Documentação - Trabalho Prático 2 de Algoritmos e Estruturas de Dados III

Arthur Vieira Silva e Felipe Augusto Moreira Chaves

18 de Maio de 2023

Sumário

1 Contextualização

Neste trabalho prático, temos como objetivo principal determinar, a partir de um grid S com dimensões RxC, as quais representam as linhas e as colunas do Grid, respectivamente, qual seria a energia mínima necessária para iniciarmos na posição $S[1][1]$ e chegarmos até a posição $S[R][C]$, de modo que a energia não seja menor ou igual a 0 durante todo o percurso.

Além disso, temos que as linhas são enumeradas de 1 à R de cima para baixo e as colunas de 1 à C da esquerda para a direita. Temos também uma observação importantíssima que só podemos nos movimentar para baixo $(i+1, j)$ ou para a direita $(i, j+1)$, e, em cada célula do Grid, temos uma poção ou um monstro. Uma poção na célula (i, j) aumenta $S[i][j]$ da nossa energia, enquanto um monstro na célula (i, j), retira $S[i][j]$ da nossa energia.

Dessa maneira, implementamos duas estratégias para solucionarem esse problema descrito anteriormente. A seguir, introduziremos ambos os métodos adotados, listaremos as rotinas utilizadas durante a implementação, faremos uma análise de complexidade dessas rotinas, além de uma análise dos resultados obtidos durante algumas execuções, por fim faremos uma breve conclusão a respeito do trabalho realizado.

2 Introdução

Na resolução deste prático, após uma profunda análise de diversos métodos para solucionar o problema proposto, decidimos implementar duas estratégias, na primeira estratégia, utilizamos um **Algoritmo Guloso** que em alguns casos produz a resposta desejada e em outros produz uma resposta aproximada, na segunda tentiva, utilizamos **Programação Dinâmica** para encontrarmos a solução do problema.

2.1 Primeira Estratégia

Inicialmente, pensamos em uma maneira de resolver o problema por meio de um **Algoritmo Guloso.** Nessa implementação, após a leitura do arquivo de entrada, em cada caso de teste, iniciamos com o mínimo de energia na

posição $S[1][1] = 1$. Temos uma variável que armazena a nossa energia durante todo trajeto, iniciando também com o valor 1. Nessa abordagem gulosa proposta, dada uma energia em uma posição $S[i][j]$, temos uma função que seleciona se iremos para a posição abaixo da posição atual ou à direita da posição atual.

Essa função de seleção escolha a posição em que há o maior ganho de energia ou a menor perda de energia. Porém, se estivermos em uma posição na qual só é possível caminhar para baixo, isto é, atingimos a última coluna do Grid, então só temos a opção de ir para a posição $(i+1, j)$ e, se estivermos em uma posição na qual só é possível caminhar para a direita, isto é, atingimos a última linha do Grid, então só temos a opção de ir para a posição $(i, j+1)$.

Além disso, se estivermos em uma posição no Grid na qual qualquer escolha que fizermos a nossa energia ficará menor que 1 então, a energia mínima que iniciamos na posição $S[1][1]$ será incrementada em uma unidade, retornaremos para a posição inicial e iremos realizar os mesmos passos descritos anteriormente com a nova energia mínima necessária.

Essa é uma abordagem gulosa pois, escolhe aquele caminho que parece ser mais promissor no momento mas, após a escolha de uma determinada posição, essa decisão nunca é reconsiderada, independentemente do que venha acontecer no futuro.

O exemplo a seguir retrata o comportamento do algorimo descrito anteriormente.

Ex: Dado o grid S de dimensões 3x3:

$$\begin{matrix} 0 & -2 & 3 \\ 1 & -3 & 5 \\ -3 & 4 & 0 \end{matrix}$$

1- Note que, ao iniciarmos com energia mínima igual à 1 e energia atual também igual 1, a primeira escolha será caminharmos para $S[2][1]$ e a nossa energia será 2 porém, note que, estando na posição $S[2][1]$, se escolhermos ir tanto para baixo quanto para a direita, a nossa energia ficará menor que 1.

Sendo assim, retornamos para a posição inicial e iniciaremos com o mínimo de energia e energia atual com o valor 2.

2- Novamente iremos caminhar para baixo atualizando a nossa energia atual para 3 mas, vamos nos deparar com a mesma situação anterior. Desse modo, iremos voltar novamente para a posição inicial e com um mínimo de energia e energia atual iguais à 3.

3- Agora caminhamos para baixo, atualizando a nossa energia atual para 4. Após chegarmos na posição $S[2][1]$, note que qualquer um dos dois caminhos que escolhermos teremos o mesmo impacto. Nesse caso, comparamos as posições $S[i+1][j+1] = 4$ e $S[i][j+2] = 5$, escolhendo aquela que for maior ou igual a outra, tentando evitar uma futura escolha equivocada. Dessa maneira, vamos para posição $S[3][1]$ atualizando nossa energia atual para 1. Por fim, iremos para as posições $S[3][2]$ e $S[3][3]$, alcançando a última posição do Grid com energia atual igual à 5.

Logo, o mínimo de energia que precisamos ter na posição $S[1][1]$ nesse problema é **3.**

Portanto, note que essa abordagem gulosa produz uma saída correta em alguns casos porém, pode ser que, para certas entradas, tenhamos uma saída aproximada, já que em algum passo ao caminhar no Grid, a decisão feita pode não ter sido a melhor escolha, afetando assim, o resultado final.

2.2 Segunda Estratégia

Para uma segunda abordagem do problema então descrito, implementamos uma estratégia baseada na técnica de **Programação Dinâmica.** Nesse paradigma, calculamos a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela. Desse modo, a grande utilidade de utilizarmos essa técnica é que uma vez que um subproblema é resolvido, a resposta é armazenada em uma tabela e, portanto, não há necessidade de se calcular o subproblema novamente.

Para utilização desse método, criamos uma segunda matriz auxiliar denominada mat com as mesmas dimensões do Grid S. Vamos percorrer essa matriz da primeira posição $mat[1][1]$ até a última posição $mat[R][C]$ e, em cada posição da matriz será armazenado o valor da energia mínima necessária até o momento. Para isso, temos que:

- Se $i = 1$ e $j = 1$ então, a energia mínima será $mat[1][1] = 1$;
- Se $1 < i \leq R$ e $1 < j \leq C$ então, $mat[i][j] = S[i][j] + min(mat[i-1][j], mat[i][j-1])$. Porém, se $S[i][j] > 0$, temos que $mat[i][j] = min(mat[i-1][j], mat[i][j-1])$;
- Se $i = 1$ e $j > 1$ então, $mat[i][j] = S[i][j] + mat[i][j-1]$. Porém, se $S[i][j] > 0$, temos que $mat[i][j] = mat[i][j-1]$;
- Se $i > 1$ e $j = 1$ então, $mat[i][j] = S[i][j] + mat[i-1][j]$. Porém, se $S[i][j] > 0$, temos que $mat[i][j] = mat[i-1][j]$;
- Por fim, se $mat[i][j] \leq 0$ então, $mat[i][j] = (mat[i][j] \times -1) + 1$.

Ao final do preenchimento da matriz auxiliar, o resultado final estará na célula $mat[R][C]$. O exemplo a seguir retrata o comportamento do algorimo descrito anteriormente:

Ex: Dado o grid S de dimensões 2x3 :

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

A matriz auxiliar, na primeira iteração, estará da seguinte forma:

$$\begin{matrix} 1 & \infty & \infty \\ \infty & \infty & \infty \end{matrix}$$

- $mat[1][2] = 1$;
- $mat[1][3] = -3 + 1 = (-2 \times -1) + 1 = 3$;
- $mat[2][1] = 1$;
- $mat[2][2] = -2 + min(1, 1) = (-1 \times -1) + 1 = 2$;
- $mat[3][3] = 0 + min(3, 2) = 2$.

Logo, a matriz auxiliar ao final do *loop* será:

$$\begin{matrix} 1 & 1 & 3 \\ 1 & 2 & 2 \end{matrix}$$

E o mínimo de energia que precisamos ter na posição $S[1][1]$, nesse problema, é **2.**

Nesse sentido, é possível perceber que nessa estratégia não é necessário realizar os mesmos cálculos repetidamente, uma vez que foi-se utilizado a técnica de **Programação Dinâmica** para a resolução do problema, na qual utilizamos os resultados de subproblemas já calculados para obter os resultados de outros subproblemas maiores, como exemplificado anteriormente.

3 Listagem das Rotinas

No desenvolvimento desse trabalho, foram utilizadas algumas funções que facilitaram tanto o processo de contrução do código quanto o entendimento do programa que estava sendo desenvolvido. Assim sendo, a seguir serão apresentados os procedimentos que foram utilizados em ambas as estratégias, haja vista que as duas estratégias utilizam basicamente os mesmos procedimentos, com exceção de que a segunda estratégia não utiliza as funções **Seleciona** e **Estrategia1,** mas sim a função **Estrategia2:**

- **AlocaGrid:** Esta função foi utilizada apenas para alocar dinamicante cada Grid S com R linhas e C colunas.

- **Seleciona:** Utilizamos esse procedimento para, em cada posição do Grid, selecionarmos qual será a nossa futura posição. Ele recebe como paramentos o próprio Grid S, a nossa energia atual, o número de linhas e colunas do Grid, além da nossa posição atual (i, j). Nesta função sempre escolhemos a próxima posição que haverá o maior ganho de energia ou a menor perda de energia. Primeiramente, se for possível caminhar tanto para baixo quanto para a direita então, comparamos as posições $S[i+1][j]$ e $S[i][j+1]$ e escolhemos a mais vantajosa. Porém, se essas posições possuem valores iguais, tentaremos minimizar ao máximo uma futura escolha errada, comparando as posições $S[i+1][j+1]$ e $S[i][j+2]$ e selecionando a que for mais vantajosa. Por outro lado, se estivermos na última linha do Grid, só podemos optar em ir para a direita e, se estivermos na última coluna do Grid, só podemos caminhar para baixo.

- **Estrategia1:** Nesta função, implementamos uma abordagem gulosa para resolução do problema, utilizando a função **Seleciona** descrita anteriormente. Para isso, iniciamos com o mínimo de energia igual a 1, a energia atual igual a 1 e os índices i e j também iguais ao valor

1. Dessa maneira, teremos um *loop* que será realizado enquanto não alcançarmos a posição $S[R][C]$ no Grid. Dentro desse laço, chamamos a função **Seleciona** a todo momento, atualizando a energia atual e os índices i e j. Contudo, se em algum momento a nossa energia atual atingir um valor abaixo de 1, a energia mínima necessária será incrementada, a energia atual receberá a energia mínima e retornaremos para a posição $S[1][1]$. Desse modo, ao final do procedimento retornaremos o mínimo de energia para cada caso de teste.

- **LiberaGrid:** Esta função foi utilizada apenas para liberar cada Grid após obtermos o resultado final.
- **Estrategia2:** Procedimento utilizado na técnica de **Programação Dinâmica.** Alocamos uma matriz auxiliar $mat[R][C]$ e também temos duas variáveis que armazenam os valores de $mat[i-1][j]$ e $mat[i][j-1]$. Temos um *loop* que será realizado até a posição $mat[R][C]$, no qual armazenamos, em cada posição da matriz auxiliar, o mínimo de energia que é necessário em cada momento, fazendo o uso de valores previamente armazenados. Preenchemos essa matriz de acordo com as regras já vistas na seção 2.2. Ao final, o resultado estará em $mat[R][C]$.
- **SalvarOutput:** Utilizada apenas para salvarmos o arquivo de saída com os resultados obtidos, por meio de um *array* de tamanho t.
- **LerArquivo:** Por fim, este procedimento foi utilizado para a leitura de todos os dados do arquivo de entrada fornecido. Dentro desta função, chamamos outros procedimentos, tais como **AlocaGrid**, **Estrategia1** (se ela for selecionada), **Estrategia2** (se ela for selecioanada) e **SalvarOutput.**

4 Análise de Complexidade das Rotinas

Vamos determinar qual a **função de complexidade** e a **ordem** do algoritmo implementado. Para isso, consideramos como operações mais significativas o **número de comparações** realizadas. Seja uma **função de complexidade *f*,** tal que $f(n)$ é a medida do tempo necessário para executar o algoritmo para um problema de tamanho n. Faremos uma análise de complexidade de cada rotina individualmente e, por fim, uma análise de cada estratégia.

A função usada para alocar o Grid realiza apenas uma comparação, logo $f(n) = 1$ e ela é $O(1)$. Na função **Seleciona** também não temos *loops* que dependem do tamanho do Grid, as comparações dessa função são realizadas uma única vez, isto é, depende do número de chamadas na função **Estrategia1** logo, $f(n) = 1$ e ela também é $O(1)$.

Na função **Estrategia1,** temos que considerar se será preciso reiniciar a variável *min* ou não. Com isso, considere k o número de vezes que o *loop* é reiniciado para a posição inicial logo, temos que $f(n) = k + (R \times C)$. Note que se não for preciso reiniciar o laço em nenhum momento, $k = 0$. Porém, o valor de k pode ser máximo e tender ao infinito, isto é, a cada iteração é necessário reiniciarmos o *loop.* E o valor exato de k depende dos valores do Grid e de como esses valores estão dispostos.

A função **LiberaGrid** não realiza nenhuma comparação. Já em **Estrategia2,** realizamos 1 comparação para alocarmos a matriz auxiliar e, o número de comparações que são realizadas dentro do laço é $f(n) = R \times C$, ou seja, independente do tamanho do Grid, temos que percorrer todas as suas posições.

A função para salvar o arquivo de saída também realiza apenas uma comparação. Por fim, temos que a função **LerArquivo** possui algumas comparações que são realizadas um número fixo de vezes, isto é, $O(1)$, como para verificar se o arquivo de entrada foi lido corretamente, se a memória foi alocada corretamente e para salvar o arquivo de saída. Já o laço é executado t vezes e $f(n) = t$.

Portanto, temos que a função de complexidade da primeira estratégia adotada é $f(n) = t \times (k + (R \times C)$ e temos que a sua complexidade é $O(R \times C)$, para o melhor caso, pior caso e caso médio. A segunda estratégia possui função de complexidade $f(n) = t \times (R \times C)$ e complexidade também $O(R \times C)$, para o melhor caso, pior caso e caso médio.

5 Análise dos Resultados Obtidos

Após vários testes, selecionamos três exemplos que consideramos mais relevantes para as nossas análises tanto do tempo de sistema quanto do tempo

de usuário, e sua relação com os tempos do relógio. Para isso, utilizamos as funções **getrusage** e **gettimeofday.**

O tempo de usuário representa o tempo de CPU gasto no código no modo usuário (fora do *kernel*) dentro do processo, é apenas o tempo real de CPU usado durante a execução do programa. O tempo de sistema é a quantidade de tempo da CPU gasto durante o período que o programa executa no modo *kernel*. Já o tempo de relógio é todo o tempo decorrido, isto é, do início ao fim da chamada, inclui o tempo utilizado em outros processos e o tempo que a execução fica parada esperando terminar operações de entrada/saída. A seguir estão as 3 matrizes utilizadas para análise:

$$\begin{matrix} 0 & 87 & 49 & 42 & 11 \\ 57 & 27 & 92 & 59 & 19 \\ 35 & 14 & 90 & 17 & 75 \\ 89 & 77 & 98 & 60 & 33 \\ 10 & 69 & 86 & 74 & 0 \end{matrix} \quad \text{(S1)}$$

$$\begin{matrix} 0 & -591 & -643 & -118 & -655 & -784 & -804 & -106 & -446 & -325 & -102 & -502 \\ -322 & -240 & -831 & -316 & -159 & -383 & -936 & -464 & -768 & -929 & -682 & -722 \\ -297 & -737 & -847 & -574 & -327 & -596 & -490 & -247 & -469 & -440 & -394 & -689 \\ -756 & -166 & -757 & -111 & -452 & -715 & -549 & -881 & -362 & -709 & -328 & -218 \\ -884 & -356 & -273 & -587 & -843 & -566 & -822 & -249 & -273 & -751 & -298 & -784 \\ -437 & -810 & -459 & -646 & -992 & -972 & -351 & -938 & -118 & -859 & -220 & -711 \\ -263 & -876 & -594 & -642 & -991 & -826 & -642 & -734 & -497 & -763 & -414 & -209 \\ -617 & -245 & -320 & -408 & -646 & -776 & -343 & -886 & -597 & -114 & -772 & 0 \end{matrix} \quad \text{(S2)}$$

$$\begin{matrix} 0 & -3 & -19 & 4 \\ 16 & -17 & 0 & 3 \\ -20 & -18 & -25 & 11 \\ 17 & -19 & 19 & -11 \\ 1 & -15 & -17 & -16 \\ 1 & 11 & 19 & 0 \end{matrix} \tag{S3}$$

O grid $S1$ tem dimensões 5x5 e possui apenas poções nas suas células, o que faz com que o algortimo guloso execute de maneira mais rápida. O segundo grid possui 8 linhas e 12 colunas e possui apenas monstros em suas células, o que torna a primeira estratégia menos eficiente. Já o Grid $S3$ de dimensões 6x4, possui poções e monstros em suas células e, isso pode prejudicar a execução da primeira estratégia. A seguir temos uma tabela que comprava tais afirmações por meio de dados a respeito do tempo de sistema, tempo de usuário e tempo de relógio:

Dados obtidos com as funções getrusage e gettimeofday		
Variação do tempo	Estratégia 1	Estratégia 2
Sistema	0s - 0,000749s	0s - 0,000497s
Usuário	0,000484s - 0,000635s	0,000404s - 0,000527s
Relógio	0,000482s - 0,000632s	0,000401s - 0,000526s

6 Conclusão

Portanto, na realização deste trabalho conseguimos implementar a **abordagem gulosa** sem maiores problemas, por ser um método que não necessariamente vai encontrar a solução correta, isto é, encontra soluções

subótimas para o problema. Já na implementação da estratégia utilizando **Programação Dinâmica,** não conseguimos solucionar o problema da maneira desejada desse modo, em certos testes, não conseguimos obter a resposta correta mas, em vários testes, conseguimos chegar ao resultado desejado.

7 Referências

ZIVIANI, N. **Projeto de Algoritmos: com implementações em PASCAL e C.** 3 ed. São Paulo: Cengage Learning, 2013.