**Universidade Federal de São João del Rei - UFSJ**
**Departamento de Ciência da Computação - DCOMP**
**Disciplina de Algoritmos e Estrutura de Dados - AEDS III**

# Aplicação e análise de algoritmos para caminhamento em grafo.

**Daniel Marques e Rian Wagner.**

Maio, 2023.

## 1. Introdução:

O caminhamento de grafos é um conceito fundamental em teoria dos grafos e possui diversas aplicações em ciência da computação e outros campos. É uma técnica que permite explorar e percorrer os vértices e arestas de um grafo de forma sistemática [1]. A natureza versátil dessa técnica permite que ela seja aplicada em uma ampla variedade de problemas e domínios, incluindo redes de computadores, otimização, jogos, biologia computacional, análise de redes sociais, entre outros.

O Problema proposto apresenta um desafio que envolve a exploração de um grid mágico (que pode ser modelado como grafos) com poções e monstros. O objetivo é levar Harry Potter até o artefato sem deixar sua energia cair para zero ou menos.

O objetivo é determinar a energia mínima com a qual Harry deve começar na célula (1, 1) para que ele mantenha uma energia positiva ao longo de sua jornada até a célula (R, C). Diferentes abordagens podem ser utilizadas para resolver esse problema, dependendo das restrições e do desempenho desejado.

No primeiro momento, podemos comparar o problema com o problema do “Mundo de Wumpus”, proposto em 1972, por Gregory Yob, e que pode-se obter uma solução usando o caminho de grafo. No entanto, as duas situações apresentam problemas distintos e mecânicas diferentes.

Para o problema proposto na disciplina, elaboramos duas estratégias diferentes, como é pedido nas especificações. As estratégias escolhidas foram um algoritmo que utiliza programação dinâmica e uma heurística gulosa. O algoritmo de programação dinâmica é o responsável por dar as saídas corretas, melhor caso. Já a heurística gulosa, obtivemos alguns resultados ótimos e outros próximos do melhor caso, assim como esperado, uma vez que algoritmos gulosos podem retornar soluções aproximadas [2].

## 2. Formato de entrada e saída:

A entrada é feita com um arquivo *entrada.txt*. A primeira linha do arquivo deve conter o número de casos teste (T) e para cada caso de teste temos:

- A primeira linha contém dois números inteiros separados por espaço: R e C, representando o número de linhas e colunas do grid, respectivamente.
- As próximas R linhas descrevem o conteúdo do grid, onde cada linha contém C inteiros separados por espaço. Esses inteiros representam o valor de cada célula do grid. Valores negativos indicam a presença de monstros, enquanto valores não negativos representam poções mágicas.

A saída é dada também por um arquivo *saida.txt* que exibe na tela o resultado de cada caso de teste.

Para gerar mais casos de testes e registrá-los para a obtenção dos resultados, elaboramos um gerador de entradas de acordo com as especificações de entrada do trabalho.

Exemplo de entrada e saída:

*entrada.txt*

---

1
2 2
0 1
2 0

---

*saida.txt*

---

*1*

---

**3. Modelagem e principais funções:**

O problema do Grid de Harry Potter envolve um grid bidimensional representado por uma matriz S de tamanho RxC. Cada célula do grid contém um valor inteiro que representa a energia daquela posição. Essa energia pode ser positiva, representando uma poção que adiciona energia ao mago, ou negativa, representando um monstro que retira energia do mago.

Esse problema pode ser modelado como um problema de grafos, em que cada posição (i,j) no grid é tratada como um vértice do grafo. As conexões entre as posições vizinhas são consideradas como arestas do grafo. Além

disso, as arestas são ponderadas pelos valores das células correspondentes, refletindo o impacto daquela posição na energia do mago.

### 3.1 Modelagem algoritmo que utiliza programação dinâmica:

Objetivo é percorrer todo grid da célula *(0,0)* até a célula *(R-1, C-1)* e determinar a energia mínima necessária. Para encontrar a energia mínima necessária, utilizamos uma matriz *"pd"* com as mesmas dimensões do grid. Essa matriz é a matriz de programação dinâmica, cada célula *pd(i,j)* armazena a energia mínima necessária para chegar ao vértice *(i,j)*. O cálculo da energia mínima é realizado de forma iterativa, percorrendo a matriz "pd" de baixo para cima e da direita para a esquerda. Essa direção de percurso é bem conhecida para resolução de problemas com programação dinâmica, conhecida como *"bottom-up"*. Essa abordagem é baseada no princípio da otimalidade de Bellman, que afirma que uma solução ótima global pode ser construída a partir de soluções ótimas de subproblemas [3].

A energia mínima é calculada comparando as energias dos vértices adjacentes abaixo e à direita. A menor energia entre esses vértices é escolhida e subtraída do valor da célula *(i,j)* na matriz *S*. Caso o resultado dê maior que zero, ele é atribuído à célula *pd(i.j)* Caso contrário, é atribuído o valor 1, indicando que pelo menos 1 de energia é necessária para percorrer aquele vértice.

Ao final do processo, o valor em *pd(0,0)* representa a energia mínima necessária para percorrer o caminho da célula inicial *(0,0)* até a célula final *(R-1, C-1)*.

#### 3.1.1 Principais funções:

**a)** Função monta_gridPD() é a função responsável por inicialmente montar a matriz de programação dinâmica:

```
void monta_gridPD(int R, int C, int **pd) {
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++) {
            pd[i][j] = INF;
        }
    }
```

```
    }
```

---

**b)** Função dinamica() é a responsável por todo calculo da matriz de programação dinâmica:

---

```
    void dinamica(int R, int C,int **pd, int **grid) {
        pd[R-1][C-1] = 1;

        for (int i = R-1; i >= 0; i--) {
            for (int j = C-1; j >= 0; j--) {

                if (i == R-1 && j == C-1){
                    continue;
                }
                int energMin = INF;
                if (i+1 < R) {
                    energMin = pd[i+1][j];
                }
                if (j+1 < C) {
                    if (energMin < pd[i][j+1]) {
                    } else {
                        energMin = pd[i][j+1];
                    }
                }
                if (energMin - grid[i][j] > 0) {
                    pd[i][j] = energMin - grid[i][j];
                } else {
                    pd[i][j] = 1;
                }
            }
        }
```

---

**c)** Função monta grid() é a função responsável por montar o grid obtido do arquivo de entrada:

---

```
    void monta_grid(FILE* entrada, int R, int C, int **grid) {
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++) {
            fscanf(entrada, "%d", &grid[i][j]);
        }
    }
}
```

---

Esta função é compartilhada pelas outras estratégias também. Portanto, ela só será mencionada aqui.

### 3.2 Modelagem algoritmo que utiliza a heurística gulosa:

Tendo o mesmo objetivo e especificações da abordagem que utiliza programação dinâmica, o algoritmo recebe os parâmetros de entrada, monta o grid e aplica a lógica gulosa percorrendo todos os elementos da matriz *S* e atualizando a energia acumulada.

Quando a energia acumulada se torna insuficiente (menor ou igual a zero) para percorrer a matriz a partir de um determinado ponto, o algoritmo atualiza a energia mínima necessária. Logo após isso ele garante que a energia seja no mínimo 1 o que faz com que o percurso da matriz seja feito com a menor energia aproximada. Ao iniciar a energia acumulada em um ponto em que ela não é suficiente, o algoritmo evita o acúmulo de uma energia negativa que poderia ser arrastada para as próximas posições da matriz. O algoritmo guloso garante que a energia mínima seja sempre pelo menos 1 e busca otimizar a energia acumulada em cada ponto da matriz.

#### 3.2.1 Principais funções:

a) Função guloso() é a responsável pela abordagem gulosa:

```
int guloso(int R, int C, int **grid) {

 int minEnerg = 1;
 int energia = 1;

 for (int i = 0; i < R; i++) {
       for (int j = 0; j < C; j++) {
             energia += grid[i][j];
             if (energia <= 0) {
             minEnerg = maxValor(minEnerg, abs(energia) + 1);
             }
             energia = maxValor(1, energia);
       }
 }
 printf("%d\n", minEnerg);
 return minEnerg;
}
```

**4. Análise de complexidade:**

Para determinar a complexidade do tempo e espaço do código, é necessário analisar quanto tempo leva para executar o pior caso em relação ao tamanho da entrada.

Algoritmo de programação dinâmica percorre toda a matriz de entrada, com R linhas e C colunas, em dois loops aninhados. Em cada iteração, ele realiza algumas operações de verificação e atualização de variáveis. É necessário um espaço adicional para armazenar duas matrizes, 'pd' e 'grid', ambas com R linhas e C colunas. Portanto, sua complexidade tanto de tempo quanto de espaço é O(R x C).
Esse algoritmo é eficiente para problemas com grids de tamanho razoável, pois é capaz de calcular a solução ótima de forma rápida, evitando a repetição de cálculos desnecessários. No entanto, o espaço utilizado pode ser significativo para grandes grids.

O Algoritmo Guloso também percorre toda a matriz de entrada em dois loops aninhados, em cada iteração ele realiza a soma do valor da posição atual e faz uma verificação.Portanto do mesmo modo do algoritmo dinâmico, a complexidade de tempo do algoritmo guloso também é O(R x C).
O algoritmo guloso se faz igualmente eficiente para problemas com grids de tamanho razoável. No entanto, sua abordagem gulosa pode não garantir a solução ótima em todos os casos, pois faz escolhas locais de acordo com a informação disponível no momento. Em algumas situações, o algoritmo guloso pode fornecer soluções aproximadas. No entanto, a simplicidade e eficiência deste algoritmo o tornam uma boa opção em muitos cenários.

**4.1 Tabela da complexidade de tempo:**

| **Algoritmo** | **Melhor caso** | **Caso médio** | **Pior caso** |
|---|---|---|---|
| Algoritmo programação dinâmica | O(1) | O(R x C) | O(R x C) |
| Algoritmo Guloso | O(1) | O(R x C) | O(R x C) |

**5. Resultados:**

Com exceção dos 3 exemplos já dados pelo professor, as demais entradas foram geradas utilizando um gerador de entradas feito para o

problema. Os tamanhos gerados para linhas e colunas estão sempre no intervalo [100,1000], ou seja, não ultrapassam uma matriz S[1000,1000]. Já os valores em cada posição S[i,j] estão sempre entre -1000 e 1000. O gerador não possui limitação quanto a quantidade de casos de testes.

Ambiente de compilação e execução : “WSL 2 Ubuntu 22.04 LTS”
Compilador : MinGW gcc
Processador : “Ryzen 5 5600H” e “Intel(R) Core(TM) i5-7200U ” .

Cada teste foi executado 3 vezes. Os tempos de execução foram obtidos a partir da
média aritmética de cada execução.
Obs: ‘s’ = Segundos.

**5.1 Tabela dos tempos de execução:**

| Tamanho da entrada | | 3 Exemplos do professor | 10 Entradas | 50 Entradas | 100 Entradas | 500 Entradas |
|---|---|---|---|---|---|---|
| Tempos de Usuário | Programação Dinâmica | 0.000442s | 0.152283s | 0.853845s | 1.068600s | 6,909560s |
| | Algoritmo Guloso | 0.000457s | 0.123176s | 0.770481s | 1.435438s | 7,690707s |
| Tempos de Sistema | Programação Dinâmica | 0.000000s | 0.000000s | 0.014013s | 0.026613s | 0,163241s |
| | Algoritmo Guloso | 0.000000s | 0.009771s | 0.010884s | 0,079906s | 0,206576s |

**5.1.2 Análise dos tempos:**

Quanto ao tamanho das entradas observamos que a abordagem gulosa se demonstrou mais eficiente em termos de tempo, para entradas com menos de 50 casos de teste, a partir desse número o algoritmo de programação dinâmica passa a ser mais eficiente.

### 5.2 Tabela dos resultados de entrada:

| Matrizes | 2x3<br>0 1 -3<br>1 -2 0 | 2x2<br>0 1<br>2 0 | 3x4<br>0 -2 -3 1<br>-1 4 0 -2<br>1 -2 -3 0 | 1x5<br>0 -1 -3 1 0 | 5x1<br>0<br>-1<br>-3<br>1<br>0 |
|---|---|---|---|---|---|
| Algoritmo Dinâmico | 2 | 1 | 2 | 5 | 5 |
| Algoritmo Guloso | 2 | 1 | 3 | 3 | 3 |

#### 5.2.2 Análise dos resultados:

Como problema se trata da minimização da vida inicial, o algoritmo dinâmico me garante a solução ótima em todas as entradas, até mesmo em casos excêntricos como as matrizes únicas demonstradas na tabela.

Já o algoritmo guloso conseguiu solução ótima em 2 das 3 entradas exemplo, o que é uma boa aproximação. Apesar dessa boa aproximação, para as entradas de matrizes únicas ele acabou retornando valores que contradizem as regras propostas para o nosso problema, demonstrando que para casos excêntricos ele não se sai tão bem. Isto provavelmente se dá devido ao jeito que as matrizes estão organizadas, já que pode haver trechos no qual momentaneamente um caminho pareceu melhor para o algoritmo guloso e logo após isso ele encontrou muitas posições com valores negativos.

## 6. Conclusão:

Em síntese, a Programação Dinâmica e o Algoritmo Guloso são abordagens eficientes. A programação dinâmica é ideal quando é indispensável a solução ótima e também quando ela pode ser decomposta em subproblemas sobrepostos, enquanto o algoritmo guloso é uma escolha adequada quando uma solução aproximada é suficiente e a escolha local leva a uma solução globalmente ótima.

Independentemente disso, podemos dizer que o algoritmo de programação dinâmica é a escolha a se fazer já que o algoritmo guloso é diretamente afetado pelo método de como as matrizes estão organizadas, podendo voltar valores muito distantes da solução ótima.

Maio, 2023.

## 7. Bibliotecas:

Para o funcionamento do programa, é necessário incluir as seguintes bibliotecas:

```
#include <stdio.h> //Biblioteca padrão de entrada e saída.
#include <stdlib.h> //Funções de alocação de memória.
#include <stdbool.h> //Manipulação de variáveis lógicas.
#include <sys/time.h> //Medição dos tempos.
#include <sys/resource.h> //Medição dos tempos.
```

## Referências:

[1] Gross, J. L., & Yellen, J. (2005). Handbook of Graph Theory. CRC Press.
[2] Szwarcfiter, J. L., & Markenzon, L. (2007). Algoritmos em grafos. LTC.
[3] UNB. Programação Dinâmica. Slides de aula. Brasília: Universidade de Brasília, 2021. Acesso em: 03 maio 2023.