UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

ALGORITMOS E ESTRUTURAS DE DADOS III

Aluno: Lucas Lagôa Nogueira
Curso: Ciência da Computação
Professor: Leonardo C. Rocha

São João del Rei, abril de 2016

1 – Introdução

A humanidade sempre procurou meios de facilitar sua vida, utilizando o ramo da matemática e também da computação, como podemos observar no trabalho prático, que uma analogia à um campo de morangos com algumas regras de colheita, pode ser um objeto de estudo de programação e outras estruturas de dados.

Sobre o assunto de paradigmas de programação que foi o escolhido para resolver o problema proposto, tem como definição o conceito que pressupõe a forma que o programador tem sobre a programação e execução de um programa, ou seja, como o programador vai estruturar o problema. E especificamente nesse trabalho, será abordado a programação dinâmica.

2 – Soluções e propostas de implementação

Para a resolução do problema proposto pelo trabalho prático, foi utilizado um dos paradigmas de programação, o método de programação dinâmica, que é um método que a solução ótima pode ser computada a partir da solução previamente calculada e memorizada.

2.1 – Programação dinâmica:

Visando resolver o problema proposto cujo era resolver o problema de seu amigo da colheita de morangos, só que com algumas regras para percorrer esse caminho, procurei utilizar princípios da programação dinâmica, pois ela se encaixaria seria bem aplicada no problema.

Como a programação dinâmica utiliza uma solução ótima previamente calculada para os próximos passos, resolvi aplicar esse método primeiramente calcular a primeira coluna, que seria o caso base, já que na primeira coluna só podemos realizar movimentos para cima e para baixo. Então, ela verificaria se ela possui algum zero (bloqueio de caminho), e realizaria a maior soma possível entre os números que a contém, tanto para cima e para baixo, e sempre pegaria a maior soma, que no caso seria o resultado ótimo.

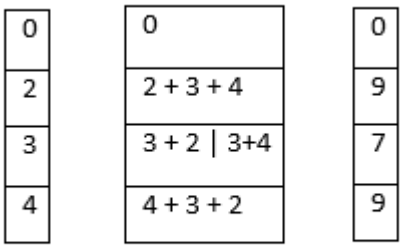

Figura 1: Esquema da primeira coluna.

Como podemos observar na figura 1, o resultado adquirido é o resultado ótimo que iremos utilizar previamente para calcular o restante dos caminhos no campo de morango. E esse resultado ótimo, que está armazenado na matriz de resultados, será utilizado somando com a próxima coluna (simulando o caminhamento para a direita) e guardando

esses valores em dois vetores, um que simula a subida e outro que simula a descida, e previamente estes vetores receberão o mesmo valor. Nessa soma, devemos observar se algum dos valores é 0. Se algum dos valores for 0, não podemos realizar o movimento então ambos vetores receberão 0 nessa posição, caso não, recebem o valor da soma normalmente em cada posição.

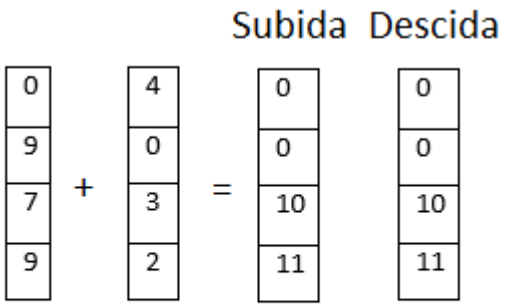

Figura 2: Esquema dos primeiros vetores de subida e descida.

Observando a figura 2, podemos analisar agora as bordas de cada vetor separadamente, pois as bordas podem ocorrer os casos de teletransporte. Em ambos os casos, devemos comparar os valores do vetor e o valor da coluna analisada no momento, pegando o maior valor e substituindo em um dos seus novos vetores atualizados.

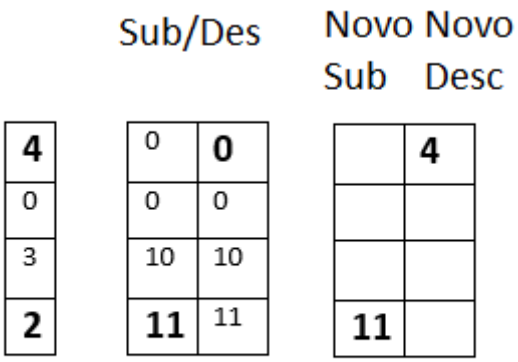

Figura 3: Análise das bordas dos vetores de subida e descida.

Depois da análise das bordas, devemos agora verificar agora os valores mais internos de cada vetor, seguindo uma regra estabelecida.

- Comparar se os valores dos vetores subida/descida na posição com a coluna analisada no momento, são 0.
- Se forem 0, os novos vetores recebem 0 também
- Caso não seja, devemos verificar o anterior na coluna, se ele for diferente de 0, os novos vetores recebem da cédula e o valor da coluna (próximo na descida e anterior na subida).

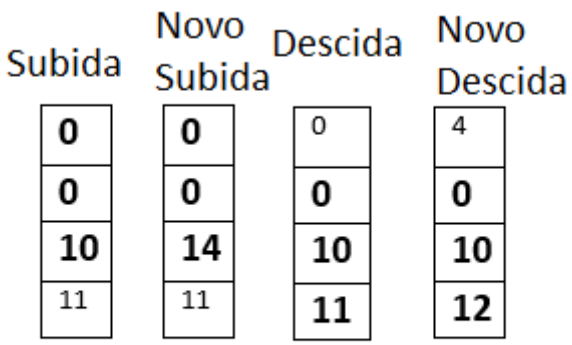

Figura 4: Cálculos para os novos vetores de subida e descida.

E logo após os cálculos, como podemos observar acima, compararemos os valores dos novos vetores de subida e descida, e com os maiores valores formaremos um novo vetor ótimo, que o mesmo será utilizado para refazer os passos que foram descritos acima, até atingir a última coluna (se houverem caminhos disponíveis) que é o critério de parada. E o maior valor da última coluna, será o maior número de morangos possíveis que podemos coletar.

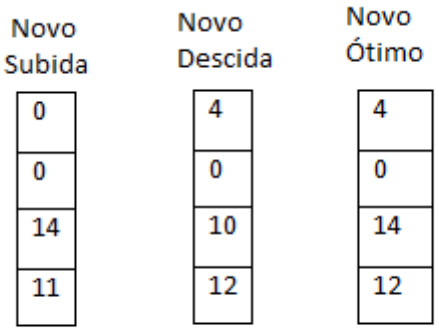

Figura 5: Formação do novo vetor ótimo após os cálculos.

3 – Organização do programa

O programa foi implementado em 5 módulos, que foram no caso o main.c, args.c, args.h, tp2.c, tp2.h, e explicarei um pouco abaixo.

3.1 – main.c

No main.c, foi computado basicamente as funções de tempo e também a leitura dos campos de morango do arquivo de entrada, juntamente com os resultados finais.

3.2 – args.c

No args.c do programa, foi computado a função de leitura de parâmetro argc e argv.

3.3 – args.h

No args.h do programa, foi computado a estrutura do arg e o título da função de leitura de parâmetro.

3.4 – tp2.c

Nesse módulo, possui as funções utilizadas para realizar as operações necessárias, como a de alocar matrizes, calcular a primeira coluna e o restante.

3.5 – tp2.h

Nesse módulo, foi computado os títulos das funções que estão implementadas no módulo de mesmo nome porém .c

E também possui um MakeFile para compilar o programa.

4 – Análise de complexidade

Nessa seção, será debatido uma análise da complexidade das funções utilizadas no algoritmo e também a importância da função para a realização do programa.

4.1 – Função para alocar as matrizes utilizadas pela programação dinâmica.

long int **alocarmatriz (long int linhas, long int colunas);

Essa função depende do número de linhas e colunas para alocar as matrizes que serão utilizadas, primeiramente alocamos um vetor de acordo com o número de linhas, e logo após realiza-se um for para o número de colunas que a matriz apresentará. Então essa função possui uma complexidade de O(n).

4.2 – Função para calcular a primeira coluna.

long int *calcprimeiracoluna(long int **vertices, long int **resultado,long int linhas);

Essa função realiza as operações necessárias para a primeira coluna, e ela possui um for mais externo que percorre o número de linhas e dentro desse for, possui outros dois for que realizam a subida e a descida, para o de subida sua complexidade é O(n) e o de descida possui mesmo comportamento O(n), então para os for mais internos, temos max(O(n),O(n)) = O(n), também devemos destacar operações de atribuições, if e elses, porém eles possuem complexidade O(1), logo max(O(n),O(1)) = O(n). E como possuimos um for mais externo que contém ambos interiores, temos O(linha x coluna) = O(n*n) = $O(n^2)$.

4.3 – Função para calcular o restante das colunas.

void calculaall (long int **vertices, long int **resultado, long int linhas, long int colunas);

Essa função realiza as operações necessárias nas colunas restantes, utilizando um for externo que movimenta as colunas, logo após o fim das operações mais internas, que possui 5 for sendo que:

- O primeiro movimenta as linhas para o preenchimento dos vetores de subida e descida, e algumas condições com if e eles, então sua complexidade é max(O(n),O(1)) = O(n).

- O segundo realiza as operações necessárias do preenchimento do vetor de subida para vetor de subida atualizado com algumas condições if e elses, então sua complexidade é max(O(n),O(1)) = O(n).

- O terceiro realiza as operações necessárias do preenchimento do vetor de descida para o vetor de descida atualizado e algumas condições if e elses, então sua complexidade também é max(O(n),O(1)) = O(n).

- O quarto realiza as operações para verificar qual melhor valor ótimo na situação atual com ifs e elses também, então sua complexidade é max(O(1),O(n)) = O(n).

- E o último transfere o resultado ótimo para a matriz de resultados, então sua complexidade é O(n).

E agora analisando o for mais externo com os internos podemos observar que a complexidade total da função é (O(n) * 5) = O(5n) * O(n) = O(5(linha x coluna)²) = O(n²).

4.4 – Função calcula resultado.

long int resposta (long int **resultado, long int linhas, long int colunas);

Essa função utiliza um for para percorrer a última coluna e achar o maior elemento logo essa função possui complexidade: max (O(1),O(n)) = O(n).

4.5 – main.c do algoritmo

Analisando o main.c desse algoritmo, podemos destacar que possui um for para preencher a matriz com os valores recebidos do arquivo, outro for para o recebimento do primeiro vetor ótimo na matriz resultado, operações de atribuição e também as funções descritas acima que foram chamadas na main, então a complexidade total da main é max(O(n),O(n),O(n),O(1),O(n²)) = O(n²).

5 - Testes e análises de resultados

Para a resolução de alguns testes, foi utilizada uma base de entrada que gerava matrizes aleatórias e assim, foi verificado o tempo de execução de cada entrada e esses valores serão apresentados em tabelas abaixo.

4 x 4	0,000494s
20 x 20	0,000718s
50 x 50	0,002037s
100 x 100	0,006581s
200 x 200	0,025287s
500 x 500	0,109547s
1000x1000	0,469093s
2000x2000	1,952003s
5000x5000	13,7267s
7500x7500	34,884107s
10000x10000	67,856989s

Matrizes x Tempo

Tabela 1: Testes para matrizes quadradas.

100x100	0,006581s
100x500	0,026025s
100x1000	0,049327s
100x2000	0,090999s
100x5000	0,215529s
100x10000	0,419767s

Matrizes x Tempo

Tabela 2: Testes para linha-fixa e varia coluna.

100x100	0,006581s
500x100	0,026981s
1000x100	0,050450s
2000x100	0,096339s
5000x100	0,237938s
10000x100	0,497241s

Matrizes x Tempo

Tabela 3: Testes para fixa-coluna e varia linha.

E com os resultados obtidos, podemos analisar separadamente cada uma das tabelas por gráficos

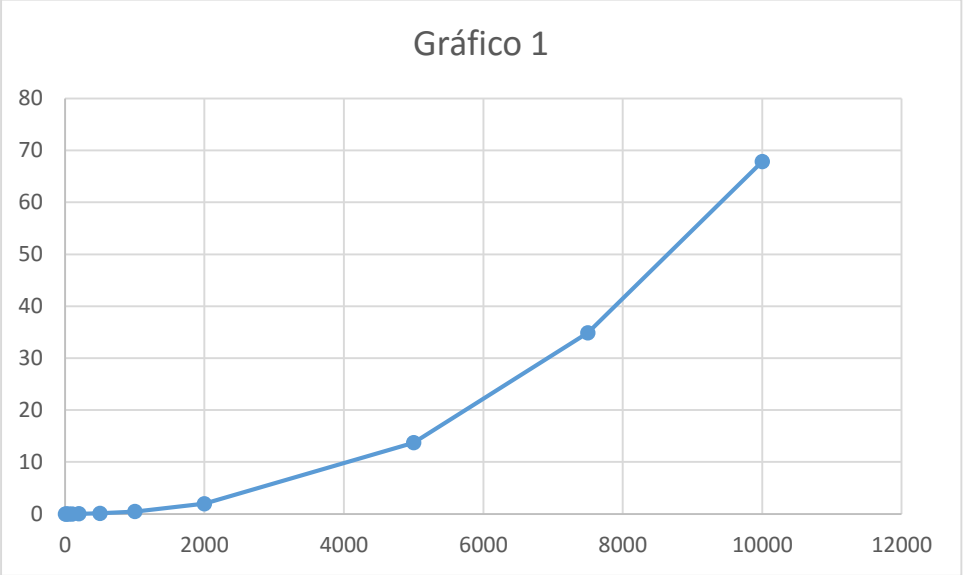

Gráfico1: Gráfico de tempo x matriz

Analisando o gráfico acima, podemos observar um comportamento que tende ao quadrático das matrizes quadradas pelo tempo de execução, no qual pode ser explicado que quando as linhas e as colunas são parecidas os valores tendem a ficar n² por causa da linha x coluna assim explica-se seu comportamento quadrático e podemos verificar que para matrizes pequenas como 4x4, o valor de tempo é irrelevante, mas quando aumentamos o tamanho da matriz para valores como 10000x10000, o tempo de resposta do algoritmo é considerável, beirando os 67s sendo o pior caso da entrada.

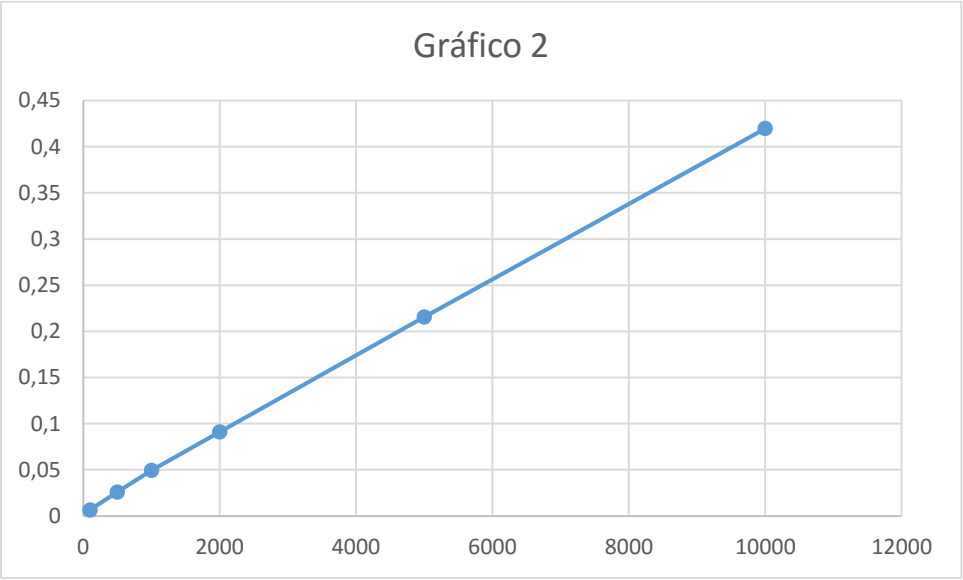

Gráfico2: Gráfico para linha fixa e coluna variando.

Podemos observar, que com as linhas fixas e a variação os valores realizados que foram calculados crescem semi-linearmente de acordo com a entrada, pois um valor ficará fixado e ela só rodará com a coluna no caso, explicando seu comportamento sendo quase linear, e quando maior a entrada, maior será o tempo de execução.

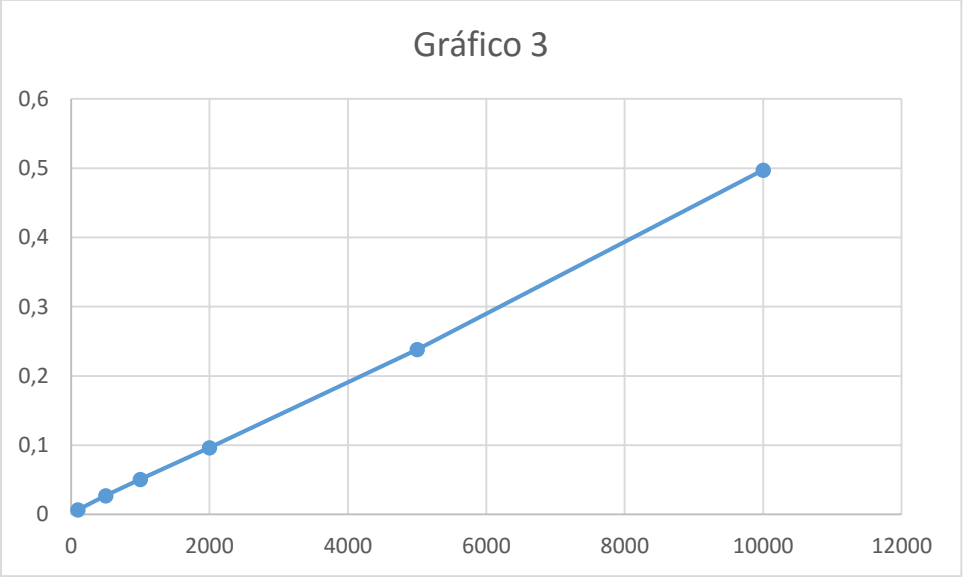

Gráfico 3: Gráfico para coluna fixa e linha variando.

Analisando o gráfico 3, podemos observar o mesmo comportamento do gráfico 2 que segue um comportamento semi-linear também, pois nesse caso as linhas que variam e só rodará para as linhas e as colunas, e para esse caso o algoritmo mesmo com valores

grandes executou em tempo menores que 1s.

6 – Considerações Finais

A implementação do trabalho ocorreu em partes sem muitos problemas, mas alguns tópicos merecem atenção à serem debatidos. O algoritmo de programação dinâmica tem sucesso em seu resultado, porém quando consideramos números elevados de matrizes como 10000 x 10000, o algoritmo tem alguns problemas em relação ao tempo que este seria um pouco alto, mas se o tempo não for algo relevante nesse caso, o algoritmo consegue seu resultado ótimo sem grandes problemas.

7 – Referências bibliográficas

- ROCHA, Leonardo C. D. ***Slides da unidade curricular de Algoritmos e Estruturas de Dados III***, 2016.