Ciência da Computação

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

AEDS III Trabalho Prático II

Alunos:
Vinicius de Almeida Lima (212050070)
Matheus Duraes da Cunha Pereira (212050094)

Professor: Leonardo Rocha

Introdução

No trabalho prático de AEDS 3 nos foi proposto resolver um problema para Harry Potter. O problema consiste em ajudar Harry a atravessar um tabuleiro mágico, Harry só pode andar para o lado direito ou para baixo. cada célula do tabuleiro possui uma poção ou um monstro e Harry precisa saber qual o menor valor de vida ele precisa para atravessar o tabuleiro sem morrer.

Contextualização

O problema proposto consiste em implementar uma forma de descobrir qual o numero de vida mínima de Harry para que ele consiga atravessar o tabuleiro sem sua vida chegar a 0.

usando os conhecimentos adquiridos na disciplina de AEDS III foi proposto a seguinte solução.

Modelagem e Solução

Para modelarmos o problema foi definido uma matriz que representa o tabuleiro, e devemos descobrir qual o menor valor da posicao [0][0] para que possa-se chegar a posição [LINHA][COLUNA] sem que a soma nunca chegue a 0 ou menos. Podemos interpretar a matriz como um grafo que se caminha para baixo ou para o lado direito.

Como foi pedido duas soluções, foi proposto uma solução utilizando Busca em Profundidade e outra solução utilizando Programação Dinâmica

Busca em Profundidade

Para essa solução utilizamos um algoritmo de Busca em Profundidade. implementamos uma busca em profundidade recursiva em uma matriz bidimensional. Ela é usada para percorrer a matriz a partir de uma posição inicial (i, j) até a posição final (LIN-1, COL-1), explorando todas as possíveis combinações de caminhos possíveis.

Essa solução foi implementada usando a função buscaEmProfundidade() que recebe os seguintes parametros:

- "matriz": um ponteiro para a matriz de números inteiros referente ao tabuleiro.
- "visited": um ponteiro para uma matriz auxiliar que indica quais posições da matriz original já foram visitadas durante a busca.
- "LIN" e "COL": os tamanhos da matriz (número de linhas e colunas, respectivamente).
- "i" e "j": as coordenadas atuais na matriz durante a recursão.
- "minNum": a soma mínima atual dos números encontrados durante a busca.
- "sum": a soma acumulada dos números encontrados durante a recursão.
- "minNumAbsoluto": o valor mínimo absoluto encontrado até o momento.

A função começa adicionando o valor atual da posição (i, j) à soma acumulada "sum". Se a soma for negativa, é ajustado o valor de "minNum" somando o valor absoluto da soma negativa, e a soma é reiniciada como zero. Isso garante que "minNum" armazene a soma mínima de todos os caminhos possíveis.

Em seguida, a função verifica se a posição atual é a posição final (LIN-1, COL-1). Se for, compara "minNum" com "minNumAbsoluto" e atualiza "minNumAbsoluto" se "minNum" for menor.

Após isso, a posição atual é marcada como visitada no matriz auxiliar "visited" e são feitas duas chamadas recursivas para continuar a busca em profundidade. Primeiro, verifica se é possível mover para baixo (i+1, j), desde que a posição não tenha sido visitada anteriormente. Em seguida, verifica se é possível mover para a direita (i, j+1), seguindo a mesma condição.

Depois de explorar todos os caminhos possíveis a partir da posição atual, a posição é marcada como não visitada novamente (visited[i][j] = 0) e "minNumAbsoluto" é retornado.

No geral, a função busca por todos os caminhos possíveis na matriz, calculando a soma mínima ao longo de cada caminho, e retorna o valor mínimo absoluto encontrado.

- **Análise de Complexidade:**

A complexidade da função "buscaEmProfundidade" depende do tamanho da matriz, representado pelos parâmetros LIN e COL. Vamos denotar o tamanho da matriz como "N = LIN * COL".

  - O tempo de execução da função é afetado principalmente pela recursão e pelas condições de parada.
  - Durante a recursão, a função explora todas as possíveis combinações de caminhos na matriz. Para cada célula, há no máximo duas chamadas recursivas (uma para baixo e outra para a direita), desde que a célula adjacente não tenha sido visitada antes. Portanto, o número máximo de chamadas recursivas é proporcional ao número de células na matriz, ou seja, LIN * COL.
  - Além das chamadas recursivas, há operações simples executadas em cada célula, como soma, comparação e atribuição. Essas operações têm complexidade O(1) e são executadas no máximo LIN * COL vezes.

Portanto, a complexidade de tempo total da função é O(LIN * COL) no pior caso, onde LIN e COL representam o tamanho da matriz.

Quanto ao espaço, a função utiliza duas matrizes auxiliares: "visited" e "matriz". Ambas têm o mesmo tamanho da matriz original, portanto, a complexidade de espaço é O(LIN * COL) no pior caso.

Programação Dinamica

Para essa solução implementamos um algoritmo de programação dinâmica para encontrar o menor caminho em uma matriz de custos. A função recebe uma matriz de inteiros "matriz" com "LIN" linhas e "COL" colunas como entrada.

O algoritmo segue os seguintes passos:

- Aloca três matrizes de inteiros bidimensionais, chamadas de matrizDP, matrizDPAux e matrizDPSoma, com as mesmas dimensões da matriz de entrada.
- Preenche a primeira linha e a primeira coluna da matrizDP com os valores acumulados da primeira linha e da primeira coluna da matriz de entrada, respectivamente. Também preenche as matrizes auxiliares matrizDPAux e matrizDPSoma com valores mínimos e somas acumuladas, respectivamente.
- Preenche o restante da matrizDP, linha por linha, coluna por coluna, seguindo a seguinte lógica:
    - Calcula o valor máximo entre o elemento acima (matrizDP[i-1][j]) e o elemento à esquerda (matrizDP[i][j-1]) na matrizDP.
    - Verifica qual dos dois elementos resultou no máximo valor e realiza as seguintes ações:
        - Se o elemento acima foi o máximo, atualiza a soma acumulada e o valor mínimo da matrizDPSoma e matrizDPAux, respectivamente, considerando o elemento acima.
        - Se o elemento à esquerda foi o máximo, atualiza a soma acumulada e o valor mínimo da matrizDPSoma e matrizDPAux, respectivamente, considerando o elemento à esquerda.
    - Verifica se a soma acumulada é negativa e, se for, ajusta o valor mínimo para o valor absoluto da soma acumulada e redefine a soma acumulada como zero.
    - Armazena na matrizDP o valor do elemento atual da matriz de entrada somado ao máximo valor encontrado.
    - Atualiza a matrizDPAux e matrizDPSoma com os valores mínimo e soma acumulada atualizados.
- Retorna o valor da última posição da matrizDPAux, que representa o valor mínimo do caminho percorrido na matriz.
- Libera a memória alocada para as matrizes auxiliares.

Em resumo, a função utiliza a programação dinâmica para calcular o caminho de menor custo em uma matriz, considerando que é possível mover apenas para a direita ou para baixo.

Análise de Complexidade:

A análise de complexidade da função "programacaoDinamica" pode ser feita considerando o número de linhas (LIN) e colunas (COL) da matriz de entrada.

- Alocação de memória para as matrizes auxiliares:
    - Alocação de memória para a matrizDP: O(LIN * COL), pois é necessário alocar LIN * COL elementos inteiros.
    - Alocação de memória para a matrizDPAux: O(LIN * COL), pois é necessário alocar LIN * COL elementos inteiros.
    - Alocação de memória para a matrizDPSoma: O(LIN * COL), pois é necessário alocar LIN * COL elementos inteiros.

  Portanto, a complexidade total para alocar memória é O(LIN * COL).
- Preenchimento da primeira linha e da primeira coluna da matrizDP:
    - O preenchimento da primeira linha requer percorrer COL elementos da matriz de entrada e executar operações constantes em cada um deles. Portanto, a complexidade é O(COL).
    - O preenchimento da primeira coluna requer percorrer LIN elementos da matriz de entrada e executar operações constantes em cada um deles. Portanto, a complexidade é O(LIN).

  Assim, a complexidade total para preencher a primeira linha e a primeira coluna é O(LIN + COL).
- Preenchimento do restante da matrizDP:
    - Existem dois loops aninhados. O loop externo executa LIN - 1 iterações e o loop interno executa COL - 1 iterações.
    - Dentro do loop interno, há um conjunto de operações constantes que são executadas, como comparações, adições, atualizações de valores nas matrizes auxiliares e verificação de soma negativa.

  Portanto, a complexidade do preenchimento do restante da matrizDP é O((LIN - 1) * (COL - 1)) = O(LIN * COL).
- Liberação de memória alocada:
    - Liberação de memória para a matrizDP: O(LIN * COL), pois é necessário liberar LIN * COL elementos inteiros.
    - Liberação de memória para a matrizDPAux: O(LIN * COL), pois é necessário liberar LIN * COL elementos inteiros.
    - Liberação de memória para a matrizDPSoma: O(LIN * COL), pois é necessário liberar LIN * COL elementos inteiros.

  Portanto, a complexidade total para liberar memória é O(LIN * COL).

Em resumo, considerando todas as etapas da função, a complexidade total é:

O(LIN * COL) + O(LIN + COL) + O(LIN * COL) + O(LIN * COL) + O(LIN * COL) = O(LIN * COL).

Assim, a complexidade da função "programacaoDinamica" é O(LIN * COL), onde LIN é o número de linhas e COL é o número de colunas da matriz de entrada.

Listagem das Rotinas

- InputFile():
  Essa função abre um arquivo especificado pelo usuário ao executar o programa. Verifica se o arquivo foi aberto corretamente. Então lê o número de testes a serem realizados. Para cada teste lê o número de linhas e colunas do tabuleiro e então aloca o tabuleiro em uma matriz e também aloca uma matriz de coloração (visited).
  A função verifica qual estratégia será utilizada para resolver o problema e então passa a matriz para a função respectiva da estratégia escolhida.
  A estratégia escolhida retorna o resultado para a função de InputFile que então armazena em um array de resultados e o repassa para a função responsável por criar o arquivo de saída.
- outputFile():
  Essa função é responsável por criar o arquivo de saída contendo os resultados de cada estratégia.
- Print_Matriz:
  Essa função recebe uma matriz com o número de linhas e colunas e printa ela. (não foi usada em nenhuma solução, apenas para auxiliar no desenvolvimento do código)
- buscaEmProfundidade():
  Essa função é responsável pela solução ótima de busca em profundidade listada acima.
- programacaoDinamica():
  Essa função é responsável pela solução ótima de programação dinâmica listada acima.
- melhorCaminho():
  Função auxiliar da estratégia de programação dinâmica para decidir qual o melhor caminho a ser seguido para construir uma nova matriz.

Conclusão

De acordo com os testes realizados, pode-se concluir que os algoritmos funcionaram de forma esperada para as entradas, tanto em tempo de execução quanto em resultados finais. A única limitação do programa são para tabuleiros extremamente grandes, visto que, para entradas extremamente grandes pode não ser possível alocar espaço dinâmicamente para o tamanho necessário, gerando assim uma sobrecarga na pilha e um erro no programa.

Ademais, para atender ao trabalho proposto, fez-se necessário uma pesquisa acerca de algoritmos de busca em profundidade e um entendimento sobre programação dinâmica com o intuito de entregar soluções ótimas ao problema proposto.

Considerações Finais

Em geral concluímos que a experiência da realização do trabalho foi proveitosa e envolveu de forma coerente os conteúdos trabalhados na matéria de AEDS 3 . Sendo portanto uma maneira efetiva e desafiadora de aprendizado e prática.

Referências

[1] Slides dispostos no Campus Virtual da Universidade Federal de São João Del-Rei (UFSJ), na disciplina de Algoritmos e Estrutura de Dados III: https://campusvirtual.ufsj.edu.br/portal/2023_1n/course/view.php?id=1198

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Algoritmos: Teoria e Prática.* 3a edição. Elsevier, 2012. ISBN 9788535236996