Assignment 2: Distância de edição de árvore

Camila Lopes, Felipe Campolina, Henrique Diniz, Leandro Guido, Marcelo Augusto, Victor Colen

Instituto de Ciências Exatas e Informática – Pontifícia Universidade Católica de Minas Gerais
{camila.lopes, felipe.campolina, henrique.diniz, leandro.guido, marcelo.augusto, victor.costa}@pucminas.br

[1]Pontifícia Universidade Católica de Minas Gerais (PUC-MG)

Abstract. *This article explores the edit distance between trees, an essential metric for quantifying structural similarity between two trees. We analyze three main algorithms: the naive recursive algorithm, Tai's algorithm, and the Zhang-Shasha algorithm. Detailed implementations and empirical tests are presented to evaluate the efficacy and efficiency of each approach. The results show that, while the naive recursive algorithm has exponential growth in execution time, the Tai and Zhang-Shasha algorithms perform significantly better, making them more suitable for practical applications. Applications in bioinformatics, computational linguistics, and image analysis are discussed.*

Resumo. *Este artigo explora a distância de edição entre árvores, uma métrica essencial para quantificar a similaridade estrutural entre duas árvores. Analisamos três algoritmos principais: o algoritmo recursivo ingênuo, o algoritmo de Tai e o algoritmo de Zhang-Shasha. Implementações detalhadas e testes empíricos são apresentados para avaliar a eficácia e a eficiência de cada abordagem. Os resultados mostram que, enquanto o algoritmo recursivo ingênuo tem crescimento exponencial no tempo de execução, os algoritmos de Tai e Zhang-Shasha apresentam desempenho significativamente melhor, sendo mais adequados para aplicações práticas. Aplicações em bioinformática, linguística computacional e análise de imagens são discutidas.*

1. Introdução

A distância de edição entre árvores é uma medida que quantifica o mínimo custo de operações necessárias para transformar uma árvore $T1$ em outra árvore $T2$. As operações permitidas geralmente incluem inserções, deleções e substituições de nós. Este problema tem aplicações significativas em áreas como bioinformática, onde é usado para comparar estruturas de RNA, e em linguística computacional, onde auxilia na correção de erros sintáticos.

O algoritmo recursivo ingênuo é a abordagem mais simples e direta para calcular a distância de edição, utilizando chamadas recursivas para explorar todas as possibilidades de edição. No entanto, devido à sua complexidade exponencial, este método é computacionalmente inviável para árvores grandes. Esta abordagem tem uma complexidade temporal de $O(3^n)$, onde n é o número total de nós nas árvores, tornando-se impraticável para árvores de tamanhos moderados a grandes.

Tai [Tai 1979] propôs um algoritmo eficiente que resolve o problema em tempo polinomial $O(V \cdot V' \cdot L^2 \cdot L'^2)$, onde V e V' são os números de nós nas árvores $T1$ e $T2$, respectivamente, e L e L' são as profundidades máximas das árvores $T1$ e $T2$. Este algoritmo utiliza

uma técnica de mapeamento que transforma uma árvore em outra por meio de uma série de operações de edição, minimizando o custo total.

Zhang e Shasha [Zhang and Shasha 1989] apresentaram um algoritmo com complexidade $O(\min\{n_{\text{height}}, n_{\text{leaves}}\} \cdot \min\{m_{\text{height}}, m_{\text{leaves}}\} \cdot nm)$, onde n_{height} e n_{leaves} são a altura e o número de folhas em $T1$, e m_{height} e m_{leaves} são a altura e o número de folhas em $T2$. Se as árvores forem balanceadas, a complexidade se torna $O((n \log n)(m \log m))$, onde n e m são o número de nós em $T1$ e $T2$, respectivamente. Este método é amplamente reconhecido por sua eficiência e simplicidade na implementação.

Neste trabalho, as abordagens mencionadas são implementadas e testadas empiricamente para avaliar seu desempenho em termos de complexidade temporal e espacial. Os resultados dos testes comparativos são apresentados nas seções seguintes, onde cada algoritmo é detalhado e suas complexidades são discutidas.

2. Responsabilidades do Grupo

Tabela 1. Responsabilidades dos membros do grupo

Membro do Grupo	Responsabilidades
Camila Lopes	Implementação do algoritmo recursivo ingênuo e redação das seções de introdução e conclusão.
Felipe Campolina	Implementação do algoritmo de Tai e elaboração dos testes empíricos.
Henrique Diniz	Implementação do algoritmo de Tai e análise de resultados.
Leandro Guido	Implementação do algoritmo de Zhang-Shasha e revisão bibliográfica.
Marcelo Augusto	Implementação do algoritmo de Zhang-Shasha e geração de dados de teste.
Victor Colen	Coordenação geral do projeto e revisão final do texto.

3. Princípios e Conceitos Fundamentais

Nesta seção, são apresentados os princípios e conceitos fundamentais necessários para a compreensão da distância de edição entre árvores. Esses conceitos incluem árvores, distância de edição, operações de edição, e aplicações práticas.

3.1. Árvores

Uma árvore é uma estrutura de dados hierárquica composta por nós (ou vértices) conectados por arestas (ou ramos). A árvore possui as seguintes propriedades:

- **Raiz:** O nó principal da árvore, de onde todos os outros nós descendem.
- **Nós Internos:** Nós que possuem pelo menos um filho.
- **Folhas:** Nós que não possuem filhos.
- **Subárvore:** Qualquer nó e todos os seus descendentes formam uma subárvore.

Cada nó em uma árvore pode ter zero ou mais filhos, e a árvore não contém ciclos, ou seja, não há caminho que comece e termine no mesmo nó [Cormen et al. 2009].

Formalmente, uma árvore T é um grafo acíclico conectado, definido como um par ordenado $T = (V, E)$, onde V é o conjunto de vértices (nós) e E é o conjunto de arestas (ramos) [Knuth 1997].

3.2. Distância de Edição

A distância de edição entre duas árvores T_1 e T_2 é uma medida que quantifica o custo mínimo necessário para transformar T_1 em T_2 por meio de uma série de operações de edição. Esta métrica é utilizada para comparar a similaridade estrutural entre duas árvores [Zhang and Shasha 1989].

3.3. Operações de Edição

As operações de edição permitem transformar uma árvore em outra e são definidas como:

- **Inserção de Nó:** Adicionar um novo nó em uma posição específica da árvore.
- **Deleção de Nó:** Remover um nó existente e conectar seus filhos diretamente ao nó pai.
- **Substituição de Nó:** Alterar o rótulo ou valor de um nó existente.

Cada uma dessas operações tem um custo associado, e o objetivo é encontrar a sequência de operações com o menor custo total para transformar uma árvore na outra [Tai 1979].

3.4. Aplicações Práticas

A distância de edição entre árvores possui diversas aplicações práticas em diferentes campos:

- **Bioinformática:** Utilizada na comparação de estruturas de RNA, onde as árvores representam a estrutura secundária das moléculas de RNA [Shapiro and Zhang 1988].
- **Linguística Computacional:** Auxilia na correção de erros sintáticos em árvores de análise gramatical [Joshi 1987].
- **Análise de Imagens:** Empregada na comparação de hierarquias de partições em imagens para detectar semelhanças e diferenças estruturais [Martin et al. 2001].

Essas aplicações demonstram a relevância da distância de edição entre árvores na resolução de problemas complexos em diversas áreas científicas e tecnológicas.

3.5. Princípio dos Algoritmos

Os algoritmos para calcular a distância de edição entre árvores são baseados em princípios fundamentais que guiam o processo de transformação de uma árvore T_1 em outra árvore T_2. A seguir, são discutidos os principais princípios que fundamentam esses algoritmos.

3.5.1. Abordagem Recursiva

O princípio básico por trás dos algoritmos de distância de edição é a decomposição recursiva do problema. A abordagem recursiva divide o problema original em subproblemas menores, calculando a distância de edição para subárvores de T_1 e T_2. Essa abordagem permite explorar todas as possíveis operações de edição, embora possa ser computacionalmente intensiva [Zhang and Shasha 1989].

3.5.2. Programação Dinâmica

Para superar a ineficiência da abordagem recursiva ingênua, utiliza-se a programação dinâmica. Este princípio envolve o armazenamento de resultados de subproblemas em uma tabela (ou matriz), evitando cálculos redundantes. A programação dinâmica permite construir a solução ótima para o problema original a partir das soluções ótimas dos subproblemas [Cormen et al. 2009].

3.5.3. Memoization

Memoization é uma técnica utilizada em programação dinâmica para otimizar algoritmos recursivos. Consiste em armazenar os resultados de subproblemas já calculados, de forma que, quando esses subproblemas forem novamente encontrados, seus resultados possam ser diretamente reutilizados sem a necessidade de novo cálculo.

A importância do memoization é evidente na redução do tempo de execução de algoritmos recursivos. Sem essa técnica, um algoritmo pode recalcular a mesma solução várias vezes, resultando em uma complexidade exponencial. Com o memoization, cada subproblema é resolvido apenas uma vez, reduzindo a complexidade para um tempo polinomial [Ahuja et al. 1993].

4. Funcionamento dos algorítimos

Nessa seção, o artigo apresenta o funcionamento básico de cada um dos algoritmos implemnetados.

4.1. Recursivo Ingênuo para Distância de Edição entre Árvores

O algoritmo de recursão ingênua para calcular a distância de edição entre duas árvores é inspirado na abordagem de distância de edição entre strings. A ideia é processar os nós das árvores um a um, começando pelas folhas até a raiz. Existem duas possibilidades para cada par de nós sendo processados: ou eles correspondem ou não correspondem. Se os últimos nós de ambas as árvores corresponderem, não há necessidade de realizar nenhuma operação. Caso contrário, podemos realizar todas as três operações de edição: inserção, substituição e remoção. Calculamos recursivamente o resultado para o restante das subárvores e, ao final, selecionamos a menor resposta.

A seguir, apresentamos a árvore recursiva para este problema:

4.1.1. Relações de Recorrência para a Distância de Edição

A função `edit_distance` calcula a distância de edição entre duas strings, `str1` e `str2`, usando uma abordagem recursiva. Os parâmetros de entrada são as strings `str1` e `str2`, e os índices `M` e `N` que representam as posições atuais nas strings.

Parâmetros de Entrada

- `str1`: Primeira string
- `str2`: Segunda string
- `M`: Índice atual na string `str1`
- `N`: Índice atual na string `str2`

$$\text{edit\_distance} = \begin{cases} \text{edit\_distance}(str1, str2, M-1, N-1) & \text{se } str1[M] = str2[N] \\ 1 + \min \begin{cases} \text{edit\_distance}(str1, str2, M-1, N-1), \\ \text{edit\_distance}(str1, str2, M, N-1), \\ \text{edit\_distance}(str1, str2, M-1, N) \end{cases} & \text{caso contrário} \end{cases}$$

4.1.2. Casos Base para a Distância de Edição

- Quando $T1$ se torna vazio, ou seja, $M = 0$, retornamos N, pois são necessários N nós para converter uma árvore vazia em $T2$.
- Quando $T2$ se torna vazio, ou seja, $N = 0$, retornamos M, pois são necessários M nós para converter uma árvore vazia em $T1$.

4.1.3. Passo a Passo do Algoritmo Recursivo Ingênuo

O algoritmo recursivo ingênuo para calcular a distância de edição entre duas árvores segue os seguintes passos:

1. **Verificar se uma das árvores está vazia**: Se uma das árvores estiver vazia, a distância de edição será igual ao número de nós da outra árvore, pois é necessário inserir todos os nós da árvore não vazia na árvore vazia.
2. **Comparar os últimos nós das duas árvores**: Se os últimos nós das duas árvores forem iguais, a distância de edição para esses nós será a mesma que a distância de edição das subárvores restantes.
3. **Calcular a distância de edição para os três casos possíveis quando os últimos nós são diferentes**:
    - Inserir: Calcular a distância de edição após inserir o nó no final da segunda árvore.
    - Remover: Calcular a distância de edição após remover o nó do final da primeira árvore.
    - Substituir: Calcular a distância de edição após substituir o nó do final da primeira árvore pelo nó do final da segunda árvore.
4. **Retornar a menor distância de edição**: Selecionar a menor distância de edição entre as três opções calculadas no passo anterior.

Abaixo está o pseudo-código do algoritmo:

Algorithm 1 Distância de Edição Recursiva Ingênua

Input: Árvores $T1$, $T2$ e tamanhos m, n
Output: Distância de edição entre $T1$ e $T2$
Function `editDist` $(T1, T2, m, n)$ **:**

if $m == 0$ **then**
  return n
if $n == 0$ **then**
  return m
if $T1[m-1] == T2[n-1]$ **then**
  return `editDist` $(T1, T2, m-1, n-1)$
return $1 + \min$ { `editDist` (*T1, T2, m, n-1*) , `editDist` (*T1, T2, m-1, n*) , `editDist` (*T1, T2, m-1, n-1*)

4.1.4. Complexidade de Tempo e Espaço

Complexidade de Tempo A complexidade de tempo do algoritmo recursivo ingênuo é exponencial. Para cada par de nós comparados, o algoritmo faz três chamadas recursivas, resultando em uma árvore de chamadas de profundidade $m + n$. Portanto, a complexidade de tempo é:

$$O(3^{\max(m,n)})$$

onde m é o número de nós na árvore $T1$ e n é o número de nós na árvore $T2$. Isso ocorre porque em cada passo, o algoritmo explora três possíveis caminhos (inserir, remover, substituir).

Complexidade de Espaço A complexidade de espaço do algoritmo é determinada pela profundidade da pilha de chamadas recursivas. No pior caso, a profundidade da recursão é igual ao número máximo de nós em uma das árvores. Portanto, a complexidade de espaço é:

$$O(\max(m, n))$$

No entanto, como o algoritmo utiliza apenas um número constante de variáveis locais e não aloca espaço adicional além da pilha de chamadas, a complexidade espacial é linear em relação ao comprimento das árvores.

4.2. Algoritmo de Tai para Distância de Edição entre Árvores

O problema de correção de árvore para árvore é determinar, para duas árvores ordenadas e rotuladas T e T', a distância de T para T' medida pela sequência de operações de edição de custo mínimo necessárias para transformar T em T'. As operações de edição investigadas permitem mudar um nó de uma árvore para outro nó, deletar um nó de uma árvore ou inserir um nó em uma árvore. Um algoritmo é apresentado que resolve este problema em tempo polinomial.

4.2.1. Relações de Recorrência para a Correção de Árvores

$$\text{edit\_distance}(T, T', M, N) = \begin{cases} \text{edit\_distance}(T, T', M-1, N-1) & \text{se os nós correspondem} \\ 1 + \min \begin{cases} \text{edit\_distance}(T, T', M-1, N-1), \\ \text{edit\_distance}(T, T', M, N-1), \\ \text{edit\_distance}(T, T', M-1, N) \end{cases} & \text{se os nós não correspondem} \end{cases}$$

4.2.2. Casos Base para a Correção de Árvores

- Quando T se torna vazio, ou seja, $M = 0$, retornamos N, pois são necessários N nós para converter uma árvore vazia em T'.
- Quando T' se torna vazio, ou seja, $N = 0$, retornamos M, pois são necessários M nós para converter uma árvore vazia em T.

4.2.3. Passo a Passo do Algoritmo de Tai

1. **Calcular** $E[s : u : i, t : v : j]$.
   O primeiro passo do algoritmo é calcular $E[s : u : i, t : v : j]$ para todas as combinações possíveis de s, u, t, v, j. Essa etapa envolve determinar o custo mínimo para transformar a subárvore de T enraizada

Algorithm 2 Algoritmo para Calcular $E[s : u : i, t : v : j]$

Data: Definir $f^n(x) = f(f^{n-1}(x))$ para $n \geq 1$ e $x > 1$, onde $f(x)$ é o pai do nó x
Result: Algoritmo para o passo (1)

for $i = 1, 2, \ldots, |T|$ **do**
  for $j = 1, 2, \ldots, |T'|$ **do**
    for $u = f(i), f^2(i), \ldots, 1$ **do**
      for $s = f(u), f^2(u), \ldots, 1$ **do**
        for $v = f(j), f^2(j), \ldots, 1$ **do**
          for $t = f(v), f^2(v), \ldots, 1$ **do**
            if $s = u = i$ *e* $t = v = j$ **then**
              $E[s : u : i, t : v : j] = r(T[i] \to T'[j])$
            else if $s = u = i$ *e* $t < v = j$ **then**
              $E[s : u : i, t : v : j] = E[s : u : i, t : f(j) : j - 1] + r(A \to T'[j])$
            else if $s < u = i$ *e* $t = v = j$ **then**
              $E[s : u : i, t : v : j] = E[s : f(i) : i - 1, t : v : j] + r(T[i] \to A)$
            else
              E[s:u:i, t:v:j] = $\min \begin{cases} E[s : x : i, t : v : j] \\ E[s : u : i, t : y : j] \\ E[s : u : x - 1, t : v : y - 1] + E[x : x : i, y : y : j] \end{cases}$
              _

T[x] é o filho de T[u] no caminho de T[u] a T[i], e T'[y] é o filho de T'[v] no caminho de T[v] a T'[j]

Neste algoritmo:

- Examina-se todos os nós i na árvore T e todos os nós j na árvore T'.
- Para cada nó i na árvore T, olha-se todos os "pais"e "avós"u desse nó.
- Para cada "pai"e "avô"u, olha-se todos os "bisavôs"s.
- Para cada nó j na árvore T', olha-se todos os "pais"e "avós"v desse nó.
- Para cada "pai"e "avô"v, olha-se todos os "bisavôs"t.

As condições de transformação são verificadas e os custos mínimos são atualizados de acordo. Este cálculo determina o custo mínimo de transformar uma subárvore de T em uma subárvore correspondente de T'.

2. **Calcular MIN_M(i, j)** para todos i, j, onde $1 \leq i \leq |T|$ e $1 \leq j \leq |T'|$.

Algorithm 3 Algoritmo para Calcular MIN_M(i, j)

MIN_M$(1, 1) = 0$
for $i = 2, 3, \ldots, |T|$ **do**
  for $j = 2, 3, \ldots, |T'|$ **do**
    MIN_M$(i, j) \leftarrow$ INFINITE
    for $s = f(i), f^2(i), \ldots, 1$ **do**
      for $t = f(j), f^2(j), \ldots, 1$ **do**
        temp $\leftarrow$ MIN_M(s, t) + $E[s : f(i) : i - 1, t : f(j) : j - 1] - r(T[s] \rightarrow T'[t])$
        MIN_M$(i, j) \leftarrow \min(temp, MIN_M(\text{i, j}))$
    MIN_M(i, j) $\leftarrow$ MIN_M(i, j) + r(T[i] $\rightarrow T'[j]$)

Neste algoritmo:
- Examina-se todos os pares de nós i na árvore T e j na árvore T'.
- Inicializa-se a transformação de $1, 1$ como zero.
- Para cada nó i na árvore T:
- Para cada nó j na árvore T':
- Define-se MIN_M(i, j) como infinito no início.
- Para cada "pai"e "avô"s do nó i:
- Para cada "pai"e "avô"t do nó j:
  - Calcula-se um valor temporário com base nas transformações anteriores.
  - Atualiza-se MIN_M(i, j) com o menor valor encontrado.
- Adiciona-se o custo de transformar $T[i]$ em $T'[j]$ a MIN_M(i, j).

3. **Calcular** $D(i, j)$ para todos i, j, onde $1 \leq i \leq |T|$ e $1 \leq j \leq |T'|$.

Algorithm 4 Algoritmo para Calcular $D(i, j)$

$D(1, 1) \leftarrow 0$
for $i = 2, 3, \ldots, |T|$ **do**
  $D(i, 1) \leftarrow D(i - 1, 1) + r(T[i] \rightarrow A)$
for $j = 2, 3, \ldots, |T'|$ **do**
  $D(1, j) \leftarrow D(1, j - 1) + r(A \rightarrow T'[j])$
for $i = 2, 3, \ldots, |T|$ **do**
  for $j = 2, 3, \ldots, |T'|$ **do**

$$D(i, j) \leftarrow \min \begin{cases} D(i, j - 1) + r(A \rightarrow T'[j]), \\ D(i - 1, j) + r(T[i] \rightarrow A), \\ \text{MIN\_M}(i, j) \end{cases}$$

Nesse algorítimo :

- Para cada nó j na árvore T':
- Calcular $D(i, j)$ considerando três possíveis opções
- Adicionar o custo de transformar A (um nó vazio) no nó $T'[j]$ ao custo de transformar a parte anterior da árvore T até o nó $j - 1$.
- Adicionar o custo de transformar o nó $T[i]$ em A ao custo de transformar a parte anterior da árvore T' até o nó $i - 1$.
- Utilizar o menor número de mudanças calculado previamente (MIN_M(i, j)).

4.2.4. Complexidade de Tempo e Espaço

Complexidade de Tempo A complexidade de tempo do algoritmo de Tai é polinomial. O algoritmo opera em $O(V \cdot V' \cdot L^2 \cdot L'^2)$, onde V e V' são os números de nós em T e T', respectivamente, e L e L' são as profundidades máximas de T e T', respectivamente.

Complexidade de Espaço A complexidade de espaço do algoritmo é determinada pela profundidade da pilha de chamadas recursivas e pelo armazenamento necessário para manter os resultados intermediários. Portanto, a complexidade de espaço é: O$(\max(V \cdot L, V' \cdot L'))$

No entanto, como o algoritmo utiliza estruturas de dados adicionais para armazenar os resultados intermediários, a complexidade espacial é polinomial em relação ao número de nós e à profundidade das árvores.

4.3. Algoritmo de Zhang-Shasha para Distância de Edição entre Árvores

O algoritmo de Zhang-Shasha para calcular a distância de edição entre duas árvores ordenadas e rotuladas é um método eficiente que utiliza programação dinâmica. Ele é projetado para responder a várias questões, como a distância entre duas árvores e a distância mínima quando subárvores podem ser removidas. O algoritmo consiste em três principais operações de edição: inserção, exclusão e modificação de nós.

4.3.1. Relações de Recorrência

As relações de recorrência para calcular a distância de edição entre subflorestas são definidas da seguinte forma:

Parâmetros de entrada:

- $T1[i_1..i]$: Subárvore da floresta 1.
- $T2[j_1..j]$: Subárvore da floresta 2.

$$\text{forestdist}() = \min \begin{cases} \text{forestdist}(T1[l(i_1)..i-1], T2[j_1..j]) + \gamma(T1[i] \to \Lambda) \\ \text{forestdist}(T1[i_1..i], T2[l(j_1)..j-1]) + \gamma(\Lambda \to T2[j]) \\ \text{forestdist}(T1[l(i_1)..i-1], T2[l(j_1)..j-1]) + \gamma(T1[i] \to T2[j]) \end{cases}$$

Onde: - $\gamma(T1[i] \to T2[j])$ é o custo de editar o nó $T1[i]$ para o nó $T2[j]$. - Λ representa a operação de inserção ou exclusão de nós.

A explicação dos casos acima é a seguinte:

- 1. forestdist$(T1[l(i_1)..i-1], T2[j_1..j]) + \gamma(T1[i] \to \Lambda)$: se $T1[i]$ é excluído.
- 2. forestdist$(T1[i_1..i], T2[l(j_1)..j-1]) + \gamma(\Lambda \to T2[j])$: se $T2[j]$ é inserido.
- 3. forestdist$(T1[l(i_1)..i-1], T2[l(j_1)..j-1]) + \gamma(T1[i] \to T2[j])$: se $T1[i]$ é editado para $T2[j]$.

4.3.2. Casos base

Os casos base para a distância de edição são:

$$\text{forestdist}(\Lambda, T2[j_1..j]) = \sum_{k=j_1}^{j} \gamma(\Lambda \to T2[k])$$

$$\text{forestdist}(T1[i_1..i], \Lambda) = \sum_{k=i_1}^{i} \gamma(T1[k] \to \Lambda)$$

A explicação dos casos base é a seguinte: - Quando a primeira subfloresta é vazia (Λ) e a segunda subfloresta é $T2[j_1..j]$, a distância de edição é a soma dos custos de inserção de todos os nós da segunda subfloresta. - Quando a segunda subfloresta é vazia (Λ) e a primeira subfloresta é $T1[i_1..i]$, a distância de edição é a soma dos custos de exclusão de todos os nós da primeira subfloresta.

4.3.3. Pseudo-código do Algoritmo

Algorithm 5 Algoritmo de Zhang-Shasha para Distância de Edição entre Árvores

Input: Árvores $T1$, $T2$
Output: Distância de edição entre $T1$ e $T2$
Function `forestdist(`$T1, T2$`)`**:**
  foreach i *em ordem pós-ordem de* $T1$ **do**
    foreach j *em ordem pós-ordem de* $T2$ **do**
      if $T1[i]$ *é folha* **then**
        $D[i, 0] \leftarrow i$
      end
      if $T2[j]$ *é folha* **then**
        $D[0, j] \leftarrow j$
      end

$$D[i,j] \leftarrow \min \begin{cases} \text{Forestdist}(T1[l(i)..i-1], T2[j_1..j]) + \gamma(T1[i] \to \Lambda), \\ \text{Forestdist}(T1[i_1..i], T2[l(j_1)..j-1]) + \gamma(\Lambda \to T2[j]), \\ \text{Forestdist}(T1[l(i_1)..i-1], T2[l(j_1)..j-1]) + \gamma(T1[i] \to T2[j]) \end{cases}$$

    end
  end
  return $D[|T1|, |T2|]$

4.3.4. Explicação do Pseudo-código Passo a Passo

1. **Entrada e Saída**:

- **Entrada**: Duas árvores ordenadas $T1$ e $T2$.
- **Saída**: A distância de edição entre as duas árvores, ou seja, o número mínimo de operações (inserção, exclusão, substituição) necessárias para transformar $T1$ em $T2$.

2. **Função** `forestdist`:
   - Define a função recursiva `forestdist`, que calcula a distância de edição entre subflorestas das árvores $T1$ e $T2$.
3. **Iteração sobre as Subflorestas**:
   - Para cada nó i em $T1$ na ordem pós-ordem (processa-se as folhas antes dos nós raiz):
   - Para cada nó j em $T2$ na ordem pós-ordem:
     - Se $T1[i]$ é uma folha, então a distância de edição para transformar uma subfloresta de $T1$ contendo somente i em uma árvore vazia é i (i.e., todas as i exclusões de nós de $T1$).
     - Se $T2[j]$ é uma folha, então a distância de edição para transformar uma árvore vazia em uma subfloresta de $T2$ contendo somente j é j (i.e., todas as j inserções de nós de $T2$).
4. **Cálculo da Distância de Edição**:
   - Para nós internos, a distância de edição $D[i, j]$ é calculada como o mínimo de três casos:
     (a) Exclusão de $T1[i]$: $\texttt{forestdist}(T1[l(i)..i-1], T2[j_1..j]) + \gamma(T1[i] \to \Lambda)$
     (b) Inserção de $T2[j]$: $\texttt{forestdist}(T1[i_1..i], T2[l(j_1)..j-1]) + \gamma(\Lambda \to T2[j])$
     (c) Substituição de $T1[i]$ por $T2[j]$: $\texttt{forestdist}(T1[l(i_1)..i-1], T2[l(j_1)..j-1]) + \gamma(T1[i] \to T2[j])$
5. **Retorno do Resultado**:
   - Retorna a distância de edição final $D[|T1|, |T2|]$, que é a distância de edição entre as árvores completas $T1$ e $T2$.

4.3.5. Complexidade de Tempo e Espaço

Complexidade de Tempo A complexidade de tempo do algoritmo de Zhang-Shasha é $O(|T1| \times |T2| \times \min(\text{depth}(T1), \text{leaves}(T1)) \times \min(\text{depth}(T2), \text{leaves}(T2)))$, onde $|T1|$ e $|T2|$ são o número de nós em $T1$ e $T2$, respectivamente. Isso se deve ao cálculo iterativo das distâncias para cada par de subflorestas.

Complexidade de Espaço A complexidade de espaço é $O(|T1| \times |T2|)$, pois o algoritmo armazena as distâncias em uma matriz de tamanho proporcional ao produto dos tamanhos das duas árvores.

5. Implementação e Testes

Nesta seção, apresentamos três implementações distintas para o cálculo da distância de edição entre árvores, utilizando C++ como linguagem de programação. Sendo a primeira uma adaptação do primeiro código apresentado em [GeeksforGeeks 2023], a segunda é uma replicação do algorítimo apresentado no artigo de [Tai 1979] e por fim, a última é baseado no artigo de [Zhang and Shasha 1989].

5.1. Decisões de Implementação

Este estudo avalia a eficácia e a eficiência dos algoritmos de distância de edição entre árvores através de uma metodologia que envolve a geração de árvores a partir de strings aleatórias. A seguir, são descritas as etapas e decisões implementadas para os testes.

5.1.1. Geração de Strings Aleatórias

A geração de strings aleatórias constitui a primeira etapa do processo de teste. As strings são geradas com comprimentos pré-definidos variando de 200 a 2000 caracteres, com incrementos de 200 caracteres. O pseudocódigo para a geração dessas strings é apresentado abaixo:

Algorithm 6 Geração de Strings Aleatórias

Input: Tamanho n
Output: String aleatória de tamanho n
Function `generateRandom(`n`)`**:**
    $resultado$ = null
    $caracteres$ = "abcdefghijklmnopqrstuvwxyz"
    for *i=1 to* n **do**
        $resultado$ += $caracteres$[gerarnumeroaleatorio(0, n($caracteres$)-1]
    return $resultado$

5.1.2. Transformação de String em Árvore

Após a geração, cada string é transformada em uma árvore. Cada caractere da string é convertido em um nó da árvore, com a raiz sendo o primeiro caractere e todos os caracteres subsequentes sendo filhos diretos da raiz, resultando em uma estrutura de árvore com profundidade 2. O pseudocódigo para esta transformação é fornecido a seguir:

Algorithm 7 Transormação de String em Árvore

Input: String s
Output: $root$ de uma árvore gerada a partir de s
Function `stringToTree` *(*s*)***:**
    if s==*null* **then**
        return *null*
    $raiz$ = Novo TreeNode(s[0])
    for *i=1 to* s*.tamanho* **do**
        $root$.addFilho(novo TreeNode(s[i]))
    return $root$

A escolha de transformar strings em árvores com profundidade 2 é motivada por vários fatores, principalmente relacionados ao controle da profundidade (L) da árvore e ao impacto desta profundidade no tempo de execução do algoritmo de Zhang-Shasha.

- **Profundidade (L) e Tempo de Execução**:
  - A profundidade de uma árvore (L) tem um impacto significativo no tempo de execução do algoritmo de Zhang-Shasha. O tempo de execução é diretamente

proporcional à profundidade da árvore, além do número de nós e folhas. Transformando a árvore para ter uma profundidade fixa de 2, podemos isolar a influência de L e focar no número de nós.

- **Estrutura com Profundidade 2**:
  - Transformando uma string em uma árvore onde a raiz tem todos os outros caracteres como filhos diretos, a profundidade da árvore é sempre 2. Isso simplifica a análise e permite uma comparação direta entre diferentes strings de comprimento variável sem o impacto da profundidade variável.
- **Ambiente Controlado**:
  - Em um ambiente controlado, onde queremos minimizar a influência de variáveis externas e focar em aspectos específicos do algoritmo, a transformação para uma árvore com profundidade fixa garante que estamos lidando com uma estrutura previsível e uniforme. Isso é crucial para experimentos que visam isolar o efeito de um parâmetro específico, como o número de nós.
- **Facilidade de Implementação**:
  - A implementação da transformação de strings em árvores com profundidade 2 é direta e evita a complexidade adicional que surgiria ao lidar com árvores com estruturas mais variadas e profundidades irregulares.

Portanto, a transformação de strings em árvores com profundidade 2 não só facilita a implementação e o entendimento do algoritmo, mas também permite um controle mais preciso sobre os parâmetros que influenciam o tempo de execução, proporcionando um ambiente de teste mais rigoroso e controlado.

5.2. Implementação dos algorítimos principais

Esta seção detalha as escolhas específicas dos três algoritmos de distância de edição entre árvores de nossa implementação: o método recursivo ingênuo, o algoritmo de Zhang-Shasha e o algoritmo de Tai.

5.2.1. Método Recursivo Ingênuo

O método recursivo ingênuo foi escolhido para fornecer uma linha de base para a eficiência dos algoritmos mais avançados. Apesar de sua ineficiência em casos de grandes entradas, sua simplicidade oferece uma clareza sobre o funcionamento básico das distâncias de edição entre árvores. O pseudocódigo para o método recursivo ingênuo é:

Algorithm 8 Método Recursivo Ingênuo

Input: Árvores $t1$ e $t2$
Output: Distância de edição entre árvores
Function `editDistRec(`$t1$,$t2$`)`**:**

```
if t1==null then
    return tamanho t2
if t2==null then
    return tamanho t1
custo = (t1.rótulo == t2.rótulo) ? 0:1
if t1.filhos == null and t2.filhos ==null then
    return custo
custoMin = ∞
foreach filho i de t1 do
    foreach filho j de t2 do
        custoAtual = editDistRec(t1.filhos[i], t2.filhos[j])
        foreach t1.filho != i do
            custoAtual += soma de editDistRec()
        foreach t2.filho != j do
            custoAtual += soma de editDistRec()
    custoMin = min(custoMin,custoAtual)
return custo + custoMin
```

5.2.2. Algoritmo de Zhang-Shasha

O algoritmo de Zhang-Shasha foi implementado devido à sua abordagem otimizada utilizando programação dinâmica, que é mais eficaz para árvores de tamanho moderado.No contexto deste trabalho, deixamos explícito que, ao contrário de alguns algoritmos de distância de edição de árvores, a nossa abordagem não exige que a ordem das árvores seja pós-ordem, devido a jeito que a arovre foi estruturada , comentado anteriormente nesse artigo. Isso permite maior flexibilidade na aplicação do algoritmo, uma vez que não estamos restritos a uma ordem específica de processamento dos nós. Este algoritmo é particularmente adequado para aplicações onde a precisão é crítica. O pseudocódigo para o algoritmo de Zhang-Shasha é:

Algorithm 9 Zhang-Shasha

Input: Árvores $t1$ e $t2$
Output: Distância de edição entre árvores
Function `editDistRec(`$t1$,$t2$`)`**:**
  if $t1$==*null* **then**
    return $t2$.*tamanho*
  if $t2$==*null* **then**
    return $t1$.*tamanho*
  inicializa $dp[t1.tamanho + 1][t2.tamanho + 1]$
  for *i de 0 até* $t1$.*tamanho* **do**
    $dp[i][0] = i$
  for *j de 0 até* $t2$.*tamanho* **do**
    $dp[0][j] = i$
  for *i de 1 até* $t1$.*tamanho* **do**
    for *j de 1 até* $t2$.*tamanho* **do**
      $custoSub =$ (filho i-1 de $t1$.rótulo == filho j-1 de $t2$.rótulo) ? 0:1

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1, \\ dp[i][j-1] + 1, \\ dp[i-1][j-1] + \text{custoSub} \end{cases}$$

  return *dp[m][n]+(*$t1$.*rótulo == *$t2$.*rótulo) ? 0:1*

5.2.3. Algoritmo de Tai

A implementação do algoritmo de Tai é utilizada por sua eficiência em comparar árvores com grandes disparidades em tamanho e estrutura. Este método também utiliza programação dinâmica para otimizar o cálculo da distância de edição. O pseudocódigo para o algoritmo de Tai é:

Algorithm 10 Algoritmo de Tai

Input: Árvores $t1$ e $t2$
Output: Distância de edição entre árvores
Function `transformCost(a,b)` **:**
  if a==*null and* b ==*null* **then**
    return *0*
  if a==*null* **then**
    return *1*
  if b==*null* **then**
    return *1*
  return *a.rótulo == b.rótulo ? 0 : 1*

Function `tai(`$t1$, $t2$`)` **:**
  if $t1$ == *null or* $t2$ == *null* **then**
    return *0*
  m = número de filhos de $t1$
  n = número de filhos de $t2$
  inicializa dp[m+1][n+1]
  for *i de 1 até* m **do**
    dp [i][0]= dp[i-1][0] + `transformCost` (*filho i-1 de* $t1$,*null*)
  for *i de 1 até* n **do**
    dp [0][j]= dp[0][j-1] + `transformCost` (*null,filho j-1 de* $t2$)
  for *i de 1 até* m **do**
    for *j de 1 até* n **do**

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + \text{transformCost}(\text{filho } i-1 \text{ de } t1, \text{null}), \\ dp[i][j-1] + \text{transformCost}(\text{null}, \text{filho } j-1 \text{ de } t2), \\ dp[i-1][j-1] + \text{tai}(\text{filho } i-1 \text{ de } t1, \text{filho } j-1 \text{ de } t2) + \text{custoSub} \end{cases}.$$

  return $dp[m][n]$

Cada algoritmo foi escolhido e implementado considerando-se a complexidade do problema e as características específicas das árvores a serem comparadas.

5.2.4. Teste dos Algoritmos - Main

Utilizando as árvores geradas, aplicam-se três algoritmos distintos de distância de edição: o método recursivo ingênuo, o algoritmo de Zhang-Shasha, e o algoritmo de Tai. Os testes são realizados usando pares de árvores geradas aleatoriamente. O tempo de execução de cada algoritmo é medido para avaliar a performance em diferentes tamanhos de entrada, sendo uma média do tempo de 10 execuções diferentes. A análise também inclui a avaliação da complexidade de espaço de cada método.

A abordagem adotada permite uma comparação direta da performance dos algoritmos em condições controladas, com dados de entrada variados, fornecendo uma avaliação robusta da eficácia e eficiência dos métodos em cenários práticos de aplicação.

6. Discussão e Análise de Resultados

6.1. Comparação da Complexidade Espacial dos Algoritmos

Neste trabalho, analisamos a complexidade espacial de três algoritmos principais para calcular a distância de edição entre árvores: o algoritmo recursivo ingênuo, o algoritmo de Tai e o algoritmo de Zhang-Shasha.

A complexidade espacial do algoritmo recursivo ingênuo é $O(\max(m, n))$, onde m e n são os números de nós nas árvores $T1$ e $T2$. Esta complexidade é linear em relação ao número máximo de nós em uma das árvores, uma vez que o algoritmo utiliza apenas um número constante de variáveis locais e não aloca espaço adicional além da pilha de chamadas recursivas. Por não utilizar memoization, o algoritmo recursivo ingênuo evita o armazenamento de resultados intermediários, o que resulta em uma eficiência espacial maior comparada aos outros algoritmos.

Para o algoritmo de Tai, a complexidade espacial é $O(\max(V \cdot L, V' \cdot L'))$, onde V e V' são os números de nós nas árvores $T1$ e $T2$, respectivamente, e L e L' são as profundidades máximas das árvores. Esta complexidade é determinada pela profundidade da pilha de chamadas recursivas e pelo armazenamento necessário para manter os resultados intermediários. O uso de memoization neste algoritmo melhora a eficiência temporal, mas aumenta a complexidade espacial devido ao espaço necessário para armazenar esses resultados intermediários.

O algoritmo de Zhang-Shasha apresenta uma complexidade espacial de $O(n \cdot m)$, onde n e m são os números de nós nas árvores $T1$ e $T2$. A complexidade espacial é determinada pela necessidade de armazenar as distâncias de edição em uma matriz de tamanho proporcional ao produto dos tamanhos das duas árvores. Este algoritmo também utiliza memoization para armazenar as distâncias de edição entre subárvores, o que, embora aumente a eficiência temporal, resulta em um uso maior de espaço.

Em resumo, a comparação da complexidade espacial dos três algoritmos mostra que o algoritmo recursivo ingênuo é o mais eficiente em termos de espaço, pois não utiliza memoization e depende apenas da profundidade da pilha de chamadas. O algoritmo de Tai, embora eficiente em termos de tempo devido ao uso de memoization, requer um espaço significativo para armazenar os resultados intermediários. O algoritmo de Zhang-Shasha, por outro lado, oferece um equilíbrio entre eficiência temporal e espacial, tornando-se uma escolha preferida para a maioria das aplicações práticas.

Algoritmo	**Complexidade Espacial**
Recursivo Ingênuo	$O(\max(m, n))$
Tai	$O(\max(V \cdot L, V' \cdot L'))$
Zhang-Shasha	$O(n \cdot m)$

Tabela 2. Comparação da complexidade espacial dos algoritmos de distância de edição entre árvores

6.2. Resultados Obtidos

Tamanho (caracteres)	Recursivo Ingênuo (ms)	Recursivo Ingênuo (s)	Zhang-Shasha (ms)	Zhang-Shasha (s)	Tai (ms)	Tai (s)
200	141	0	1	0	10	0
400	1178	1	6	0	40	0
600	5677	5	13	0	158	0
800	12612	12	22	0	233	0
1000	21629	21	33	0	324	0
1200	37591	37	41	0	472	0
1400	63086	63	50	0	652	0
1600	85623	85	76	0	858	0
1800	139436	139	105	0	1120	1
2000	213323	213	135	0	1590	1

Tabela 3. Comparação dos tempos de execução dos algoritmos em milissegundos e segundos

6.2.1. Recursivo Ingênuo

O algoritmo Recursivo Ingênuo apresentou um crescimento exponencial no tempo de execução à medida que o tamanho das árvores aumentou. Observamos que para árvores com 200 caracteres, o tempo de execução foi de 141 ms, mas para árvores com 400 caracteres, esse tempo aumentou drasticamente para 1178 ms (1 s). Essa tendência de crescimento exponencial continua, chegando a 5677 ms (5 s) para árvores de 600 caracteres e 21629 ms (21 s) para árvores de 1000 caracteres. Finalmente, para árvores de 2000 caracteres, o tempo de execução atingiu 213323 ms (213 s).

Este comportamento confirma a complexidade temporal $O(3^n)$ do algoritmo Recursivo Ingênuo. Apesar de sua implementação simples e direta, o tempo de execução aumenta rapidamente, tornando-o impraticável para árvores de tamanho moderado a grande. Este fato limita significativamente sua aplicabilidade em cenários reais que envolvem grandes conjuntos de dados.

6.2.2. Zhang-Shasha

Em contraste, o algoritmo de Zhang-Shasha demonstrou um desempenho significativamente melhor. Para árvores com 200 caracteres, o tempo de execução foi de apenas 1 ms. Para árvores de 400 caracteres, o tempo aumentou ligeiramente para 6 ms, e continuou a crescer de forma controlada, atingindo 13 ms para 600 caracteres e 33 ms para 1000 caracteres. Mesmo para árvores de 2000 caracteres, o tempo de execução permaneceu relativamente baixo, em 135 ms.

Com uma complexidade $O(\min(n_{height}, n_{leaves}) \times \min(m_{height}, m_{leaves}) \times nm)$, o algoritmo de Zhang-Shasha é altamente eficiente para grandes tamanhos de árvore. Esta eficiência é evidenciada pelo crescimento linear no tempo de execução conforme o tamanho das árvores aumenta. A diferença de desempenho em comparação com o algoritmo Recursivo Ingênuo

torna Zhang-Shasha uma escolha preferida para aplicações práticas que exigem a comparação de grandes estruturas de árvore.

6.2.3. Tai

O algoritmo de Tai também apresentou um desempenho bastante eficiente, embora geralmente um pouco menos eficiente que o algoritmo de Zhang-Shasha. Para árvores com 200 caracteres, o tempo de execução foi de 10 ms, aumentando para 40 ms para árvores de 400 caracteres. Para árvores de 600 caracteres, o tempo foi de 158 ms, e para árvores de 1000 caracteres, 324 ms. Para árvores de 2000 caracteres, o tempo de execução foi de 1590 ms (1 s).

Apesar de não ser tão rápido quanto o algoritmo de Zhang-Shasha, o algoritmo de Tai ainda oferece uma alternativa viável e eficiente ao Recursivo Ingênuo, com uma complexidade polinomial que permite um crescimento mais controlado no tempo de execução. Esta eficiência relativa torna o algoritmo de Tai uma opção prática para a comparação de árvores em diversas aplicações, especialmente quando a estrutura das árvores não é extremamente grande.

6.3. Análise de Resultados

Nesta seção, apresentamos uma análise detalhada dos resultados obtidos com os quatro gráficos gerados, que comparam o tempo de execução dos algoritmos de Distância de Edição de Árvores: Recursivo Ingênuo, Zhang-Shasha e Tai. Os gráficos estão organizados de forma a facilitar a visualização dos dados e a análise da complexidade dos algoritmos.

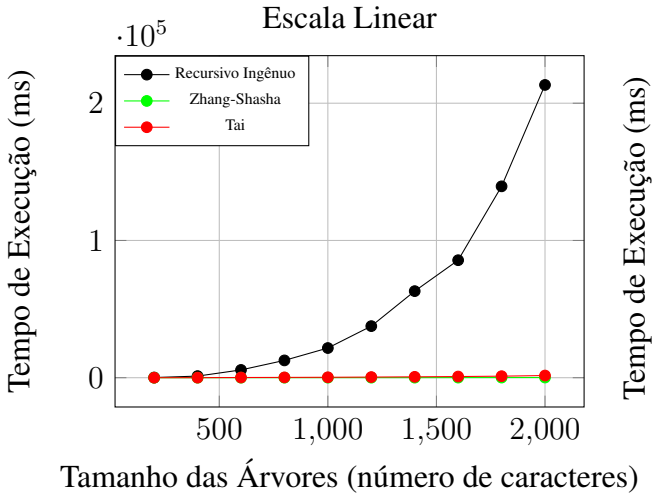

Figura 1. Comparação do Tempo de Execução (Escala Linear)

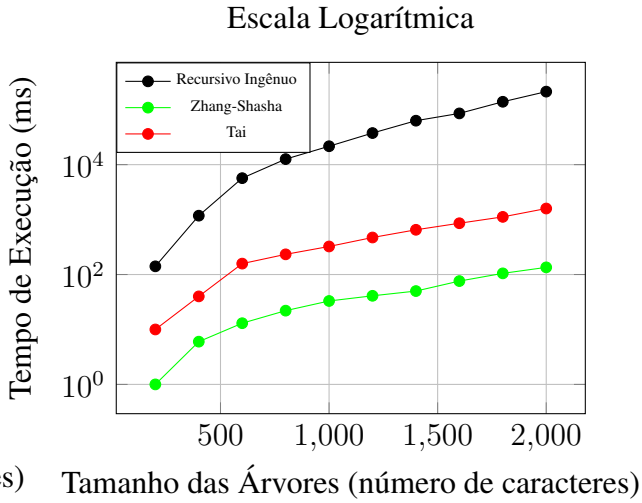

Figura 2. Comparação do Tempo de Execução (Escala Logarítmica)

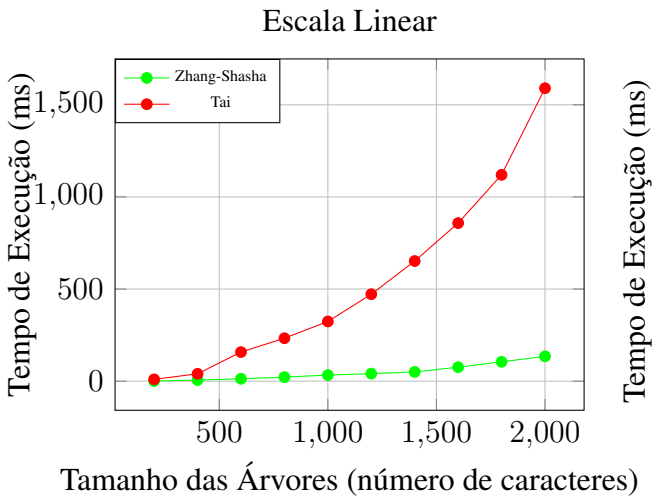

Figura 3. Comparação entre Zhang-Shasha e Tai (Escala Linear)

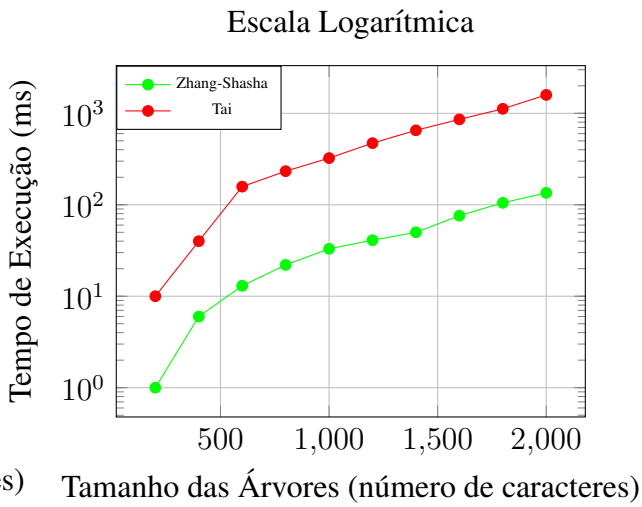

Figura 4. Comparação entre Zhang-Shasha e Tai (Escala Logarítmica)

6.3.1. Comparação Geral dos Algoritmos

Observando a Figura 1, que mostra a comparação do tempo de execução dos algoritmos em escala linear, é evidente que o algoritmo Recursivo Ingênuo possui um crescimento exponencial no tempo de execução à medida que o tamanho das árvores aumenta. Este comportamento é esperado devido à sua complexidade computacional elevada, tornando-o impraticável para árvores grandes. Por outro lado, os algoritmos Zhang-Shasha e Tai demonstram um desempenho significativamente melhor, com tempos de execução muito menores.

A Figura 2, em escala logarítmica, reforça esta observação. A escala logarítmica permite visualizar mais claramente as diferenças de crescimento entre os algoritmos. Novamente, vemos que o algoritmo Recursivo Ingênuo cresce muito mais rapidamente em comparação aos outros dois. O algoritmo de Tai apresenta um crescimento mais moderado, enquanto o algoritmo de Zhang-Shasha mantém-se com um desempenho ainda melhor, evidenciando sua eficiência superior.

6.3.2. Comparação entre Zhang-Shasha e Tai

Para uma análise mais detalhada entre os algoritmos Zhang-Shasha e Tai, as Figuras 3 e 4 mostram comparações em escalas linear e logarítmica, respectivamente. Em ambas as figuras, fica claro que o algoritmo de Zhang-Shasha é consistentemente mais eficiente do que o de Tai.

A Figura 3, em escala linear, destaca que, embora ambos os algoritmos sejam eficientes, o tempo de execução do algoritmo de Zhang-Shasha é ligeiramente menor para todos os tamanhos de árvores testados. Já na Figura 4, em escala logarítmica, essa diferença torna-se mais evidente, confirmando que o algoritmo de Zhang-Shasha mantém uma vantagem consistente em termos de tempo de execução.

Este desempenho superior do algoritmo de Zhang-Shasha pode ser atribuído à sua complexidade $O(\min\{n_{height}, n_{leaves}\} \times \min\{m_{height}, m_{leaves}\} \times nm)$, que é mais eficiente em cenários práticos, especialmente quando L é pequeno, sendo a profundidade da árvore. Nesse trabalho , devida a decisão de implementação da função `stringToTree` L sempre será igual

a 2, a eficiência do algoritmo de Zhang-Shasha é ainda mais pronunciada, permitindo um processamento mais rápido e eficaz em comparação com o algoritmo de Tai, cuja complexidade é $O(V \cdot V' \cdot L^2 \cdot L'^2)$.

A escolha de manter L igual a 2 foi uma decisão de implementação, relacionada à nossa função `stringToTree`. Esta função transforma uma string em uma árvore, onde cada caractere da string se torna um nó da árvore, e cada nó subsequente é adicionado como filho do nó anterior, resultando em uma estrutura de árvore linear. Esta estrutura simples e consistente permitiu que os testes fossem realizados em um cenário controlado, facilitando a comparação direta dos tempos de execução entre os diferentes algoritmos.

6.4. Teste de Stress

Testes de stress são cruciais para entender a viabilidade e a eficiência de algoritmos em aplicações práticas, especialmente em contextos onde o tempo de processamento é um fator crítico. Portanto, foram realizados três diferentes testes com os algoritmos de distância de edição de Zhang-Shasha e Tai, os quais foram observados um cenário de alto volume de dados isolado e juntamente com um cenário com largura máxima da árvore e um cenário com árvores adversárias quase idênticas.

Tamanho dos Dados	Tempo Zhang-Shasha (ms)	Tempo Tai (ms)
100.000	44.033	44.139
200.000	175.923	175.534
300.000	394.550	393.712

Tabela 4. Tempos de cálculo para os algoritmos Zhang-Shasha e Tai no cenário de diferentes tamanhos de dados

O comportamento ao aplicar o teste de stress nos cenários de largura máxima da árvore e alto volume de dados foram extremamente similares, no qual os resultados indicam que ambos os algoritmos apresentam tempos de cálculo que crescem de forma aproximadamente linear com o aumento do tamanho dos dados, o que é esperado dada a complexidade algorítmica. No entanto, observa-se uma pequena variação no desempenho entre os dois algoritmos, que pode ser atribuída a diferenças sutis nas implementações ou nas estruturas de dados utilizadas.

Já ao aplicar o cenário de árvores adversárias quase idênticas, foi desenvolvido pares de árvores que são estruturalmente e semanticamente quase iguais, contendo somente uma diferença. Dessa forma, o resultado indica que ambos os algoritmos, na medida em que é aumentado o volume dos dados, possuem um comportamento esperado e eficiente de modo que, mesmo para árvores com muitos nós, o tempo de execução ainda foi menor do que comparado ao cenário isolado do aumento de volume de dados. Assim, a escalabilidade desses algoritmos em condições de stress sugere que eles são robustos, mas podem requerer otimizações adicionais ou uso de técnicas paralelas para lidar com volumes de dados ainda maiores.

6.5. Conclusão da Análise

Em resumo, a análise dos gráficos permite concluir que o algoritmo Recursivo Ingênuo, apesar de conceitualmente simples, é altamente ineficiente para árvores de grande porte devido ao seu crescimento exponencial no tempo de execução. O algoritmo de Zhang-Shasha oferece uma melhoria significativa e se destaca como o mais eficiente entre os três, apresentando os melhores tempos de execução em todos os cenários analisados. O algoritmo de Tai também é eficiente, mas é superado pelo de Zhang-Shasha, especialmente em implementações onde L é pequeno.

Os testes de stress realizados até 300.000 vértices revelam que ambos os algoritmos Zhang-Shasha e Tai mantêm um desempenho consistente sob condições extremas, com tempos de processamento que crescem de maneira controlada conforme o aumento do tamanho dos dados. Este resultado é fundamental para aplicações que demandam a manipulação de grandes volumes de dados, sugerindo que, apesar das eficiências observadas, podem ser necessárias otimizações adicionais ou a adoção de técnicas computacionais mais avançadas para garantir a escalabilidade em contextos ainda mais exigentes.

Esta análise reforça a importância de utilizar algoritmos eficientes para problemas de edição de árvores, especialmente em aplicações que lidam com grandes volumes de dados.

7. Trabalhos Futuros

A análise e os testes de stress realizados neste trabalho destacam várias direções promissoras para pesquisas futuras:

- **Otimização dos Algoritmos:** Apesar da eficiência demonstrada pelos algoritmos de Zhang-Shasha e Tai, há espaço para melhorias. Estudos futuros podem explorar técnicas de otimização, como paralelização e o uso de estruturas de dados avançadas, para reduzir ainda mais os tempos de processamento, especialmente em grandes volumes de dados.
- **Implementação em Ambientes Distribuídos:** A implementação dos algoritmos em ambientes distribuídos pode ajudar a lidar com conjuntos de dados extremamente grandes. Investigar como a distribuição de tarefas pode melhorar a eficiência e a escalabilidade dos algoritmos é uma área relevante de pesquisa.
- **Comparação com Outros Algoritmos:** Embora o foco deste trabalho tenha sido nos algoritmos de Zhang-Shasha e Tai, existem outros algoritmos de edição de árvores que podem ser comparados. Avaliar o desempenho desses algoritmos em cenários semelhantes pode ajudar a identificar casos de uso específicos onde outros métodos podem ser mais eficientes.
- **Aplicações Práticas:** Explorar aplicações práticas dos algoritmos em diferentes domínios, como bioinformática, linguística computacional e análise de dados hierárquicos, pode demonstrar a utilidade e versatilidade dos métodos de edição de árvores em situações do mundo real.
- **Avaliação de Robustez:** Testar a robustez dos algoritmos frente a dados ruidosos e incompletos pode ser útil para garantir sua aplicabilidade em cenários menos controlados. Desenvolver técnicas para lidar com tais dados sem comprometer significativamente a eficiência é uma área de interesse.

Essas direções de pesquisa não apenas ampliam o conhecimento teórico sobre algoritmos de edição de árvores, mas também visam a aplicação prática em contextos que demandam alta eficiência e escalabilidade. A evolução contínua desses algoritmos pode ter um impacto significativo em diversas áreas que dependem da comparação e análise de estruturas de dados complexas.

8. Conclusão

Neste trabalho, exploramos a métrica de distância de edição entre árvores, implementando e comparando três algoritmos principais: o recursivo ingênuo, Tai e Zhang-Shasha. Nossos experimentos demonstraram que, embora o algoritmo recursivo ingênuo seja simples, ele é inviável para grandes árvores devido à sua complexidade exponencial. Em contrapartida, os

algoritmos de Tai e Zhang-Shasha se mostraram eficientes e escaláveis, com o algoritmo de Zhang-Shasha apresentando um desempenho superior na maioria dos casos. Concluímos que os algoritmos de Tai e Zhang-Shasha são mais adequados para aplicações práticas que requerem a comparação de grandes estruturas de árvore.

Referências

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT press.

GeeksforGeeks (2023). Edit distance — dp-5. `https://www.geeksforgeeks.org/edit-distance-dp-5/`. Acessado em: 18 de junho de 2024.

Joshi, A. (1987). Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? *Natural Language Processing—Theoretical, Computational and Psychological Perspectives*, pages 206–250.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.

Martin, D. R., Fowlkes, C. C., and Malik, J. (2001). A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. *Proceedings of the 8th International Conference on Computer Vision*, 2:416–423.

Shapiro, B. and Zhang, K. (1988). Comparative analysis of rna secondary structures. *Journal of Computational Biology*, 5(3):343–360.

Tai, K.-C. (1979). The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3):422–433.

Zhang, K. and Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262.