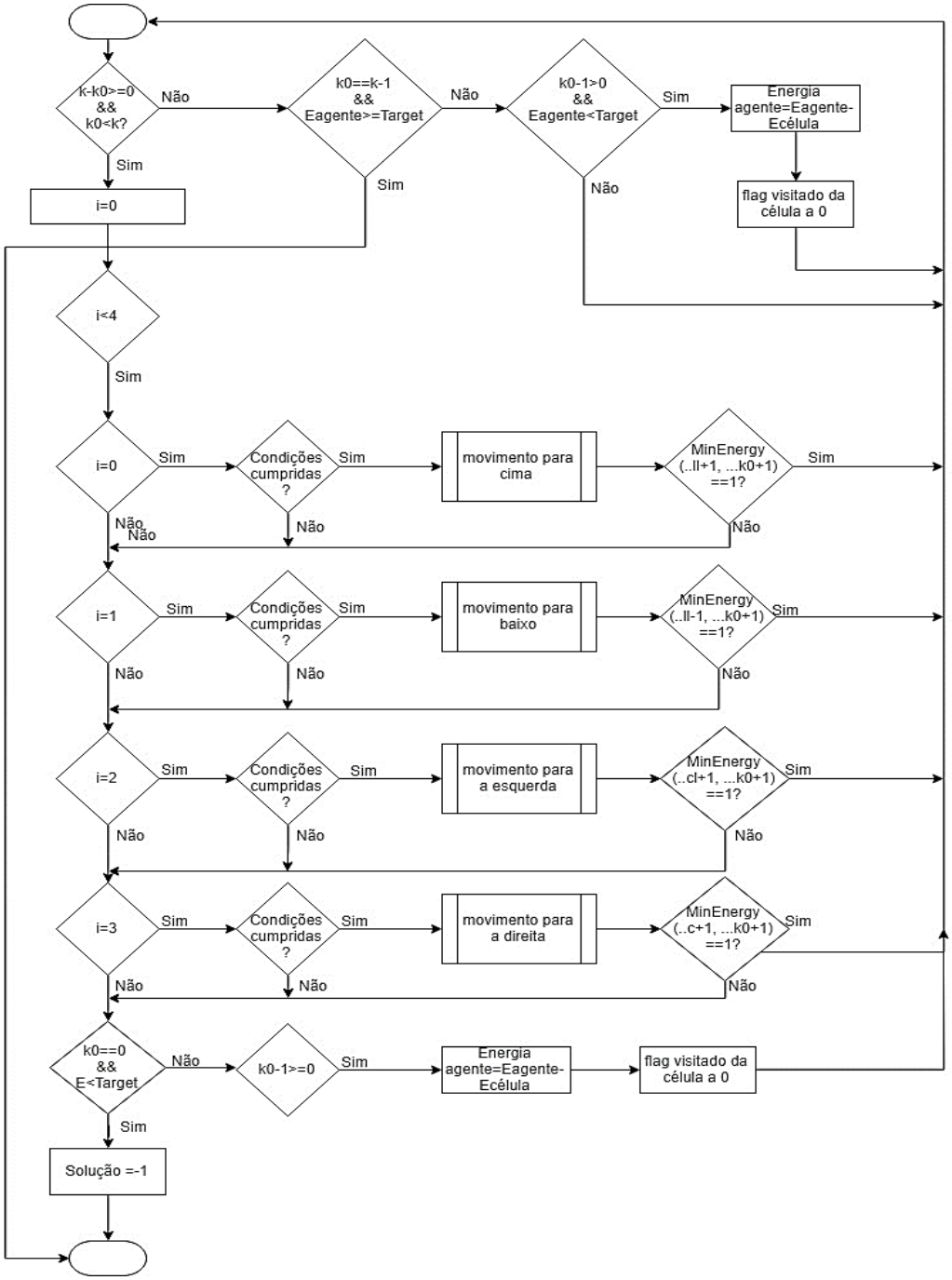

Figura 4 - Fluxograma da função MinEnergy do programa.

Para realizar o movimento, começa-se por se certificar se este é válido (através da função Conditions, no utils.c), verificando se as coordenadas da célula para a qual nos vamos deslocar não façam parte do caminho até então registado, a sua energia não mate o agente e o Upper Bound, determinado na função AddAll, seja superior ou igual à energia target. Se todos estes critérios forem cumpridos, indicado pelo valor de *return*, realiza-se efetivamente o movimento na direção desejada (função move), adiciona-se o custo/prémio dessa célula, guarda-se a sua informação no vetor do caminho e ativa-se a respetiva flag visited, como se pode ver na Figura 5.

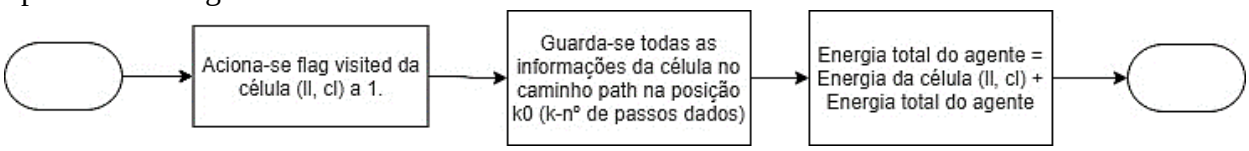

Figura 5 - Fluxograma da função Move do programa.

Nesta altura, chama-se a função MinEnergy mas incrementado/decrementando o argumento associado à linha ou coluna da célula para onde nos deslocámos e adicionando ao número de passos tomados uma unidade. Caso esta retorne o inteiro 1, tivemos sucesso e avançamos, caso contrário tem que se voltar ao último momento em que as direções não foram todas testadas, sendo primeiro imperativo retirar o prémio/custo da célula onde nos encontramos, anteriormente adicionado, e colocar a sua flag a zero.

Todo este processo de determinação do sentido a tomar e de movimento é fundamentalmente igual para qualquer uma das direções, com a pequena alteração das coordenadas, i.e., o incremento/decremento da linha ou coluna dependente do deslocamento escolhido. Esta estrutura de código impede a repetição do mesmo e simplifica a função MinEnergy e a sua implementação.

No caso de se investigar a árvore toda e não se ter encontrado um caminho que cumpra todas as condições e atinja o objetivo pretendido, indica-se que o problema não tem solução, ao associar a energia ao valor -1, e sai-se da função MinEnergy.

ii. Resolução do objetivo 2 – Energia Máxima

Já para o segundo objetivo, decidiu-se que a melhor estratégia, tanto para a generalidade do código como para a sua estruturação, seria fazer uso da função já criada para o primeiro objetivo, como se pode ver na figura 6. Consequentemente, a função MaxEnergy é recursiva,

tal como a MinEnergy, e tem como base o algoritmo de procura em profundidade com branch and bound, que serão descritos na secção "Descrição dos Algoritmos".

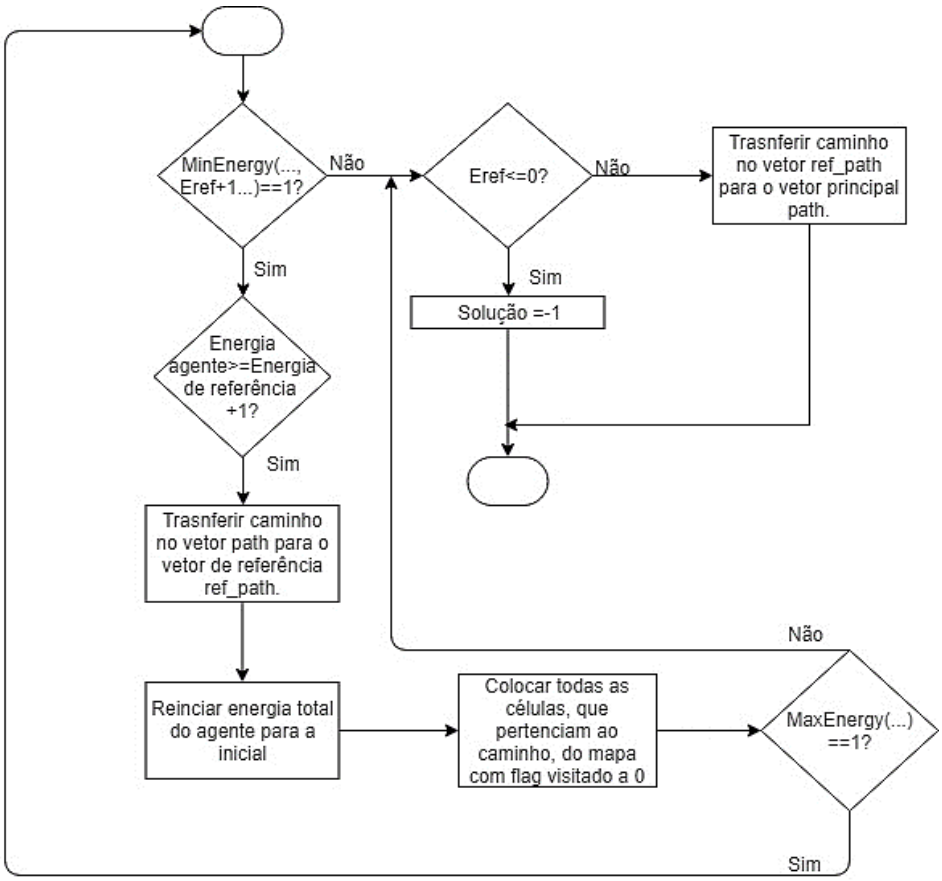

Figura 6-Fluxograma da função MaxEnergy do programa.

Como tal, começa-se por encontrar um caminho, através da função MinEnergy, em que o agente não morra, e a sua energia final seja superior a 0, isto é, superior à energia target pertencente ao caminho de referência guardado num segundo vetor fixo de dimensão k+1. Havendo sucesso nesta procura, passa-se o caminho guardado no vetor principal para o de referência, usando a função transfer_path, no matrix.c, visto que o primeiro tem energia maior. De seguida, chama-se a função MaxEnergy com todas as células do mapa como não visitadas (função Visited) e com a energia do agente de volta ao valor inicial fornecido no ficheiro de entrada (função SetMapEnergy) com a intenção de encontrar um outro caminho que seja melhor, em termos energéticos, que aquele já descoberto.

Este procedimento deve ser executado tantas vezes quanto necessário até que toda a árvore seja explorada, isto é, até que todas as possíveis decisões que dêem origem a um caminho válido sejam tomadas, e se chegue à energia máxima do mapa. Neste caso, deverá transferir-se a informação das decisões tomadas do vetor auxiliar (ref_path) para o vetor

principal (função path_transfer) e passar à escrita da solução. No caso contrário, ocorre que não existe solução.

c) Término

Se a solução existir, é preciso escrever o caminho no ficheiro de saída com as coordenadas de acordo com a matriz original, pelo que se lhes aplica a transformação vetorial inversa à descrita anteriormente, na Figura 3, e incrementa-se uma unidade para as coordenadas passarem para o sistema do ficheiro – a partir da função CoordinateConvergence, do ficheiro matrix.c.

Por fim, procede-se à escrita da solução no ficheiro de saída <.paths> de acordo com o formato pedido, ou seja, reescreve-se a primeira linha do puzzle no ficheiro de entrada seguido de um "-1" no fim da mesma se o puzzle não tiver solução ou seguido da energia total do agente se o puzzle tiver solução, juntamente com todas as coordenadas e respetivos custo/prémios do caminho. Esta é realizada utilizando a função WriteFile, do utils.c.

Acrescente-se, ainda, que é necessário libertar toda a memória alocada durante esta resolução, mais concretamente a memória da estrutura t_matrix (recorrendo à função FreeMatrix, do ficheiro matrix.c) e t_cell e, para o caso do objetivo 2, para a estrutura auxiliar t_cell. Neste momento, verifica-se se já se chegou ao final do ficheiro, se sim o programa acaba e, todavia, se ainda houver mais puzzles a resolver volta-se ao início da função Solve.

Descrição das Estruturas de Dados

Tal como já foi referido, a estrutura de dados mais adequada para guardar o mapa é uma matriz, ou seja, um vetor de vetores, visto que assim diminuem-se os custos de memória e de acesso.

No que diz respeito ao grafo, escolheu-se não se representar em memória a totalidade da árvore uma vez que seria impossível fazê-lo e ainda garantir a obediência aos limites de memória (100Mb), valorizando-se, assim, uma eficiente gestão de memória e de resolução (podendo, no caso contrário, ter-se uma quantidade de decisões/vértices elevada). Isto torna-se possível porque o grafo em questão é acíclico e, portanto, apenas se guarda em memória o caminho, ou seja, um vetor de estruturas em que cada uma representa uma célula do percurso do agente.

Em termos de implementação, criaram-se duas estruturas base: uma com a energia do agente e o mapa e outra com as informações necessárias de cada célula, como se pode ver na tabela 1.

Tabela 1 – Descrição das estruturas e limites de algumas variáveis constituintes.

Struct _t_cell	**Descrição**	
int visitited	Flag que indica se a célula faz parte do caminho ou não	{0,1}
int value	Custo ou prémio associado à célula	-
int l	Linha onde a célula se situa na matriz	l∈[1, L]
int c	Coluna onde a célula se situa na matriz	l∈[1, C]
Struct _t_matrix		
int energy	Energia do agente	E>0
struct _t_cell **map	ponteiro para a Matriz de dimensão LxC com o mapa	-

A primeira referida, considerada a principal, é uma estrutura composta por um inteiro, que guarda a energia do agente, e um duplo ponteiro, que origina uma tabela de estruturas t_cell, sendo que cada uma corresponde a uma única célula do mapa. Optou-se por estas gestão de memória devido ao acesso facilitado à informação particular e consequente uso. Embora isto nos aumente a memória usada, o acesso direto às coordenadas e ao custo/prémio permite uma mais rápida escrita do caminho e simplificação da estrutura do código – uma vez que só é necessário criar funções simples para aceder a estes dados e aplicá-los diretamente no outro ficheiro .c.

Por fim, o caminho é um vetor de tamanho fixo uma vez que se sabe sempre a quantidade de células a ser usado no mesmo e é composto por k estruturas. Acrescente-se, ainda, que para o caso de o objetivo ser atingir a máxima energia possível é necessário criar um segundo vetor de estruturas que servirá como referência com k+1 posições de forma a tornar-se mais simples a leitura da energia final do agente, isto é, na primeira posição (posição 0) do vetor apenas se regista esta informação e nas restantes k posições estão registadas as informações de cada célula desse mesmo caminho. Este vetor de referência é o primeiro caminho encontrado e só será reescrito se a energia do vetor principal for superior de maneira a que se possa percorrer toda a matriz sem perder o melhor caminho e a respetiva energia máxima já atingida.

Descrição dos Algoritmos

a) Algoritmo para a procura

O principal algoritmo do programa é o algoritmo DFS, usando branch and bound para encurtar o número de caminhos a percorrer, ou seja, diminuindo o tamanho da árvore a percorrer tal como o tempo de procura.

Numa descrição mais aprofundada, utiliza-se procura em profundidade recursivamente com a intenção de se poder facilmente andar na árvore tanto para a frente como para trás durante a procura do caminho. Ou seja, sempre que as condições permitirem efetua-se um movimento nessa dada direção e chama-se a função recursiva, se esta tiver sucesso avança ao retornar 1, caso contrário e se já estiverem esgotadas todas as direções a partir da mesma célula, retorna-se 0 e voltamos à recursiva anterior, ou seja, ao vértice anterior de maneira a explorar as restantes direções. Isto é feito sucessivamente até que se encontre um caminho válido ou se tenha percorrido a árvore toda e nenhum caminho válido tenha sido encontrado pelo que o puzzle não tem solução.

É de ter em especial atenção que sempre que se avança acende-se a lâmpada do respetivo vértice para sinalizar que este pertence ao caminho, no nosso caso a flag visited é ativada quando tem valor 1. Porém, ao voltar ao último momento em que se pode escolher decisões alternativas àquelas já tomadas é necessário apagar a lâmpada de cada vértice que deixará de pertencer ao caminho visto que estes poderão a vir ser usados para a caminho mais à frente. O algoritmo encontra-se representado nas imagens abaixo, estando o apagar e acender da lâmpada mostrado pelas mudanças de cor e verde-branco e branco-verde respetivamente.

Ainda se recorre ao algoritmo branch and bound para que não seja necessário percorrer a árvore completa, isto é, só avançamos para um caminho se neste se poder obter uma energia final igual ou superior à pretendida (ou superior à já obtida anteriormente no caso do objetivo 2). Para saber isto, soma-se à energia que o agente já possui a energia da célula para a qual nos queremos deslocar e a soma de todos os prémios num losango de raio k-x, sendo x o número de passos já tomados, centrado nessa mesma célula. A primeira direção em que esta conta dá um resultado favorável é a primeira a ser explorada. Isto também se encontra evidenciado nas imagens abaixo.

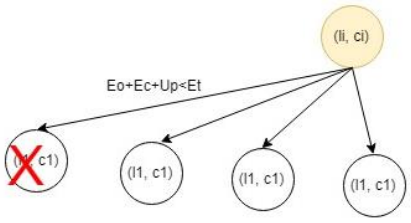

Imagem 1 - 1º teste: não é possível avançar porque este caminho não "promete" um ganho máximo de energia maior que a energia pretendida

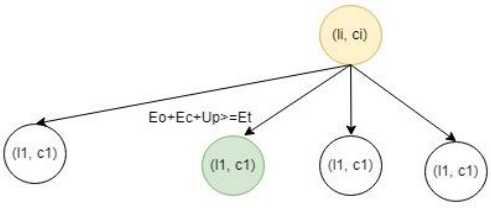

Imagem 2 - 2º Teste: é possível avançar porque há promessa de uma energia superior à Energia de target (Et).

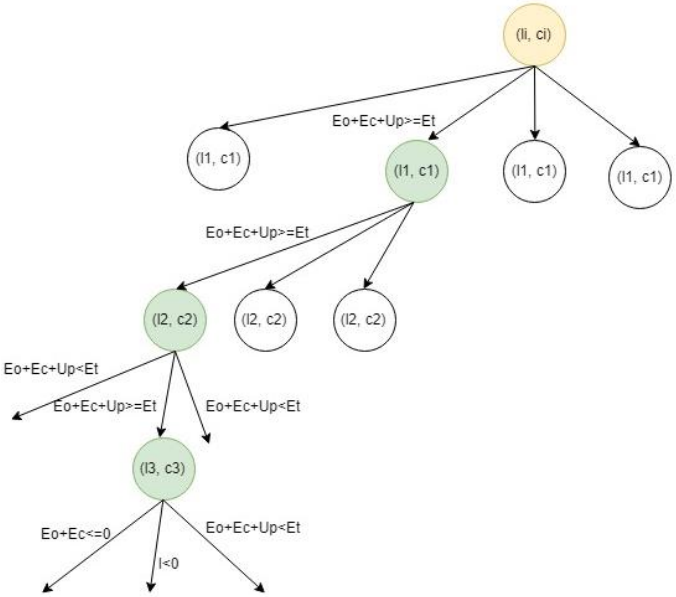

Imagem 3 - Ao chegar a um ponto em que todas as opções já foram testadas e não é possível continuar, volta-se até ao último k onde ainda existem decisões que podem ser tomadas, sem esquecer que é necessário "apagar as lâmpadas" acendidas no caminho tomado.

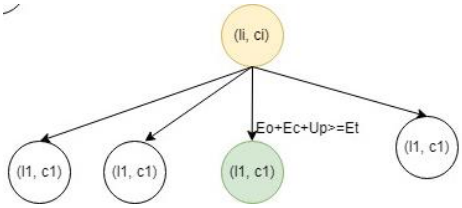

Imagem 4 – Após retornar vezes suficientes tal que se encontrem direções por explorar, estas foram percorridas, mas, neste caso, não resultaram numa solução valida. Assim, retorna-se uma vez mais, passando à direção seguinte, e apagando todas lâmpadas anteriormente acendidas.

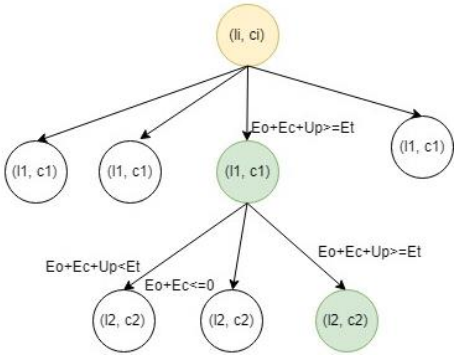

Imagem 5 – A partir do mesmo raciocínio, testam-se as direções até que uma seja válida, abrindo-se uma porta. Assim, explora-se o caminho por si proporcionado.

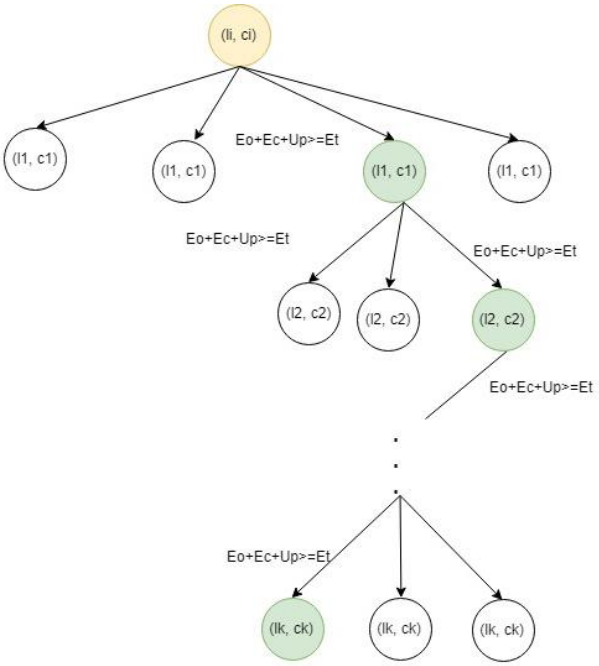

Imagem 6 - Ao chegar ao último passo, se este tiver um resultado válido não é necessário percorrer a árvore toda (isto só se aplica ao objetivo 1, porque no 2 é preciso explorar todas as decisões que prometam uma energia maior que a última encontrada).

b) Algoritmo para percorrer o losango do mapa útil

Por só interessarem as células a uma distância k da célula de partida, percebeu-se que este conjunto teria sempre o aspeto geométrico visível na Figura 7, semelhante a um losango e também referido como diamante.

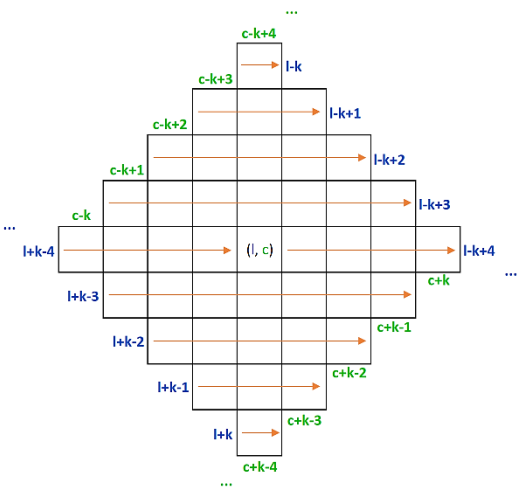

Figura 7 - Losango para o exemplo de k=4, com foco principal na evolução das coordenadas e no seu varrimento.

Foi de imediato importante estabelecer uma relação entre a largura e a altura, dependendo de k, de forma a poder percorrer as células de cima para baixo, e da esquerda para a direita, começando em (l-k, c) e acabando em (l+k, c). Dependendo da linha corrente, nomeada nl, e de k, faz-se variar a coordenada da coluna entre c-k+abs(l-nl) e c+k-abs(l-nl), identificada por nc. Se as fórmulas de início

apontarem para um valor negativo, salta-se de imediato para nl=0 ou nc=0, e assim que ultrapassarem as fronteiras superiores (L e C), essas células (não existentes) não são, nem podem ser, tidas em conta. Também são critérios de exclusão as coordenadas pertencentes à célula de partida ou a alguma célula já visitada, i.e., pertencente ao caminho, e aquelas cujos valores são negativos. Para impedir que esta célula inicial fosse contabilizada aplicou-se uma condição baseada na tabela de verdade de uma porta NAND, como está evidenciado no exemplo de código abaixo. A função onde este algoritmo é usado, função AddAll, está representada na figura 8.

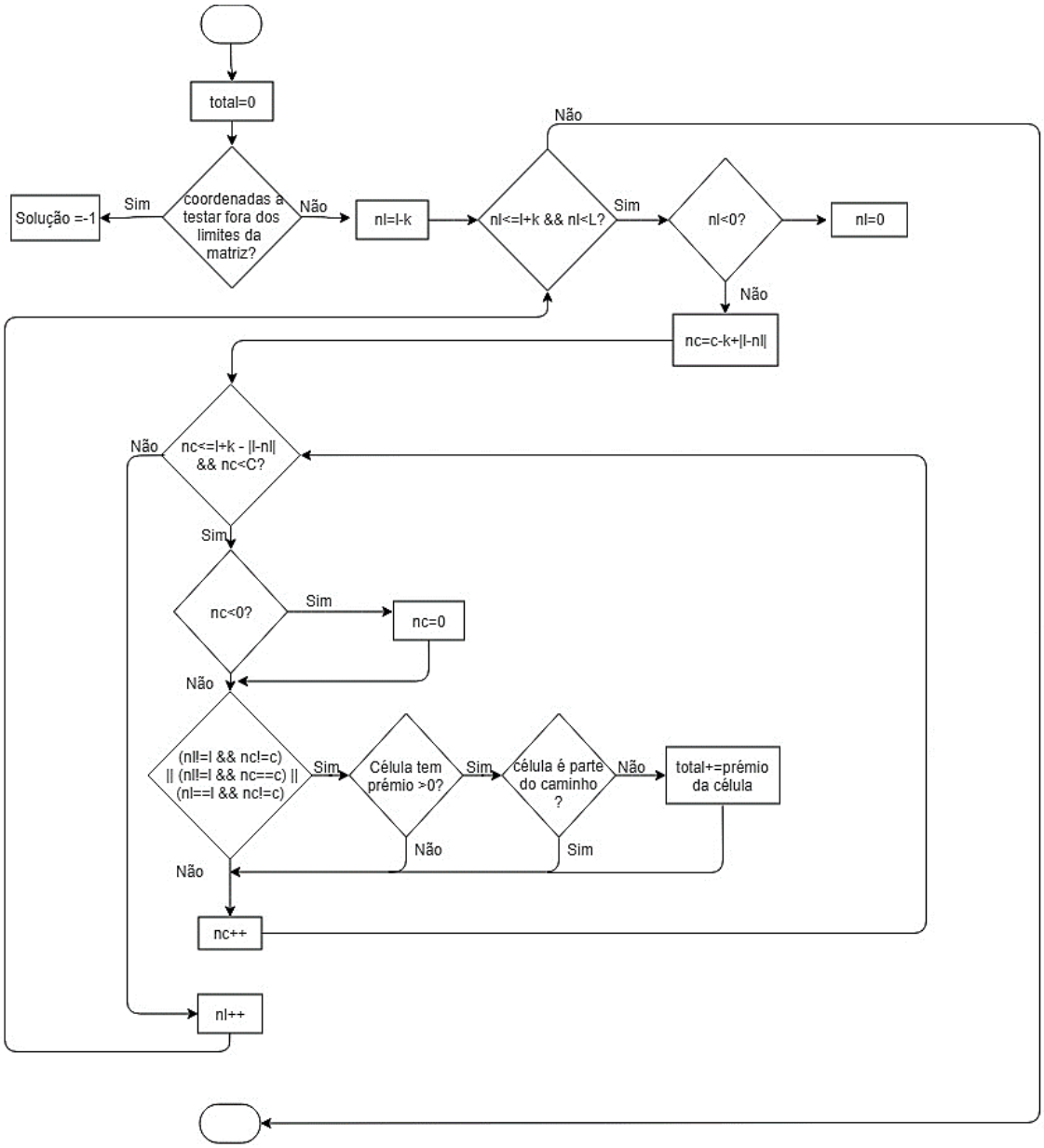

Figura 8 - Fluxograma da função AddAll do programa.

Análise dos requisitos computacionais

Na superfície, o problema é relativamente simples, no entanto é necessária alguma atenção a pequenos pormenores que poderão alterar por completo a memória usada e o tempo de execução do programa. Um destes casos é a questão de independentemente da dimensão inicial da matriz apenas ser necessário guardar em memória a secção de raio k à volta da posição inicial pois é nesta que nos vamos trabalhar. Deverá ainda ter-se em conta as condições iniciais, ou seja, se for pedido um objetivo não contemplado ou se as coordenadas iniciais se encontram fora da matriz não existe necessidade de guardar o mapa em memória e apenas basta escrever a solução no ficheiro de saída. Tudo isto contribui para que o tempo de execução e que a memória utilizada seja menor.

a) Requisitos Espaciais

Apenas duas estruturas apresentam tamanho relevante para serem contabilizadas:

a) A estrutura que guarda os dados do puzzle;
b) A estrutura que guarda o caminho;

Tendo em conta a estrutura do programa, é irrelevante o número de puzzles por ficheiro, dado que, em cada instante, apenas se encontra guardado um único problema. Assim, só o tamanho de cada puzzle é relevante para esta análise.

Como foi usado recursividade, a memória usada pela declaração de variáveis e pelo chamamento de funções não é desprezável.

Sendo N o tamanho do puzzle então a complexidade da estrutura t_matrix é $O(N^2)$ por ser uma estrutura que engloba uma matriz N*N (no pior caso será N=2k+1) e mais 1 inteiro que em comparação é desprezável. Por sua vez a estrutura t_cell tem tamanho fixo, composto por 4 inteiros. Considera-se o número de vezes que pode ser alocada e armazenada e não a estrutura em si.

a) Criação do vetor path e ref_path: a memória armazenada é proporcional ao número de passos visto que é um vetor de estruturas, cada uma composta por 4 inteiros;

b) Criação da estrutura t_matrix: a memória armazenada é proporcional às dimensões do mapa útil, uma estrutura para cada célula do mesmo.

b) Requisitos Temporais

Existem algumas funções que devemos ter em especial atenção ao fazer esta análise:

a) **AddAll**: percorre todo o losango centrado na célula inicial, por isso, no pior caso, este losango corresponde à matriz fornecida total e a respetiva complexidade será $O(N^2)$;

b) **Conditions**: a função é composta por instruções simples tirando o chamamento da função AddAll, por isso a função Conditions tem complexidade igual a $O(N^2)$

c) **MinEnergy**: é, essencialmente, composta pela evocação de outras funções e depois de si própria; É onde é aplicado DFS pelo que tem tempo de execução da ordem V^2.

d) **MaxEnergy**: independentemente de encontrar logo o caminho ou não, é necessário percorrer toda a árvore para verificar que o caminho encontrado é o melhor, logo tem requisito temporal da ordem V^2.

e) **Algoritmos dfs e branch and bound**: no pior caso todos os caminhos são viáveis pelo que poderão ter que ser todos explorados, logo é proporcional ao número de vértices/decisões, no pior caso $1 + \sum_{i=0}^{k-1} i$.

Análise Crítica

Numa primeira instância, repartimos o projeto em diferentes passos de forma a podermos encontrar um raciocínio que fosse possível implementar de forma geral e que todas as decisões de estruturas de dados, algoritmos e estratégias fossem eficientes e conciliassem memória e complexidade.

Inicialmente, ao submeter apenas passávamos 19 dos 20 testes possíveis devido a um esquecimento nosso no que dizia respeito ao agente "morrer" sem esgotar os passos todos. Contudo, este problema foi resolvido facilmente após identificar a situação concreta em que isto acontecia. Assim, conseguimos voltar a submeter e passar todos os testes.

Apesar de todos os cuidados com o gasto de memória e de tempo que tivemos em todas as fases do projeto, sabemos que existem muitas mais estratégias que poderão resultar num programa com menos tempo de execução e que fazem uso de menos memória, sendo por isso mais eficazes que a nossa.

Tendo tudo isto em consideração, acreditamos que o nosso projeto é um sucesso por termos passado a todos os testes e por ainda termos um tempo de execução relativamente baixo. O mesmo não se pode dizer da memória que gastamos, temos um pico de cerca de 63MBytes, mas como ainda se enquadra dentro do limite pedido não consideramos um valor preocupante.

Exemplo

Considere-se o ficheiro de entrada <teste.maps> abaixo representado na figura 9, que origina o ficheiro de saída <teste.paths> criado e aberto na função main.

9 9 20 7 6 5 5

	1	2	3	4	5	6	7	8	9
1	-1	-1	-1	-1	10	-1	-1	-1	-1
2	-1	-1	-1	10	-1	10	-1	-1	-1
3	-1	-1	10	-1	-1	-1	10	-1	-1
4	-1	10	-1	-1	-1	-1	-1	10	-1
5	10	-1	-1	-1	-1	-1	-1	-1	10
6	-1	10	-1	-1	-1	-1	-1	10	-1
7	-1	-1	10	-1	-1	-1	10	-1	-1
8	-1	-1	-1	10	-1	10	-1	-1	-1
9	-1	-1	-1	-1	10	-1	-1	-1	-1

	0	1	2	3	4	5	6	7	8
0	-1	-1	-1	10	-1	10	-1	-1	-1
1	-1	-1	10	-1	-1	-1	10	-1	-1
2	-1	10	-1	-1	-1	-1	-1	10	-1
3	10	-1	-1	-1	-1	-1	-1	-1	10
4	-1	10	-1	-1	-1	-1	-1	10	-1
5	-1	-1	10	-1	-1	-1	10	-1	-1
6	-1	-1	-1	10	-1	10	-1	-1	-1
7	-1	-1	-1	-1	10	-1	-1	-1	-1

9 9 -2 7 6 5 5

	1	2	3	4	5	6	7	8	9
1	-1	-1	-1	-1	10	-1	-1	-1	-1
2	-1	-1	-1	10	-1	10	-1	-1	-1
3	-1	-1	10	-1	-1	-1	10	-1	-1
4	-1	10	-1	-1	-1	-1	-1	10	-1
5	10	-1	-1	-1	-1	-1	-1	-1	10
6	-1	10	-1	-1	-1	-1	-1	10	-1
7	-1	-1	10	-1	-1	-1	10	-1	-1
8	-1	-1	-1	10	-1	10	-1	-1	-1
9	-1	-1	-1	-1	10	-1	-1	-1	-1

	0	1	2	3	4	5	6	7	8
0	-1	-1	-1	10	-1	10	-1	-1	-1
1	-1	-1	10	-1	-1	-1	10	-1	-1
2	-1	10	-1	-1	-1	-1	-1	10	-1
3	10	-1	-1	-1	-1	-1	-1	-1	10
4	-1	10	-1	-1	-1	-1	-1	10	-1
5	-1	-1	10	-1	-1	-1	10	-1	-1
6	-1	-1	-1	10	-1	10	-1	-1	-1
7	-1	-1	-1	-1	10	-1	-1	-1	-1

(a) Ficheiro de entrada (b) matrizes alocadas no programa

Figura 9 - Ficheiro de entrada e respetivas matrizes alocadas no programa

Como o puzzle passa na verificação dos parâmetros inicias (L, C, objetivo, l, c, k e Energia inicial), faz-se as contas para alocar a matriz útil:

l-k=7-5=2 >0 logo l_p=l-k-1=7-5-1=1

l+k=7+5=12>L=9 logo lp=9-1=8

c-l=1>0 logo c_p=L-1=8

c+k=6+5=11>9 logo cp=C-1=8

Ll=lp-l_p+1=8-1+1=8

Cl=cp-c_p+1=8-0+1=9

Já tendo as dimensões da matriz e os limites da mesma, aloca-se a memória para a estrutura do mapa (LlxCl) seguido do preenchimento do mesmo, ver figura 9b . Passa-se agora à conversão das coordenadas inicias:

como Ll=8<L=9 ll=k=5 e cl=5

como c_p=0, cl=Cl-k-1=9-5-1=3 mas Cl=9>=Cl=9 então cl=6-1=5

Com as coordenadas iniciais no sistema de coordenadas do ficheiro, passa-se à resolução do problema em concreto:

Como k difere de zero, existem passos a serem dados. Cria-se o vetor de dimensão fixa k nomeado path e marca-se a célula inicial como parte do caminho (i.e., visited=1). A partir daqui passa-se à procura em profundidade representada na figura 10.

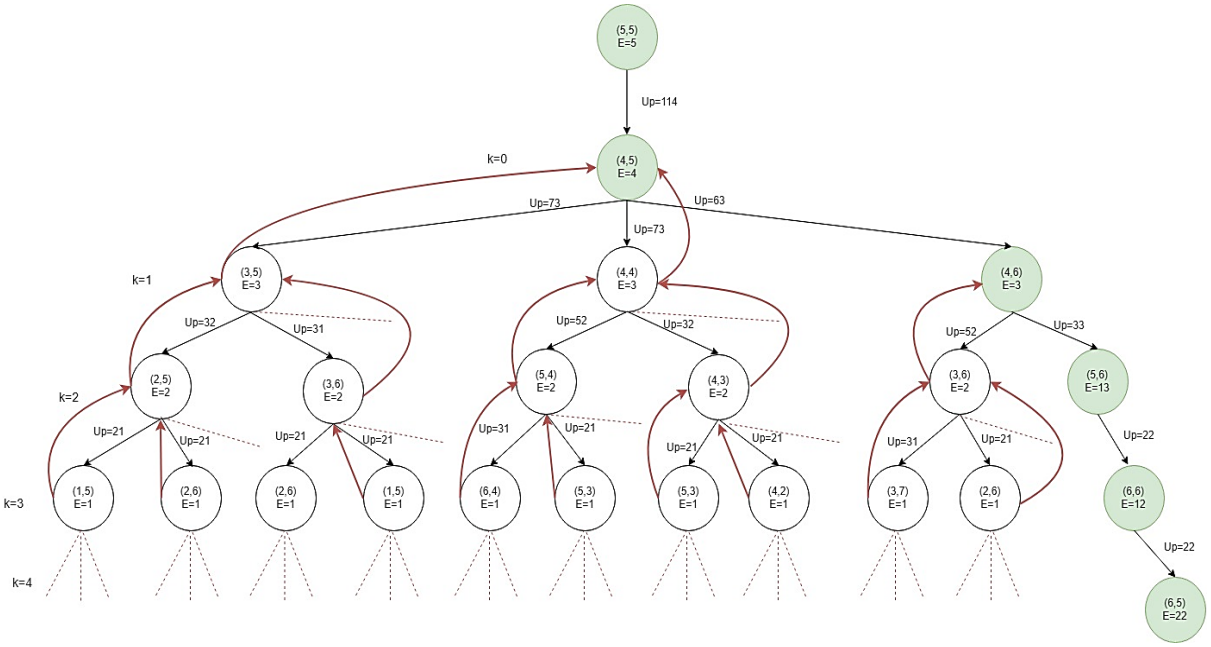

Figura 10 -Progressão do programa para o exemplo de problema de objetivo 1.

Assim, obtém-se o caminho indicado na tabela 2 que deverá, ainda, sofrer uma conversão de coordenadas para obtermos o caminho no sistema de coordenadas do ficheiro de entrada, também apresentado na tabela 2.

Caminho após procura		$l_i=l'_i$+l_p+1	$c_i=c'_i$+c_p+1	Caminho após conversão	
(4,5)	-1	$\boldsymbol{l_0}$=4+1+1=6	$\boldsymbol{c_0}$=5+0+1=6	**(6,6)**	-1
(4,6)	-1	$\boldsymbol{l_1}$=4+1+1=6	$\boldsymbol{c_1}$=6+0+1=7	**(6,7)**	-1
(5,6)	10	$\boldsymbol{l_2}$=5+1+1=7	$\boldsymbol{c_2}$=6+0+1=7	**(7,7)**	10
(6,6)	-1	$\boldsymbol{l_3}$=6+1+1=8	$\boldsymbol{c_3}$=6+0+1=7	**(8,7)**	-1
(6,5)	10	$\boldsymbol{l_4}$=6+1+1=8	$\boldsymbol{c_4}$=5+0+1=6	**(8,6)**	10

Tabela 2 - Caminho antes e depois da conversão para o sistema de coordenadas do ficheiro de entrada.

Assim, procede-se à escrita do caminho no sistema de coordenadas correto no ficheiro de saída no seguinte formato:

9 9 20 7 6 5 5 22

6 6 -1

6 7 -1

7 7 10

8 7 -1

8 6 10

Para finalizar, liberta-se a memória alocada para o vetor path e para a estrutura que guarda o puzzle.

Como o ficheiro ainda contém mais um puzzle, o programa lê a 1ª linha

9 9 -2 7 6 5 5

e procede-se às devidas verificações e cálculos para alocar a matriz útil, ver figura 9 b) , e, por fim, preenchê-la. Já que as dimensões desta matriz e as coordenadas iniciais do problema são iguais às do puzzle anterior, suprime-se, aqui, a repetição da escrita destes cálculos e refere-se apenas aos mesmos no início desta secção.

Sabendo que k é diferente de 0, aloca-se e inicializa-se o vetor principal path de dimensão k e aciona-se a flag visited da célula inicial. Como o objetivo é -2, é necessário criar-se e inicializar-se um segundo vetor, que servirá como um auxiliar, de dimensão k+1 em que a primeira dimensão guarda apenas a energia total do caminho guardado e as restantes k posições guardam o caminho.

A partir do mapa útil representado na figura 7, a procura do caminho com energia máxima começa por chamar a função MinEnergy com energia de target igual a 1, uma vez que esta é a primeira procura e, por isso, o vetor de referência ref_path encontra-se vazio. Esta procura está representada na figura 11.

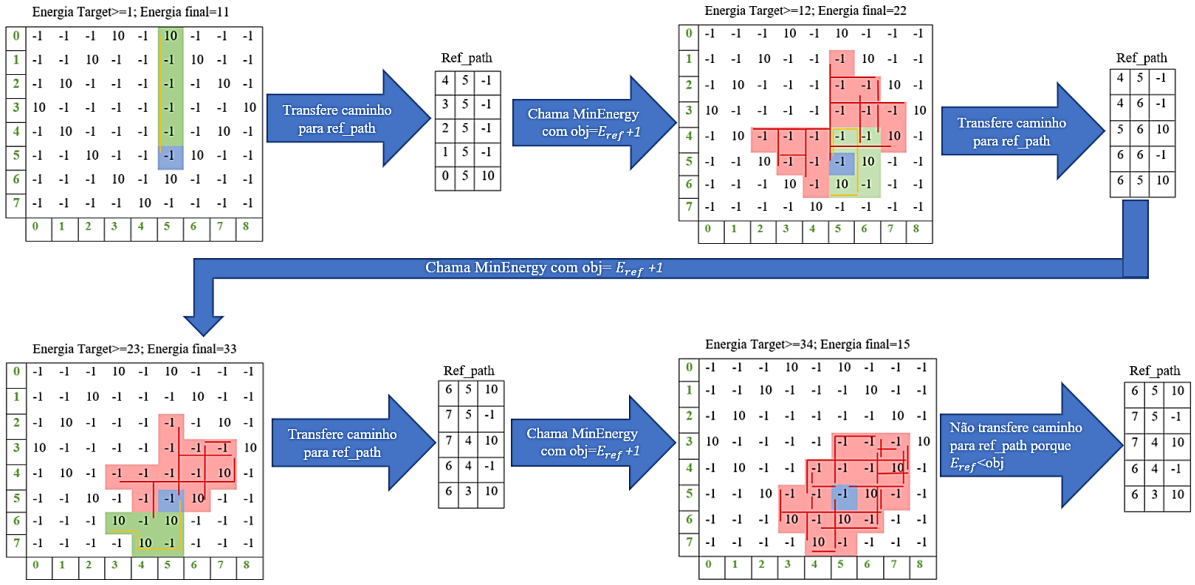

Figura 11 - Progressão do programa para o exemplo de problema de objetivo 2.

Após obter o caminho da energia máxima é necessário converter o mesmo para o sistema de coordenadas do ficheiro de saída e proceder escrita do caminho no sistema de coordenadas correto no ficheiro de saída no seguinte formato, deixando uma linha em branco entre as soluções dos puzzles:

9 9 -2 7 6 5 5 33

6 5 10

7 5 -1

7 4 10

6 4 -1

6 3 10

Por fim e como o objetivo do puzzle é -2, liberta-se a memória alocada para os dois vetores e para a estrutura t_matrix que guarda o puzzle.

Como não existem mais puzzles no ficheiro, o programa sai do loop para a leitura do ficheiro, liberta a memória alocada para a criação do nome do ficheiro de saída e fecha ambos os ficheiros.

Bibliografia

[1] AED (IST/DEEC) Slides das Aulas Teóricas 2016.

[2] Na realização deste projeto foram, ainda, consultados os seguintes websites:

a. http://stackoverflow.com/