
Universidade Federal
de São João del-Rei

# TRABALHO PRÁTICO 2

**Lucas Gonçalves Nojiri**
**Arthur Antunes Santos Silva**

Este trabalho prático tem por objetivo exercitar conceitos e práticas de matrizes e células no grid. Baseado no código em C para a disciplina de AEDS3 do curso de Ciência da Computação da Universidade Federal de São João del Rei.

**São João del Rei**

**Maio de 2023**

# Sumário

# 1 Introdução

Problema: Cálculo de energia de Harry Potter ao atravessar um grid para chegar ao artefato.

Descrição do problema: O objetivo deste problema é determinar a energia que Harry Potter terá ao atravessar um grid. O grid é representado por uma matriz S de tamanho R x C, onde cada célula contém um número inteiro que representa o efeito que aquela célula terá em Harry Potter.

Se S[i][j] for um valor negativo, isso indica a presença de um monstro na célula (i, j), e Harry perderá S[i][j] unidades de energia ao atravessá-la. Se S[i][j] for um valor positivo, isso indica a presença de uma poção na célula (i, j), e Harry ganhará S[i][j] unidades de energia ao atravessá-la.

Entrada: O programa receberá um arquivo de entrada contendo as informações sobre o grid e seus artefatos. O arquivo terá o seguinte formato:

Os valores no arquivo de entrada representam o grid S com R linhas C colunas que contém a localização de um artefato, onde cada célula contém um número inteiro S[i][j] que representa o efeito que aquela célula terá no Harry Potter. Se S[i][j] for negativo, isso significa que há um monstro naquela célula e Harry perderá S[i][j] de energia ao passar por ela. Se S[i][j] for positivo, isso significa que há uma poção naquela célula e Harry ganhará S[i][j] de energia ao passar por ela. No exemplo de entrada, o primeiro caso de teste tem um grid 2x3 com uma célula de inimigo na posição (1,3) que irá retirar 3 de energia do herói e uma célula de poção na posição (1,2) que irá adicionar 1 de energia. O segundo caso de teste tem um grid 2x2 com uma célula de poção na posição (1,2) e uma célula de poção na posição (2,1). O terceiro caso de teste tem um grid 3x4 com uma célula de monstro na posição (1,2) que irá retirar 2 de energia de Harry, uma célula de inimigo na posição (1,3) que irá retirar 3 de energia do Harry e uma célula de poção na posição (1,4) que irá adicionar 1 de energia para o Harry.

Saída: O programa deverá calcular e exibir a energia total que Harry Potter terá ao atravessar o grid.

Foi proposto como objetivo do trabalho encontrar duas estratégias para solucionar este problema.

## 1.1 Requisitos

As implementações devem ser feitas na linguagem C (C++ pode ser usado para o tratamento de strings), usando a biblioteca padrão da linguagem.

O arquivo de entrada deve seguir os seguintes requisitos:

- Grid = S.

- A linha 1 é um inteiro T, indica o número casos de testes.

- R = linhas C = colunas. A segunda linha mostra ao tamanho do grid.

- Cada teste consiste em R e C, na primeira linha seguido pela descrição do grid em R linhas, cada uma contendo C inteiros.

- As linhas são numeradas de 1 a R de cima para baixo e as colunas são numeradas de 1 a C da esquerda para a direita.

- Células com S[i][j] $<$ 0 contêm monstros, outras contêm poções mágicas. Célula negativa = mostro. Célula positiva = poção.

Por exemplo, para o primeiro caso de teste do problema apresentado, temos que T=3, o primeiro grid S possui R=2 linhas e C=3 colunas, e sua descrição é:

$$\begin{bmatrix} 0 & 1 & -3 \\ 1 & -2 & 0 \end{bmatrix}$$

Ou seja, o grid possui 2 linhas e 3 colunas, e os valores de energia nas células são:

(1,1) = 0 (1,2) = 1 (1,3) = -3. Esses são os valores das energias em cada coordenada.

(2,1) = 1 (2,2) = -2 (2,3) = 0

Dessa forma o arquivo de entrada.txt deve seguir este modelo:

3 -> Número de testes

2 3 -> Teste 1

0 1 -3 -> Cada valor é o valor que esta no nó

1 -2 0

2 2 -> Teste 2

0 1

2 0

3 4 -> Teste 3

0 -2 -3 1

-1 4 0 -2

1 -2 -3 0

Na sequência, há R linhas contendo C inteiros cada, descrevendo a matriz do jogo. Cada célula contém um inteiro S[i][j], indicando se a célula contém um monstro (quando S[i][j] $<$ 0) ou uma poção mágica (quando S[i][j] $>=$ 0). As linhas da matriz são numeradas de 1 a R, de cima para baixo, e as colunas são numeradas de 1 a C, da esquerda para a direita.

# 2 Mapeamento do código

### 2.0.1 grafo.h

```
1    define MAX 100
2    typedef struct no {
3        int i, j, valor;
4    }No;
```

### 2.0.2 grafo.c

1. **int File(int argc, char **argv, char *input, char *output)**

   Esta função lê os argumentos da linha de comando (argc e argv) e armazena os nomes dos arquivos de entrada e saída em variáveis (input e output, respectivamente). Ela retorna 0 se a leitura for bem-sucedida e -1 caso contrário.

2. **int Leitura_input(char *input, int *teste, int lin[], int col[], int grid[][MAX][MAX])**

   Abre o arquivo de entrada especificado e lê as informações dos testes. Ela armazena o número de testes em teste, as dimensões das grades em lin e col, e os valores das grades em grid. A função retorna 1 se a leitura for bem-sucedida e 0 caso contrário.

3. **void Nos(int teste, int lin[], int col[], int grid[][MAX][MAX], No nos[][MAX])**

   Essa função preenche a matriz de estruturas nos com os dados dos grids. Cada elemento da matriz nos representa um nó na grade e armazena suas coordenadas (i, j) e valor correspondente.

4. **int tp2(int argc, char **argv)**

   É a função principal que realiza o processamento dos testes. Ela chama as funções anteriores para ler o arquivo de entrada, preencher a matriz nos e executar a estratégia selecionada nos testes. A função lê o valor da estratégia a ser executada e armazena na variável estrategia. Se estrategia for igual a 1, chama a função verifica para obter o valor mínimo inicial do caminho e grava o resultado no arquivo de saída. Se estrategia for igual a 2, ele itera sobre os testes e chama a função Energia_minima para encontrar o valor mínimo inicial de energia em cada grade. Em seguida, grava o resultado no arquivo de saída. Após o processamento de todos os testes, o arquivo de saída é fechado.

### 2.0.3 estrategia1.h

```
1    #include "grafo.h"
2    int* maiorCaminho(int lin, int col, int grid[][MAX]);
3    int verifica(int* array);
```

### 2.0.4 estrategia2.h

```
1  #include "grafo.h"
2  int min(int a, int b);
3  int Energia_minima(int lin, int col, int grid[][MAX]);
```

### 2.0.5 estrategia1.c

1. **int* maiorCaminho(int lin, int col, int grid[][MAX])**

   Esta função encontra o caminho com a maior soma de elementos em uma grade dada. Ela começa no canto superior esquerdo da grade e, de forma iterativa, se move para baixo ou para a direita para chegar ao canto inferior direito. A função aloca dinamicamente um array para armazenar os elementos do caminho e retorna o ponteiro para esse array.

2. **int verifica(int* array)**

   Esta função verifica a validade de um caminho representado por um array de inteiros. Ela recebe um array como entrada, onde cada elemento representa o valor de um passo no caminho. A função percorre o array e calcula uma soma acumulada x somando cada elemento a x. Se a soma acumulada x se tornar menor que 1 em algum momento, ela ajusta o valor de x para garantir que ele seja sempre maior ou igual a 1. A função retorna o valor inicial ajustado de x, que representa o ponto de partida ideal para o caminho.

### 2.0.6 estrategia2.c

O problema foi resolvido pelo método da programação dinâmica, sendo assim possível calcular o valor necessário em cada posição do grid.

1. **int Energia_minima(int lin, int col, int grid[][MAX])**

   Essa função implementa a estratégia 2 para encontrar a energia mínima necessária para percorrer um caminho da posição (0,0) até a posição (lin-1, col-1) nos grids. O valor da última posição (lin-1, col-1) da matriz ultima_pos é definido como 1, pois não é necessário energia adicional para estar nessa posição. Em seguida, preenche-se os valores da última coluna e linha da matriz ultima_pos.

   Começando da penúltima coluna e linha e descendo até a primeira posição (0, 0), o valor mínimo necessário para alcançar cada posição é calculado subtraindo-se o valor da posição atual do grid. Se o valor calculado for menor ou igual a 0, ele é ajustado para 1, pois é necessário pelo menos 1 de energia. Em seguida, preenche-se os valores das outras posições da matriz ultima_pos. Começando da penúltima linha (lin-2) e indo até a primeira linha (0), e para cada linha, começando da penúltima coluna e indo até a primeira coluna (0), o valor mínimo necessário para alcançar cada posição é calculado.

   Ele é o menor valor entre o valor da posição abaixo e o valor da posição à direita, subtraído do valor da posição atual do grid. Se o valor calculado for menor ou igual a 0, ele é ajustado para 1. Finalmente o valor necessário para a posição (0,0) da matriz ultima_pos é retornado como a energia mínima necessária para percorrer o caminho desejado na grade.

2. **int min(int a, int b)**

   A função min compara dois valores a e b e retorna o menor valor entre eles usando o operador ternário.

### 2.0.7 main.c

Esta secção é responsável pelo funcionamento do programa sendo a função principal do sistema, ela chama a função tp2 que então realiza todas as operações dos testes. Por fim, a main imprime o uso de memória e o tempo de execução, que são medidos pelo gettusage e gettimeofday.

# 3 Complexidade

## 3.1 Complexidade das funções

### 3.1.1 grafo.h

Este código é apenas um cabeçalho em C que define algumas estruturas de dados e declarações de função. Sobre a complexidade, o código não possui nenhum algoritmo ou lógica implementada. Sendo uma coleção de bibliotecas e definindo os tipos das funções.

### 3.1.2 grafo.c

1. **int File(int argc, char **argv, char *input, char *output)**

   O código possui complexidade O(n), n sendo o número de argumentos, existe apenas um loop while que percorre a linha de comando apenas uma vez, assim podemos dizer que sua complexidade é O(1), caso não for inserido nenhum argumento a complexidade também será O(1), desta forma o melhor caso teria a complexidade de O(2).

2. **int Leitura_input()**

   O código possui complexidade O(n), n sendo o número de pontos lidos no arquivo de entrada, pois o código lê cada ponto do arquivo em um loop while e armazena os pontos em um array alocado dinamicamente, o restante são entradas mais simples com complexidade O(1).

3. **void Nos()**

   O código possui complexidade O(n * m * p) ,esta função preenche uma matriz com os dados dos nós. A complexidade dessa função depende do tamanho dos dados.

4. **int tp2(int argc, char **argv)**

   A complexidade da função tp2 possui uma complexidade de O(L * C + aux) no caso da estratégia 1 e O(L * C) no caso da estratégia 2.

   Esta função é a função principal do programa. Ela chama as funções anteriores e realiza operações adicionais. A complexidade dessa função depende das chamadas de função realizadas dentro dos loops. Na estratégia 1, há um loop que executa a função maiorCaminho e em seguida uma função verifica. Na estratégia 1, o loop chama a função maiorCaminho e a função verifica. A complexidade da função maiorCaminho depende dos valores de lin e col, podendo ser O(L * C). A função verifica, descobre o array resultante e sua complexidade é linear em relação ao tamanho do array, que é O(aux). Portanto, a complexidade do loop da estratégia 1 é O(L * C + aux). Na estratégia 2, o loop chama a função Energia_minima. A complexidade dessa função depende dos valores de lin e col, resultando em O(L * C).

### 3.1.3 estrategia1.h

Este código é um arquivo de cabeçalho em C que define as duas funções: maiorCaminho e verifica.

1. **int* maiorCaminho(int lin, int col, int grid[][MAX])**

   A função maiorCaminho possui um loop while que itera até atingir a última posição da matriz. A complexidade desse loop depende do tamanho da matriz, ou seja, O(L * C), onde lin é o número de linhas e col é o número de colunas da matriz.

2. **int verifica(int* array)**

   A função verifica possui um loop for sobre o array de entrada. A complexidade desse loop esta relacionado ao tamanho do array, representado por aux. Assim a complexidade dessa função é O(aux).

### 3.1.4 estrategia2.h

1. **int min(int a, int b)**

   A função min realiza uma comparação entre dois valores a e b e retorna o menor deles. Essa função possui uma complexidade constante, sendo O(1).

2. **int Energia_minima(int lin, int col, int grid[][MAX])**

   A complexidade dessa função é determinada pelos loops for que preenchem os valores da matriz ultima_pos é O(L*C) A função Energia_minima é responsável por calcular o valor mínimo necessário para percorrer um caminho em uma matriz grid do ponto (0,0) até o ponto (lin-1, col-1) sendo o último elemento.

### 3.1.5 estrategia1.c

1. **void Prog_Dinamica(int lin, int col, int grid[][MAX])**

   A complexidade da função Prog_Dinamica é O(L*C).

   Onde L representa o número de linhas e C representa o número de colunas do grid passado como argumento, possuindo 2 loops "for"aninhados que percorrem o grid de trás pra frente, ou seja, da última posição até a primeira, dentro deles existem operadores de atribuição e comparação, tendo uma complexidade constante de O(1).

### 3.1.6 estrategia2.c

1. **int Energia_minima(int lin, int col, int grid[][MAX])**

   A complexidade dessa função é O(L * C). Existem 3 loops nesta função.

   O 1°para preencher os valores da última coluna: Um loop de lin-2 até 0. Com uma complexidade de O(L).

   O 2° Loop preenche os valores da última linha: Um loop de col-2 até 0. Portanto, tem uma complexidade de O(C).

   O 3° Loop para preencher os valores das outras posições: Com o loop de lin-2 até 0 no primeiro for, e de col-2 até 0 no segundo for. Tendo uma complexidade de O(L * C).

### 3.1.7 main.c

```
1    int main(int argc, char **argv) {
2    struct rusage buff;
```

A complexidade do código pode ser comparada com a complexidade das funções, que são dependentes em relação ao tamanho do arquivo, ao número de testes e ao tamanho das matrizes (L e C). Função File: Essa função possui um loop que lê os argumentos de linha de comando e realiza operações de cópia de strings. Sua complexidade é em relação ao tamanho dos argumentos de linha de comando, que é dado por argc. Sendo complexidade de O(argc).

Função Leitura_input: Essa função lê o conteúdo de um arquivo e armazena em array. Seu tempo de execução depende do tamanho do arquivo e das dimensões dos array, mas considerando apenas o tamanho do arquivo, sua complexidade é em relação ao tamanho do arquivo.

Função Nos: Essa função preenche uma matriz de estruturas com base nos array lidos anteriormente. Sua complexidade depende do tamanho das matrizes e do número de estruturas a serem preenchidas. Sua complexidade é O(teste * L * C), onde teste é o número de testes e lin e col são as dimensões das matrizes.

Determinar a estratégia a ser utilizada e chamar a função correspondente (maiorCaminho ou Energia_minima). Portanto, a complexidade dessas operações é O(teste) multiplicada pela complexidade de cada uma das funções.

Abrir e fechar o arquivo de saída: Essas operações têm uma complexidade constante e não impactam significativamente a complexidade total do código.

# 4 Testes

## 4.1 Performance e Resultados

A busca em largura não foi implementada nesse código, porém foi testada como uma das estratégias que seriam utilizadas mas no final foi descartada, percorrendo um labirinto representado por uma matriz bidimensional. Começando na posição inicial (0, 0) e utilizando uma fila, ela visita os nós vizinhos na ordem em que foram descobertos. O objetivo é encontrar o caminho que leva à posição final (lin - 1, col - 1) com o maior valor acumulado ao longo do percurso. O algoritmo mantém uma variável para armazenar o valor máximo encontrado. Ao final da busca, o valor máximo é retornado como resultado. O código principal lê os dados do labirinto de um arquivo e chama a função de busca para cada labirinto, exibindo o valor mínimo encontrado para cada um.

### 4.1.1 Estatégia 1


(a) Teste 1 (b) Teste 2

Figura 1

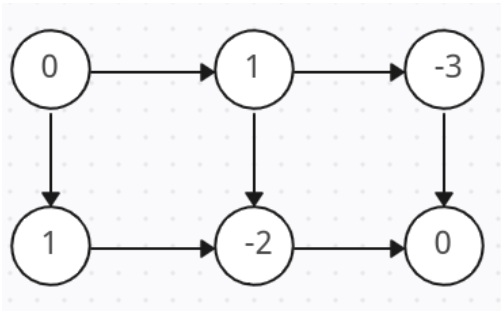
### 4.1.2 Estatégia 1


(a) Teste 1 (b) Teste 2

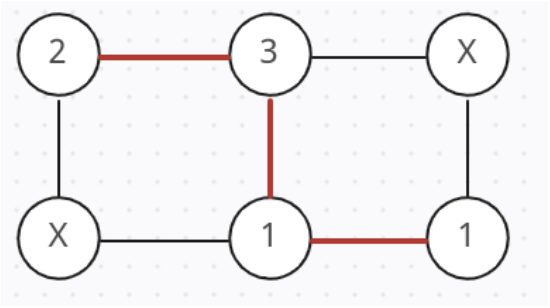
Figura 2

A partir das análises de desempenho das duas estratégias utilizadas, pode-se dizer que o gasto de memória foi o mesmo, mas ainda sim em termos de velocidade de execução o algoritmo de Programação Dinâmica se saiu melhor em comparação ao Guloso. Dessa forma, ambos os algoritmos são bem sucedidos em solucionar o problema, mas a estratégia 2 sai em vantagem, resolvendo mais rápido com o mesmo gasto de memória.

### 4.1.3 Grid 1 e a o melhor caminho



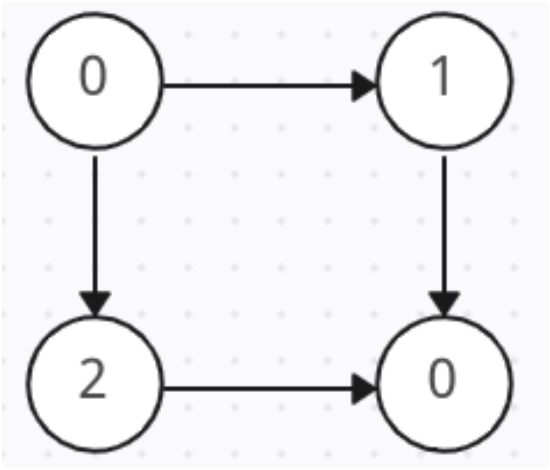

(a) Grid 1



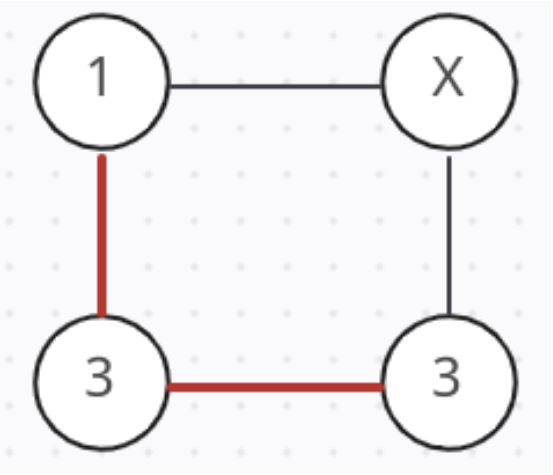

(b) Mínimo

Figura 3

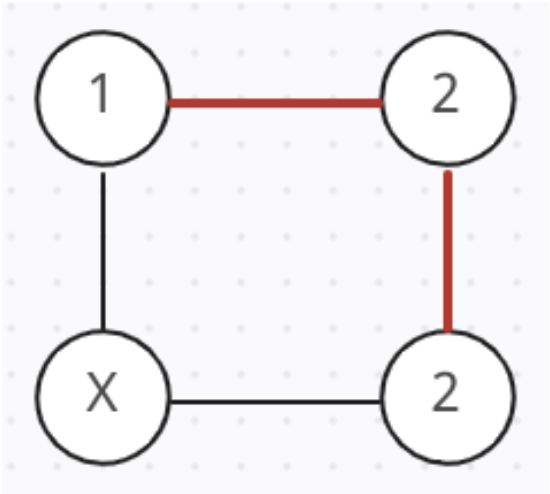
### 4.1.4 Grid 2 e a o melhor caminho




(a) grid 2



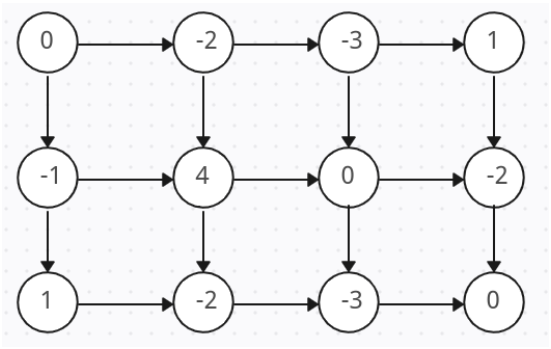

(b) Caminho 1



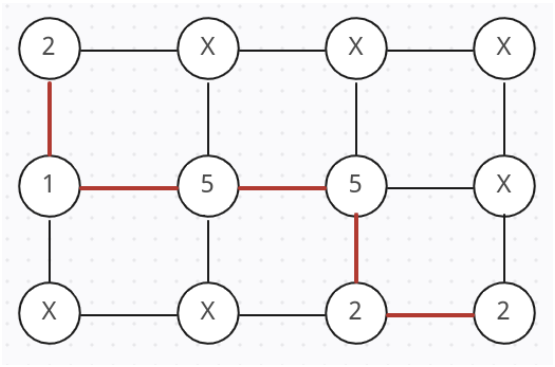

(c) Caminho 2

Figura 4

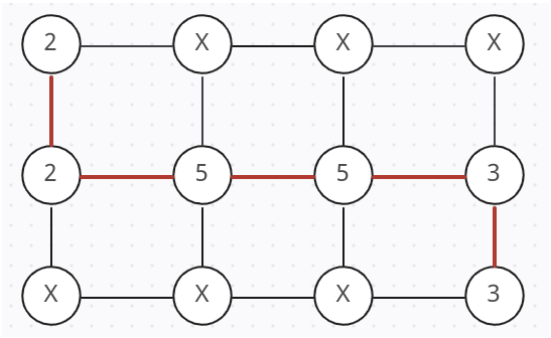
### 4.1.5 Grid 3 e a o melhor caminho




(a) grid 3




(b) Caminho 1




(c) Caminho 2

Figura 5

# 5 Conclusão

Neste problema, o desafio foi ajudar o famoso mago Harry Potter a vencer um jogo em que ele precisa encontrar um artefato poderoso em um grid mágico. O objetivo é levar Harry até o artefato sem deixar sua energia chegar a zero, pois caso isto ocorra Harry irá falhar na missão, evitando os monstros que drenam energia e aproveitando as poções enquanto percorre o grid aumentam a sua energia.

Para solucionar o problema, foram implementadas duas estratégias que calculam a energia mínima necessária para Harry percorrer o grid até a célula final sem ter sua energia esgotada. A primeira estratégia foi programação dinâmica, a função implementa a abordagem de programação dinâmica para calcular o valor mínimo necessário para percorrer da posição inicial até a posição final no grid. Ela recebe como entrada o número de linhas e colunas e a matriz do grid. O algoritmo guloso segue uma estratégia de caminhar para a célula adjacente que contém o maior valor em cada etapa, seja para baixo ou para a direita. Dessa forma, o algoritmo sempre seleciona o próximo elemento com base apenas nos valores atuais e não considera o impacto final. As estratégias envolvem encontrar o caminho ótimo pelo grid, considerando as células com poções e monstros, e calcular a energia mínima necessária para que ele termine com o mínimo de energia necessária.

O programa lê um arquivo de entrada que contém o número de casos de teste, seguido pelas dimensões do grid e a descrição das células, indicando se são poções ou monstros. O programa produz um arquivo de saída com a energia mínima necessária para cada caso de teste, partindo da célula inicial até a célula final.

Além disso, para avaliar o desempenho do programa, foram utilizadas as funções getrusage e gettimeofday para medir os tempos de usuário e sistema.

Durante o processo de codificação do algoritmo as maiores dificuldades encontradas foram as implementações das estratégias. Foram testados vários métodos, entre eles o Djikstra, que causou muita confusão, e o algoritmo não obtia resultados corretos. Por este motivo foi preferido usar os método de Programação dinâmica e algoritmo guloso.

Concluindo, pode-se dizer que a realização e os testes deste trabalho foram bem sucedidas, apesar das dificuldades em percorrer o grid e de aplicar os algoritmos para descobrir a solução. Além disso foi possível observar diferentes campos em que sem podem aplicar as mesmas técnicas desenvolvidas neste trabalho, sendo elas o clássico problema do Mundo de Wumpus por exemplo, onde você precisa encontrar e eliminar um monstro chamado Wumpus em uma caverna. Você se move entre as salas da caverna e usa pistas para descobrir a localização do Wumpus. Tenha cuidado para não cair em poços ou ficar sem flechas. Por este motivo problemas como este podem ser resolvidos com os algoritmos aplicados neste trabalho.

# 6 Referências

https://www.ime.usp.br/~leliane/IAcurso2000/Wumpus.html

https://updatedcode.wordpress.com/2015/06/27/busca-em-grafos-c/

http://wiki.icmc.usp.br/images/c/c7/2014_-_03_-_Travessias_-_Rosane.pdf