Marcos V. Zanardi
Faculdade de Informática – PUCRS

19 de setembro de 2024

Resumo

Este artigo apresenta uma solução para o primeiro desafio da disciplina de Algoritmos e Estrutura de Dados II, do semestre 2024/2. A proposta visa identificar a maior soma de notas ao longo dos caminhos de uma árvore, desde a raiz até uma folha. A eficiência do algoritmo é analisada e discutida, com a apresentação dos resultados obtidos por meio de testes práticos. As considerações finais são fornecidas com base nas análises realizadas.

I – Introdução

O desafio envolve encontrar o caminho com a maior soma de notas em uma árvore, onde as notas são atribuídas às frutas com base em critérios como cor, tamanho, aparência e aroma. Cada caminho na árvore se estende da raiz até uma folha, e o objetivo é determinar o caminho que acumula a maior pontuação possível. A solução proposta utiliza uma abordagem eficiente para percorrer a árvore e calcular a soma das notas em cada caminho. Ao longo do artigo, serão discutidas as técnicas empregadas, os resultados obtidos e as conclusões baseadas na análise de desempenho do algoritmo.

II – Abstração do Problema

A primeira característica relevante da árvore é sua dimensão, expressa em altura e largura, medidas em metros. As bifurcações são representadas pelo caractere "V", as trifurcações por "W" e as folhas (fins dos ramos) por "#". A árvore inicia em um ponto na parte inferior, mas o número exato de elementos não é previamente definido. Um exemplo de representação gráfica da árvore pode ser visto na *Figura 1*.

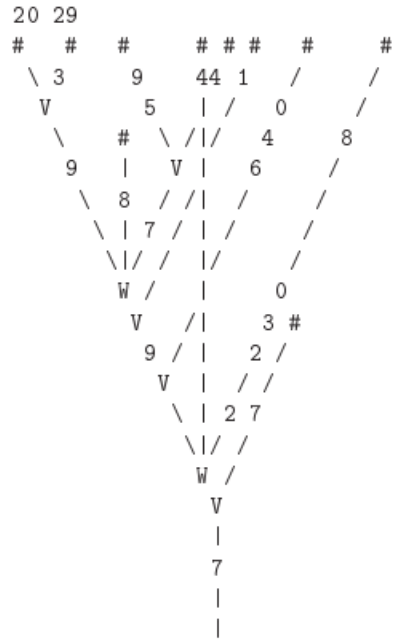


Figura 1: Representação gráfica da árvore do Tópico II, com altura(20), largura(29), bifurcações ["V"], trifurcações ["W"] e folhas ["#"].

O problema pode ser descrito como uma árvore composta por caminhos, nos quais as notas associadas a cada caminho são representadas por N. Esses caminhos são formados por sequências de ramos sucessivos, e a soma das notas desses ramos resulta na nota total do caminho. Vale ressaltar que um mesmo ramo pode pertencer a múltiplos caminhos. Além disso, à medida que um novo ramo é criado, ele herda a soma acumulada dos ramos anteriores, somando essa quantia à sua própria nota. Até o ponto de bifurcação, os ramos compartilham o mesmo valor inicial, uma vez que fazem parte de um único caminho.

III – Abstração Matemática

À medida que o número de bifurcações e trifurcações aumenta, o número de ramos r para cada caminho também cresce de forma linear, o que amplia a quantidade de níveis do caminho em relação à raiz da árvore. Portanto, podemos definir a seguinte função para contabilizar os ramos de um determinado caminho:

$$r(b,t) = b + t\ + 1$$

Explicação:

- Sendo b e t o número de bifurcações e trifurcações respectivamente presentes naquele caminho.
- O termo " + 1 " representa o tronco da arvore, caso tenha;

Por fim, podemos denotar o seguinte somatório para calcular a nota total do respectivo caminho:

$$N = \sum_{i}^{r(b,t)} notai$$

Onde:

- N é a soma das notas ao longo de um caminho.
- $r(b,t)$ representa o número total de ramos.
- $notai$ é a nota associada a cada ramo i.

O objetivo do problema é encontrar o maior valor de N possível entre os caminhos disponíveis, que representa a maior soma de notas em uma sequência de ramos sucessivos.

IV – Estrutura do Problema

Para resolver o problema, mapeamos cada elemento da árvore em uma matriz M, onde o número de linhas corresponde à altura e o número de colunas à largura da árvore. Isso permite acessar diretamente as coordenadas de cada elemento, facilitando o percurso pelos seus ramos.

Utilizamos a técnica de caminhamento *PostOrder*, na qual primeiro visitamos os filhos e, em seguida, o pai. Partimos da raiz e, ao encontrar a primeira bifurcação (seja um "V" ou "W"), percorremos o ramo e seus descendentes até atingir um "#" em todos os caminhos possíveis.

A estrutura do problema foi definida da seguinte maneira:

- **Matriz Árvore**: Responsável por mapear cada elemento da árvore, permitindo o acesso e o percurso de seus elementos por meio de suas coordenadas.

- **`int` `maior`**: Variável utilizada para armazenar a maior nota de um caminho encontrada até o momento ao percorrer os ramos da árvore.

- **`string` `atual`**: Armazena o valor da posição atual durante o percurso pela árvore, podendo ser tanto um caractere quanto um número inteiro, dependendo do tipo de elemento.

- **`Procedimento`** *`PostOrder`*: Processo de percorrer a matriz Tree, utilizado para somar as notas dos ramos ao longo dos caminhos da árvore, partindo da raiz.

- **`bool`** *`percorrido`*: Variável que indica se o ramo já foi totalmente percorrido e processado, evitando repetição de operações.

Essa estrutura permite realizar o percurso da árvore de forma eficiente, calculando as somas e identificando o maior caminho possível.

V – Criação do Algoritmo

Com base nos tópicos anteriores, projetamos o algoritmo para realizar as seguintes tarefas:

1. **Gerar uma Matriz**: Cria uma matriz com as dimensões da árvore (largura e altura), cujos dados são lidos de um arquivo.

2. **Mapear Elementos**: Atribui cada elemento da árvore a uma posição específica na matriz, com base nas informações fornecidas no arquivo.

3. **Localizar a Origem**: Identifica a origem da árvore, localizada na parte inferior da matriz, a partir de onde o percurso será iniciado.

4. **Percorrer a Árvore**: Navega pela árvore utilizando os elementos mapeados, gerando ramos e calculando a soma de suas notas ao longo dos caminhos sucessivos.

5. **Comparar Maior**: Após a finalização de cada ramo, a soma das notas é comparada com o valor armazenado da maior soma encontrada até o momento. Caso a nova soma seja maior, o valor é atualizado.

Esse fluxo garante que o algoritmo identifique corretamente o maior valor de notas em ramos sucessivos na árvore, ou seja, o caminho com maior nota.

VI – Aplicação

Primeiro, precisamos incluir a arvore em uma matriz, para assim percorrermos a arvore, lembrando que valores **`int`** são convertidos para **`string`**:

```
1  procedimento Mapeamento(string Árvore[][]):
2
3     int coluna;//input.file
4     int altura; //input.file
5
6     string elemento_atual;
7
8     para i ← 0 até altura faça:
9           para j ← 0 até coluna faça:
10          // elemento_atual lido de input.file
11          Árvore[i][j] ← elemento_atual
12          fim
13    fim
14 fim
```

***Legenda** : Tarefas do Tópico V: (1) Gerar uma matriz da árvore a partir de um arquivo e (2) Mapear os elementos para suas posições específicas.*

Após mapearmos nossa árvore, podemos procurar nossa raiz, percorrendo nossa última linha até encontrarmos a primeira coluna diferente de vazia. Ao encontrarmos a raiz, identificamos nosso ponto de partida para percorrer a matriz com a coordenada de origem.

```
1  int procedimento Raiz:
2
3     int raiz;
4
5     para j ⟵ 0 até coluna faça:
6           se Árvore[linha][j] /= NULL então
7                 retorna j
8           se não j ⟵ j + 1
9     fim
10 fim
```

***Legenda** : Tarefa do Tópico V: (3) Localizar a origem da árvore*

Em cada iteração do procedimento *PostOrder*, percorremos uma árvore representada por uma matriz, acumulando as notas dos ramos. Ele move-se em direção diagonal esquerda, direita ou diretamente para cima, somando os valores encontrados. Quando encontra um "V", "W" ou "#", o ramo é finalizado, e a função retorna ou continua recursivamente explorando novos caminhos. Esse processo garante que todos os possíveis ramos da árvore sejam percorridos, buscando a maior soma de notas.

```
1  procedimento PostOrder(string Árvore[][], int i, int j, string atual,
2                         string sentido, int val, int maior):
3
4     int ramo ⟵ val
5     bool percorrido ⟵ FALSE
6
7     enquanto percorrido ⟵ FALSE faça:
8
9           se atual ⟵ int então:
10                ramo ⟵ ramo + atual
11          fim
12
13          se sentido ⟵ DiagEsq então:
14                i--, j--
15          se não:
16                se sentido ⟵ DiagDir então:
17                      i--, j++
18                se não:
19                      i--
20                fim
21          fim
22          atual ⟵ Árvore[i][j]
23
24          se atual ⟵ "V" ou atual ⟵ "W" ou atual ⟵ "#" então:
25                percorrido ⟵ TRUE
26          fim
27    fim
28
29    //Altera maior caso valor encontrado seja maior
30    se maior < ramo faça:
31          maior ⟵ ramo
32    fim
33
34    //Se está em uma folha, deve voltar para coordenada de origem
35    se atual ⟵ "#" então:
36          retorna
37    fim
```

```
38
39      //Chamada recursiva:
40      PostOrder(Árvore[][], i, j, atual, DiagEsq, ramo, maior)
41      se atual ⟵ "W" então:
42            PostOrder(Árvore[][], i, j, atual, Cima, ramo, maior)
43      fim
44      PostOrder(Árvore[][], i, j, atual, DiagDir, ramo, maior)
45  fim
```

***Legenda** : Tarefa do Tópico V: (4) Percorrer a árvore e (5) Comparar maior*

Por fim, o procedimento **`PostOrder`** é chamado no procedimento main para percorrer a árvore **`A`** a partir de sua raiz, buscando identificar a maior nota entre os caminhos possíveis. Dessa forma, a estrutura do **`main`** é responsável por inicializar a árvore, invocar **`PostOrder`** para realizar a busca, e, ao final, exibir o maior valor encontrado entre todos os caminhos.

```
1   procedimento Main:
2
3       int coluna;//input.file
4       int altura;//input.file
5       int maior;
6
7       string Árvore[coluna][altura]
8
9       procedimento Mapeamento(Árvore)
10
11      int raiz ⟵ procedimento Raiz;
12
13      atual ⟵ Árvore[linha-1][raiz]
14
15      procedimento PostOrder(Árvore, linha, raiz, atual, A, Cima,
16                             0, maior)
17
18  fim
```

***Legenda** : Aplicação do algoritmo planejado do Tópico V Programa Principal*

VII – Análises e Resultados

Para iniciar a análise, três critérios foram definidos para avaliar o algoritmo: (1) o tempo em função do número de iterações, que mede a eficiência do algoritmo; (2) o tempo em função do número de ramos, analisando como o tamanho da árvore afeta o desempenho; e (3) a relação entre o número de ramos e iterações, que avalia o impacto do tamanho da árvore nas operações necessárias para encontrar a maior soma de um caminho. Os resultados foram obtidos a partir de testes realizados em árvores de oito dimensões: 30x30, 60x60, 90x90, 120x120, 150x150, 180x180, 250x250 e 300x300. A seguir, na Tabela 1, estão apresentados os resultados obtidos a partir dos testes realizados nas diferentes dimensões de árvores mencionadas.

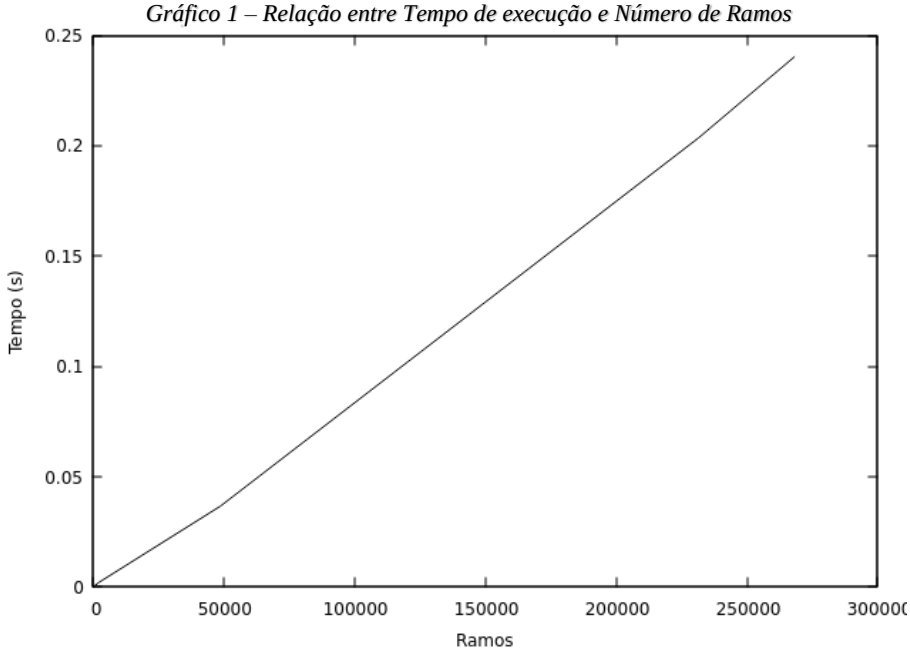
Caso	Maior Nota	Iterações	Ramos	Tempo (segundos)
30x30	28	195	31	0,000033
60x60	62	462	40	0,000059
90x90	116	578	50	0,000073
120x120	148	9320	995	0,000888
150x150	175	21996	2355	0,00263
180x180	232	380990	48804	0,037868
250x250	290	2123885	231645	0,205252
300x300	310	2487464	267986	0,244451

Tabela 1 –Expõe números obtidos através de testes experimentais

Análise I

A primeira análise examina o tempo em função do número de ramos, avaliando como o tamanho da árvore afeta o desempenho do algoritmo. O foco aqui é observar como o aumento na quantidade de ramos impacta o tempo de execução, levando em consideração que o número de operações, como percorrimento e chamadas recursivas, tende a crescer proporcionalmente à complexidade e ao tamanho da árvore. Após realizar os testes, os seguintes resultados foram obtidos e aplicados ao *Gráfico 1*:

Gráfico 1 – Relação entre Tempo de execução e Número de Ramos

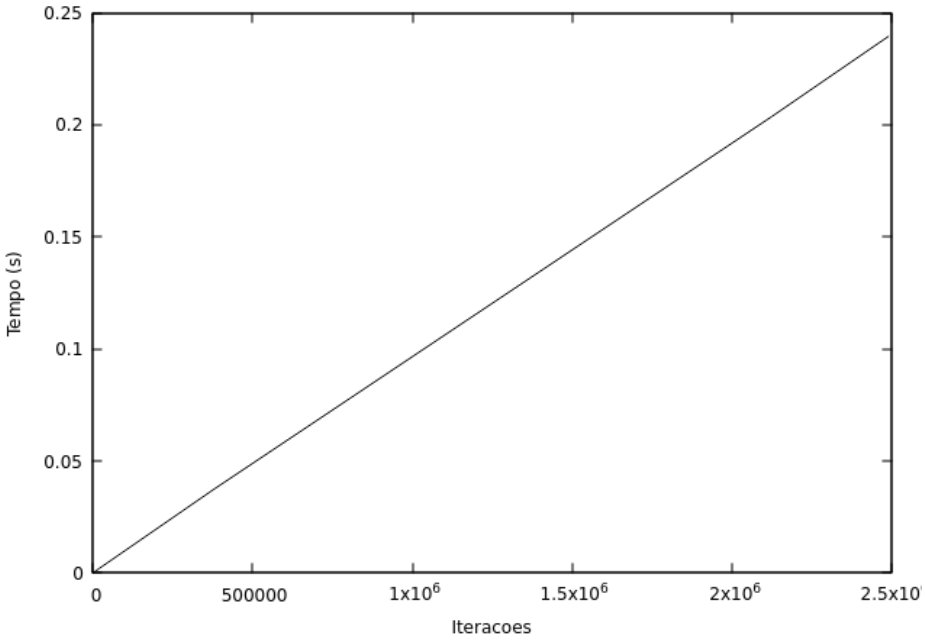


O gráfico revela uma relação linear entre o tempo de execução e o número de ramos da árvore, indicando que o tempo necessário para processar a árvore aumenta de forma proporcional ao número de ramos. Conforme o gráfico demonstra, o tempo de execução sobe de maneira direta à medida que o número de ramos da árvore aumenta. Nesta análise, o tamanho da árvore é avaliado com base no número total de ramos, e os dados mostram que o tempo de execução é consistentemente proporcional ao aumento desse número. Essa constância no comportamento do gráfico sugere que o algoritmo mantém uma eficiência sólida, independentemente do tamanho da árvore, conforme definido pelo número de ramos.

Análise II

A análise do tempo em função do número de iterações avalia a eficiência do algoritmo em três operações principais: criação de ramos, invocação do procedimento *PostOrder* e troca do maior valor. À medida que as iterações aumentam, o impacto dessas operações no tempo total de execução é medido. Para isso, contadores específicos foram implementados para capturar o tempo gasto em cada etapa, permitindo uma compreensão detalhada do efeito de cada operação no desempenho do algoritmo aplicada no *Gráfico 2*

Gráfico 2 – Relação entre Tempo de execução e Número de Iterações

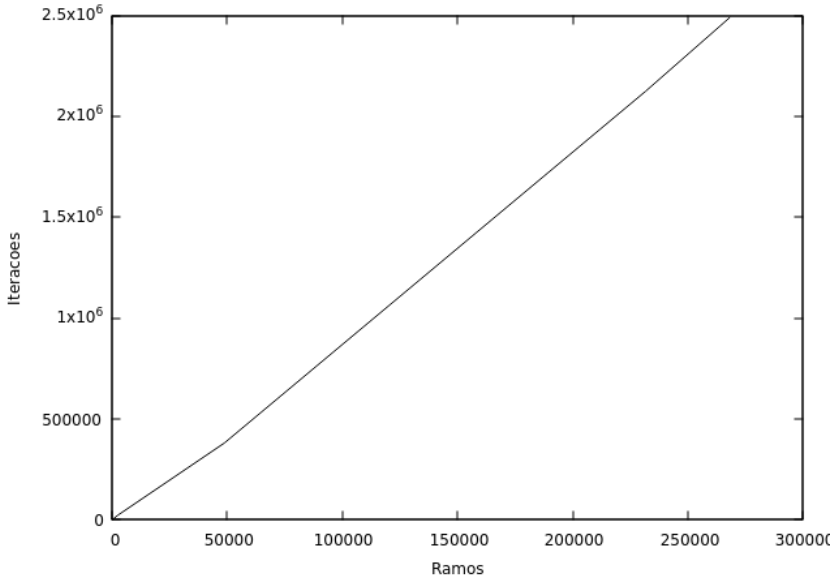


O gráfico do tempo em função do número de iterações revela uma relação linear, onde o tempo de execução aumenta proporcionalmente ao número de iterações realizadas. Cada iteração, que corresponde a uma operação ou chamada ao procedimento PostOrder, tem um custo uniforme, evidenciando a eficiência constante do algoritmo ao lidar com um número crescente de operações. A linearidade sugere que o tempo de execução é diretamente proporcional às iterações, demonstrando consistência em diferentes cenários.

Análise III

A análise da relação entre o número de ramos e o número de iterações investiga como o tamanho da árvore afeta o número de operações necessárias para determinar a maior soma de um caminho. O Gráfico 3 ilustra essa relação, mostrando uma tendência linear entre o número de ramos e o número de iterações realizadas pelo algoritmo.

Gráfico 3 – Relação entre Número de Iterações e Número de Ramos



Conforme o gráfico, o aumento no número de ramos da árvore está associado a um aumento proporcional no número de iterações. Cada ramo adicional contribui para o crescimento no número total de iterações, refletindo a necessidade crescente de operações para processar a árvore. A linearidade observada sugere que o tempo e o esforço necessários para encontrar a maior soma de um caminho aumentam de maneira constante com o crescimento da árvore.

Dado que tanto o tempo de execução em função das iterações quanto a relação entre o número de ramos e as iterações apresentam comportamentos lineares, podemos concluir que a relação entre o número de ramos e o número de iterações também deve ser linear. Essa consistência na relação linear entre diferentes métricas reforça a previsibilidade e a eficiência do algoritmo em lidar com árvores de diferentes tamanhos.

VIII – Conclusão

A análise do tempo em função do número de ramos demonstrou que o tempo de execução aumenta proporcionalmente ao crescimento da árvore. Isso sugere que o tempo de processamento escala de forma linear com o número de ramos, que representa a dimensão da árvore. Por outro lado, a análise do tempo em função do número de iterações revelou que o tempo total de execução também cresce linearmente com o número de iterações realizadas pelo algoritmo. Este comportamento linear reflete a eficiência constante do algoritmo em processar um número crescente de operações.

A relação linear observada em ambos os casos — número de ramos e número de iterações — aponta para uma eficiência consistente e previsível do algoritmo. Segundo Cormen et al. (2009), "se o tempo de execução de um algoritmo é diretamente proporcional ao tamanho da entrada, então a complexidade do algoritmo é considerada linear, ou $O(n)$" ([Cormen, Leiserson, Rivest, & Stein, *Introduction to Algorithms*, 3ª edição, MIT Press, 2009]). Este conceito é corroborado pelos resultados obtidos, que mostram que o tempo de execução cresce de maneira proporcional ao tamanho da entrada, seja pelo número de ramos ou pelo número de iterações necessárias.

Portanto, a complexidade temporal do algoritmo pode ser classificada como $O(n)$, onde n é o número de ramos na árvore. Esta conclusão é sustentada pela linearidade dos gráficos e pela consistência dos resultados obtidos, confirmando que o algoritmo é eficiente e adequado para lidar com diferentes tamanhos de entrada de forma escalável.

"*Algoritmos são, em última análise, expressões matemáticas: eles descrevem a maneira como transformamos um problema matemático em uma solução computacional.*"
— Donald E. Knuth, *The Art of Computer Programming*