Marcos V. Zanardi, Luige L. Figueredo
Engenharia de Computação

Escola Politécnica – 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, ambas medidas em metros. As bifurcações são indicadas pelo caractere "V", as trifurcações por "W" e as folhas (extremidades dos ramos) por "#". A árvore começa em um ponto na base, mas o número exato de elementos não é previamente definido. Um exemplo de sua representação gráfica pode ser visto na *Figura 1*.

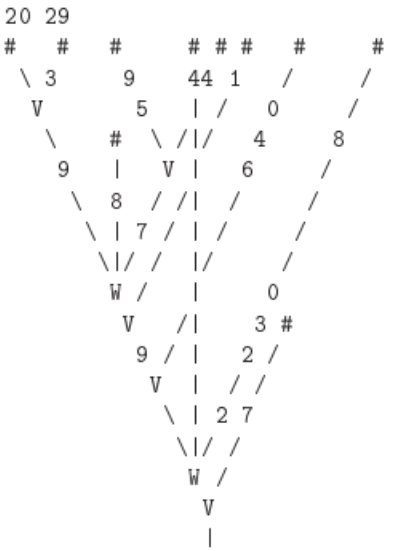


***Figura 1**: Representação gráfica da árvore mencionada no Tópico II, com altura(20), largura(29), bifurcações ["V"], trifurcações ["W"] e folhas ["#"].*

O problema pode ser descrito como a busca por um caminho em uma árvore, onde as notas associadas a cada caminho são representadas por N. Esses caminhos são compostos por sequências de ramos sucessivos, e a soma das notas de cada ramo resulta na nota total do percurso. Importante destacar que um mesmo ramo pode fazer parte de múltiplos caminhos. À medida que um novo ramo é criado, ele herda a soma acumulada dos ramos anteriores, somando esse valor à sua própria

nota. Até o ponto de bifurcação, os ramos compartilham o mesmo valor acumulado, uma vez que pertencem a um único caminho. Em resumo, a árvore é composta por um conjunto de ramos, e o objetivo é identificar o segmento com a maior nota total.

III – Abstração Matemática

Ramos Totais

Como definido no tópico anterior [II - Abstração do problema], interpretamos a árvore como um conjunto de ramos. No entanto, para aplicar a solução do algoritmo, é necessário estabelecer quantos ramos compõem a árvore. Essa definição é essencial para a execução adequada do algoritmo, pois o número de ramos impacta diretamente o cálculo dos caminhos e das notas associadas.

Para cada bifurcação "V", cria-se 2 ramos e para cada "W" 3 ramos, obtendo-se as seguintes funções:

$$b(v) = 2.v \qquad t(w) = 3.w$$

- Sendo v e w o número de bifurcações ("V") e trifurcações ("W") respectivamente presentes em qualquer árvore.

Portanto, podemos definir a função R como o número de ramos totais da árvore da seguinte forma:

$$R\big(b(v), t(w)\big) = b(v) + t(w) + 1$$

- Onde o termo "+ 1", represente o tronco da árvore, caso tenha.

Ramos no Caminho

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. Desse modo, podemos definir a seguinte função r para contabilizar os ramos de um determinado caminho:

$$r(b(v), t(w)) = b + t$$

Explicação:

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)
      int raiz ⟵ procedimento Raiz;
10
11    atual ⟵ Árvore[linha-1][raiz]
12
13    procedimento PostOrder(Árvore, linha, raiz, atual, A, Cima,
14                           0, maior)
15
16 fim
```

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

VII – Análises e Resultados

Para iniciar a análise, foram estabelecidos dois critérios para avaliar o algoritmo: (1) o número de iterações em função do número de ramos, visando analisar o desempenho do algoritmo em relação ao tamanho da árvore, representada por um conjunto de ramos; (2) o tempo de execução em função do número de ramos, verificando como o tamanho da árvore impacta o tempo de processamento. Os resultados foram obtidos a partir de testes realizados em árvores com oito diferentes dimensões: 30x30, 60x60, 90x90, 120x120, 150x150, 180x180, 250x250 e 300x300. A *Tabela 1* apresenta os resultados desses testes, agrupados de acordo com as diferentes dimensões analisadas.

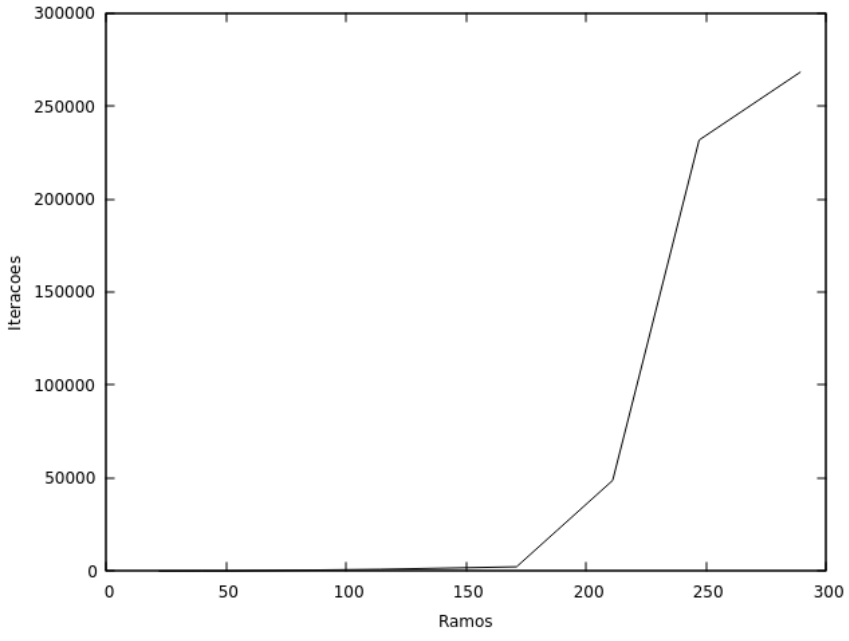
Caso	Maior Nota	Iterações	Bifurcações	Trifurcações	Ramos	Tempo (µs)
30x30	28	195	9	1	22	32
60x60	62	462	12	5	40	60
90x90	116	578	14	7	50	73
120x120	148	9320	40	11	995	920
150x150	175	21996	46	26	2355	2167
180x180	232	380990	57	32	48804	38503
250x250	290	2123885	60	42	231645	205408
300x300	310	2487464	75	46	267986	249055

Tabela 1: Expõe números obtidos através de testes experimentais

Análise I

A primeira análise examina o número de iterações em função do número de ramos, avaliando como o tamanho da árvore afeta o desempenho do algoritmo. O objetivo é observar como o aumento no número de ramos impacta a quantidade de iterações realizadas, considerando que o número de operações – chamadas recursivas –, tende a crescer à medida que a árvore se torna mais complexa. Após a realização dos testes, os resultados foram compilados e aplicados ao *Gráfico 1*.

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

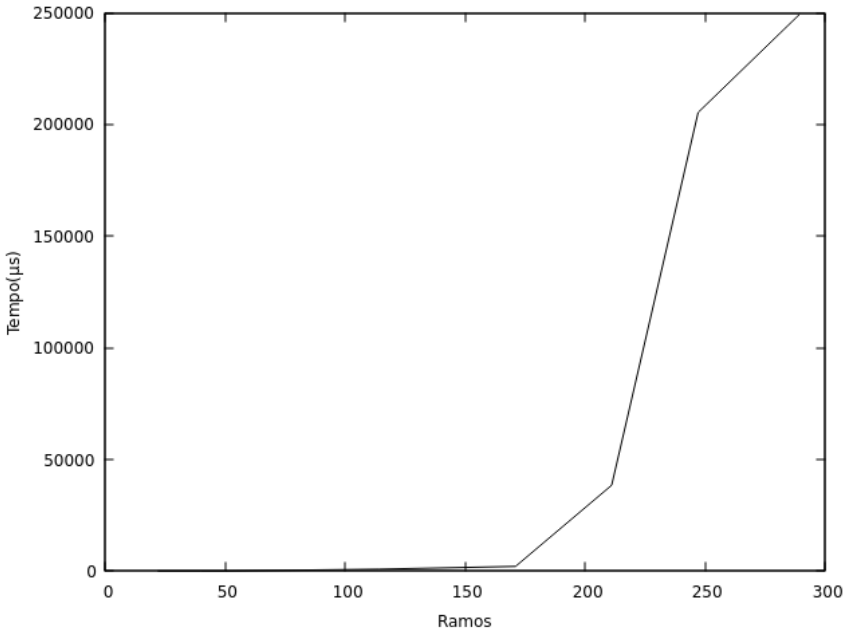


O gráfico mostra que o número de iterações permanece quase constante até cerca de 150 ramos, indicando bom desempenho com árvores menores. No entanto, a partir de 200 ramos, há um aumento exponencial nas iterações, sugerindo que o algoritmo enfrenta maior complexidade à medida que a árvore cresce, especialmente em estruturas com bifurcações e trifurcações. Esse comportamento evidencia uma degradação no desempenho do algoritmo conforme o número de ramos aumenta.

Análise II

A segunda análise examina o tempo de execução em função do número de ramos, avaliando como o tamanho da árvore impacta o tempo necessário para a execução do algoritmo. O foco é entender como o aumento no número de ramos afeta diretamente o tempo de processamento, levando em consideração que a complexidade do algoritmo cresce à medida que o tamanho da árvore aumenta. Após os testes realizados, os resultados foram compilados e apresentados no *Gráfico 2*:

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



O gráfico mostra que o tempo de execução do algoritmo é baixo e constante para árvores com até 150 ramos, indicando eficiência para tamanhos menores. A partir de aproximadamente 180 ramos, o tempo de execução começa a crescer exponencialmente, com aumentos significativos ao redor de 200 e 250 ramos. Esse crescimento exponencial sugere que a complexidade do algoritmo aumenta de forma desproporcional com o número de ramos, tornando o processamento mais demorado à medida que a árvore cresce.

Análise Analítica

Após a *Análise I* e *Análise II*, concluímos que o algoritmo tende a apresentar um comportamento exponencial. Embora a curva do gráfico não revele uma tendência exponencial clara devido ao número limitado de casos, podemos confirmar essa hipótese matematicamente. Ao aplicar uma transformação logarítmica ao eixo y, e se a nova representação se aproximar de uma linha reta em comparação com o gráfico original, isso indicará que o algoritmo realmente possui um crescimento exponencial.

Explicação Matemática – Transformação logarítmica ao eixo y

Quando aplicamos uma transformação logarítmica ao eixo y, uma função exponencial $y = a.b^x$ se transforma em uma linha reta. Isso acontece porque, ao tomar o logaritmo de ambos os lados da equação exponencial, obtemos $\log(y) = \log(a) + x.\log(b)$. Essa equação é linear na forma $\log(y) = C + m.x$, onde C e m são constantes. Portanto, a transformação logarítmica simplifica a análise de crescimento exponencial ao converter a curva em uma linha reta, facilitando a visualização e a interpretação dos dados. [*Montgomery & Runger*, 2010].

Aplicando logaritimo no eixo y sobre os *Gráficos 1* e *Gráficos 2*, obtemos os seguintes resultados nos gráficos *Gráfico 3* e *Gráfico 4*:

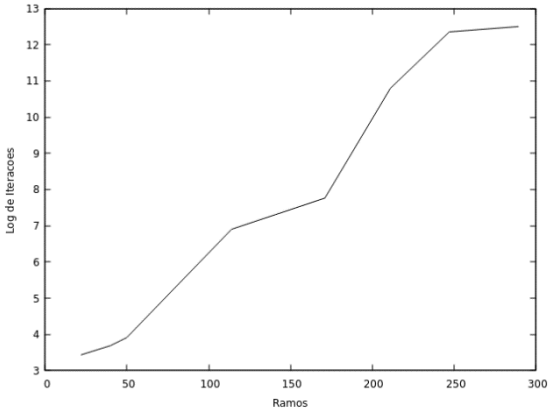


Gráfico 3 *– Logaritmo aplicado sobre eixo y do Gráfico 1*

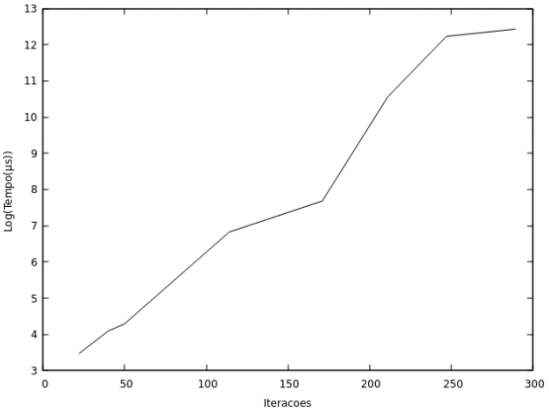


Gráfico 4 *– Logaritmo aplicado sobre eixo y do Gráfico 2*

Em suma, devido ao número limitado de casos, a transformação logarítmica pode não produzir uma linha reta perfeitamente ajustada. No entanto, mesmo com esses poucos dados, é possível observar uma tendência linear em comparação com o gráfico original. Embora a reta resultante possa não ser idealmente precisa, a transformação logarítmica ajuda a evidenciar a relação exponencial subjacente, tornando mais clara a natureza do crescimento exponencial nos dados.

VIII – Conclusão

Após as análises realizadas, concluímos que o algoritmo apresenta comportamento exponencial, com complexidade da ordem de $O(C^n)$, onde C é uma constante que varia conforme a estrutura da árvore, oscilando entre 1 e 3. A constante C assume os seguintes valores:

$C = 1$ para árvores com apenas um tronco (sem bifurcações);

$C = 2$ para árvores formadas exclusivamente por bifurcações (“V”);

$C = 3$ para árvores compostas unicamente por trifurcações (“W”).

Em uma árvore com uma combinação de bifurcações e trifurcações, C é determinado pelo maior fator de ramificação presente na estrutura.

Os gráficos de iterações e tempo indicaram um padrão de crescimento exponencial, corroborando essa hipótese. Para verificá-la, aplicamos uma transformação logarítmica aos dados, e o gráfico resultante apresentou uma tendência linear, característica de um crescimento exponencial quando plotado em escala logarítmica.

Além disso, o uso de recursividade no algoritmo intensifica essa complexidade. A recursividade, especialmente com múltiplas ramificações em cada nó, leva ao aumento exponencial no número de chamadas e operações realizadas à medida que a profundidade da árvore cresce. Conforme estudos indicam (Sedgewick & Wayne, 2011), algoritmos recursivos com várias chamadas tendem a exibir esse tipo de comportamento exponencial.

Portanto, podemos afirmar que a complexidade do algoritmo varia entre $O(1^n)$ no melhor caso e $O(3^n)$ no pior caso. Para árvores gerais com uma mistura de bifurcações e trifurcações, o fator C reflete o maior grau de ramificação, resultando no crescimento exponencial observado no número de chamadas recursivas e nas operações realizadas.

"*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*