Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

O Problema do Ancestral de Nível: Uma comparação entre implementações

Vinicius Perche de Toledo Agostini

Monografia Final

mac 0499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Carlos Eduardo Ferreira

São Paulo
5 de dezembro de 2019

Resumo

Vinicius Perche de Toledo Agostini. **O Problema do Ancestral de Nível: Uma comparação entre implementações**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

O problema do Ancestral de Nível consiste em preprocessar uma árvore enraizada T para então responder consultas AN(u, d), que pedem o ancestral do nó u com profundidade d. Vários algoritmos com diferentes complexidades já foram propostos, porém existem poucos estudos comparativos e implementações disponíveis, principalmente em português. Neste trabalho, foram implementados e comparados quatro algoritmos, levando em consideração árvores com diferentes formatos para analisar na prática o comportamento de cada um conforme o fator de ramificação varia, indo além da análise tradicional de complexidade de tempo e espaço no pior caso. Os resultados obtidos estão de acordo com o que era esperado pela análise teórica. Além disso, foi possível observar que algoritmos simples podem ser implementados de maneiras que, dependendo das características da árvore de entrada, pode-se obter performances suficientes, o que leva a reforçar o quão importante é conhecer a aplicação em questão para poder tomar a melhor decisão quanto a qual algoritmo implementar, balanceando questões como uso de memória, dificuldade de implementação e manutenção do código e eficiência.

Palavras-chave: Ancestral de Nível. Árvores. Implementação.

Abstract

Vinicius Perche de Toledo Agostini. **The Level Ancestor Problem: A comparison between implementations**. Undergraduate Thesis (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

The Level Ancestor problem consists of preprocessing a rooted tree T to answer queries AN(u, d) which request the ancestor of node u with depth d. There have been several algorithms with different complexities proposed, however, there are few comparative studies and implementations available, especially in portuguese. In this study, four algorithms were implemented and tested with trees of varied shape to analyse, in practice, each algorithm's behaviour as the branching factor changes, going beyond the traditional worst-case time and space complexity analysis. The obtained results were consistent with what was expected from theoretical analysis, but it allowed to observe that simple algorithms can be implemented in such ways that, depending on the characteristics of the input tree, it can achieve sufficient performance, reinforcing how important it is to know the actual application so that an informed decision is made about which algorithm to implement, balancing questions such as memory use, difficulty of code implementation and maintainability and its efficiency.

Keywords: Level Ancestor. Trees. Implementation.

Lista de Figuras

Lista de Tabelas

Lista de Programas

Sumário

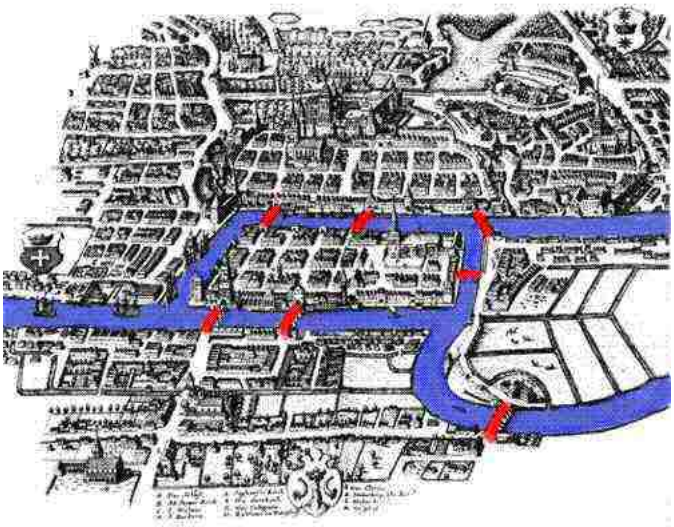
Apêndices

Capítulo 1

Uma introdução à Teoria dos Grafos

1.1 As Pontes de Königsberg

No século XVIII, a cidade de Königsberg, na Prússia (hoje Kaliningrado, na atual Rússia) era um importante centro comercial devido à sua localização próxima ao rio, que dividia a cidade em quatro regiões, interligadas por sete pontes.



Fonte: MacTutor History of Mathematics archive

Figura 1.1: *A cidade de Königsberg e suas sete pontes*

Segundo histórias, um passatempo de domingo dos cidadãos de Königsberg consistia em passear por sua cidade, até que um dia surgiu um desafio: encontrar uma forma de andar por todas as regiões passando por cada ponte apenas uma vez.

Nenhum habitante de Königsberg conseguiu encontrar a solução para este problema, porém, o desafio chegou até um homem chamado Leonhard Euler, que apesar de julgar o

problema trivial, ficou intrigado, como citado em Hopkins e Wilson (2004):

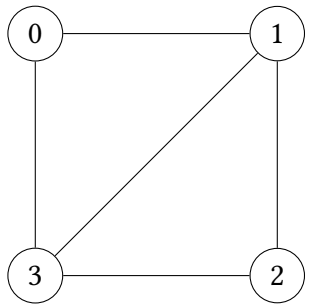
> *"This question is so banal, but seemed to me worthy of attention in that [neither] geometry, nor algebra, nor even the art of counting was sufficient to solve it."*

Euler provou que não era possível passar por toda a cidade de Königsberg sem passar duas vezes pela mesma ponte, mas também resolveu o caso geral em Euler (1741), para qualquer número de regiões e qualquer número de pontes, dando assim origem a um ramo da matemática que hoje chamamos de **Teoria dos Grafos**.

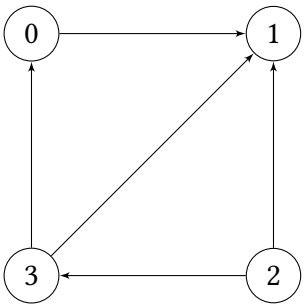
1.2 Grafos

Podemos definir um **Grafo** informalmente como um conjunto de entidades conectadas entre si. Mais formalmente, podemos definir um grafo G como um par (V, E) onde V é um conjunto de **vértices** e E um conjunto de **arestas** entre os vértices tal que $E \subseteq \{(u, v) \mid u, v \in V\}$.

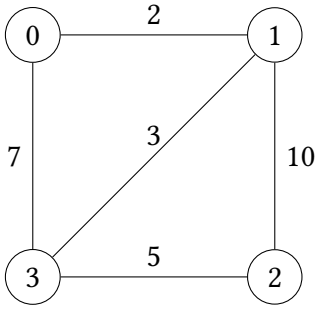
As arestas de um grafo podem ter propriedades como direção e/ou algum valor associado dependendo do contexto em que está sendo utilizado, frequentemente possibilitando soluções elegantes e eficientes para os mais diversos problemas.



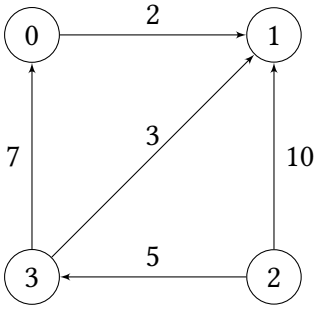
(a) *Um grafo não dirigido.*



(b) *Um grafo dirigido.*



(c) *Um grafo com valores nas arestas.*



(d) *Um grafo dirigido com valores nas arestas.*

Figura 1.2: *Exemplos de grafos.*

1.3 Árvores

Árvores são um subconjunto dos grafos de incrível importância para a Ciência da Computação, sendo usadas nas mais diversas aplicações, desde a estrutura de pastas de um sistema operacional até bancos de dados e processamento de linguagem natural.

Diferente de outras estruturas como vetores, as árvores se organizam de forma não linear, hierárquica. Os elementos de uma árvore são chamados de **nós** e comumente um nó específico é elegido para ser a sua **raiz**. Cada nó pode ter ligações com outros nós denominados **filhos**, que por sua vez podem ter seus próprios filhos e assim sucessivamente até chegar em nós que não possuem ligação com nenhum outro elemento. Chamamos estes nós de **folhas**.

Também é definida a **profundidade** de um nó como a quantidade de arestas no caminho da raiz até ele, ou seja, a raiz tem profundidade zero, seus filhos profundidade um e assim sucessivamente. Cada **nível** da árvore corresponde a uma "fatia horizontal" que contém os nós de uma certa profundidade; por definição, o nível de um nó *u* é *profundidade(u) + 1*.

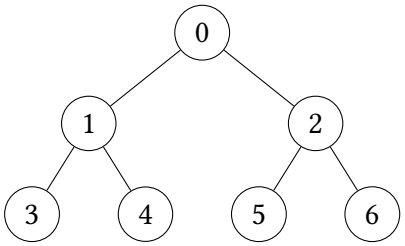
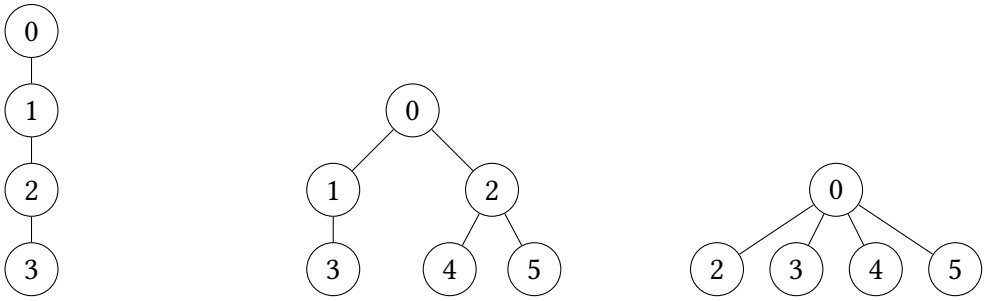


Figura 1.3: ***Exemplo de árvore.*** *O nó 0 é a raiz e tem como filhos os nós 1 e 2, cujos filhos são as folhas 3, 4 e 5, 6, respectivamente. O primeiro nível contém o nó 0 e tem profundidade zero, o segundo os nós 1 e 2 com profundidade um e o terceiro os nós 3, 4, 5 e 6 com profundidade dois.*

1.3.1 Fator de ramificação e árvores k-árias

O **fator de ramificação** de uma árvore é definido como a quantidade média de filhos que cada um de seus nós tem e pode ser obtido ao dividir o número de arestas pelo número de nós com pelo menos um filho. Uma árvore cujos nós têm não mais que k filhos é chamada de **árvore k-ária**; em particular, quando $k = 2$ estas árvores são apelidadas de **árvores binárias** e quando $k = 4$, **árvores quaternárias**. Outro caso especial é quando $k = 1$, criando uma **árvore linear**, essencialmente uma lista ligada de seus nós.

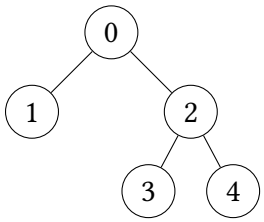


(a) *Árvore linear.* **(b)** *Árvore binária.* **(c)** *Árvore quaternária.*

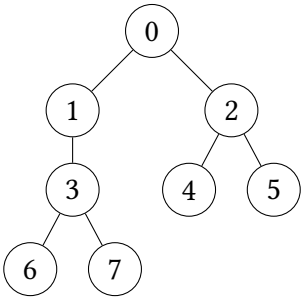
Figura 1.4: *Exemplos de árvores com fatores de ramificação diferentes.*

1.3.2 Árvores completas

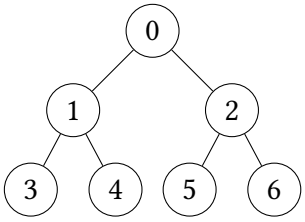
É dito que um nível i de uma árvore k-ária é completo se este possui todos os nós possíveis, isto é, k^{i-1}. Uma **árvore k-ária completa** é uma em que todos os níveis exceto possivelmente o último são completos e caso este não seja, seus nós devem estar o mais à esquerda possível.



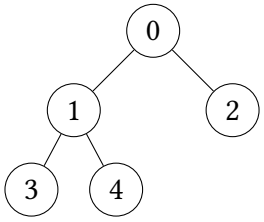
(a) *Nós 3 e 4 não estão o mais à esquerda possível.*



(b) *O terceiro nível não está completo (o nó 1 tem apenas um filho).*



(c) *Árvore binária completa.*



(d) *Árvore binária completa.*

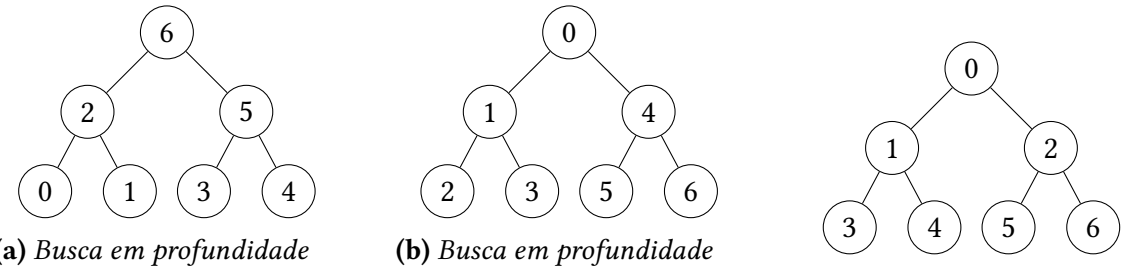
Figura 1.5: *Exemplos de árvores completas ou não.*

1.3.3 Busca em profundidade e travessias

Árvores podem ser percorridas de diferentes maneiras partindo de sua raiz, podendo "conhecer" seus nós em ordens diferentes a depender de algum critério definido para

escolher o próximo nó a ser examinado. Em particular duas maneiras bastante conhecidas para realizar a travessia de uma árvore são a **busca em profundidade** e a **busca em largura**, onde a primeira tenta se aprofundar na árvore o máximo possível antes de explorar outros caminhos e a segunda explora a árvore nível a nível, conhecendo todos os nós de determinada profundidade antes de conhecer nós mais profundos.

Uma busca em profundidade também pode ser realizada de múltiplos jeitos, variando a ordem em que o nó e seus filhos são processados. Uma **travessia em pré-ordem** significa que o nó será processado primeiro e então seus filhos serão colocados numa pilha de execução da esquerda para a direita, enquanto numa **travessia em pós-ordem** empilha os filhos do nó antes de processá-lo, como ilustrado na Figura 1.6.



(a) *Busca em profundidade em pós-ordem.*

(b) *Busca em profundidade em pré-ordem.*

(c) *Busca em largura.*

Figura 1.6: ***Ordem de visita aos nós da árvore durante as travessias.*** *O número do nó representa a ordem em que ele foi processado.*

Para o escopo desta monografia é particularmente interessante a travessia em **pré-ordem**, que traz consigo algumas propriedades úteis a serem exploradas mais a frente no texto, quando a implementação dos algoritmos for discutida.

Capítulo 2

O Problema do Ancestral de Nível e Implementações

2.1 Definição do problema

O problema do **Ancestral de Nível** é um problema fundamental de árvores que pode ser definido da seguinte forma: Dada uma árvore enraizada T com n nós, para consultas AN(u, d) onde u é um nó de T e d um número inteiro, encontre o ancestral de profundidade d de u otimizando tanto o tempo de preprocessamento quanto o de resposta às consultas.

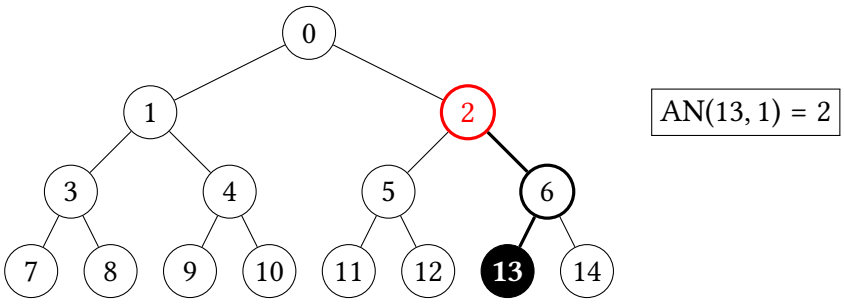


Figura 2.1: *Exemplo de consulta de Ancestral de Nível.*

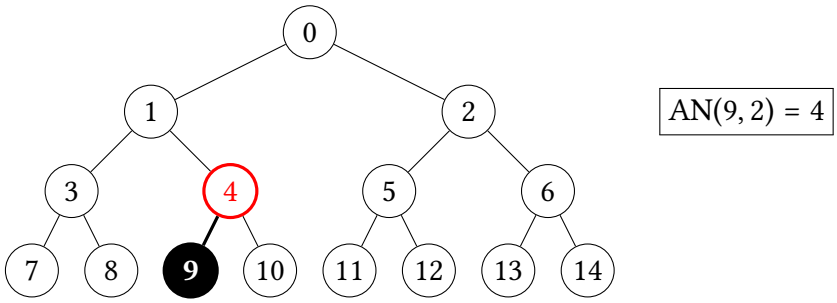


Figura 2.2: *Exemplo de consulta de Ancestral de Nível.*

Existem diversas soluções para tal problema, de diferentes níveis de complexidade e com performances teoricamente diferentes. Em particular, por exemplo, é possível realizar as consultas de forma trivial sem qualquer tipo de preprocessamento ou também preprocessar todas as possíveis consultas para que possam então ser respondidas rapidamente.

Em Bender e Farach-Colton (2002), Papamichail *et al.* (2014) e Menghani e Matani (2019) são discutidas algumas soluções, incluindo as que serão implementadas e testadas neste trabalho. Em particular, serão testadas soluções razoavelmente simples de serem implementadas, até mesmo para mostrar que estas muitas vezes podem ter performance melhor do que imagina-se se formos além de uma simples análise de pior caso.

Como motivação para o estudo deste problema, vale lembrar que o Ancestral de Nível aparece como parte de problemas mais complexos como árvores ordinais espaço-eficientes como descrito em Geary *et al.* (2006) que podem ser usadas na representação de documentos XML que suportam consultas XPath. Além disso, são usadas em Sadakane e Grossi (2006) para implementar estruturas de dados comprimidas e também em Yuan e Atallah (2009) para consultas agregadas em árvores. Por último, vemos aplicações até mesmo no campo de *hashing* em strings, como dito em Farach e Muthukrishnan (1996).

2.2 Implementações

Todas as implementações assumem a mesma implementação de árvore onde é guardada apenas sua raiz e cada nó contém um vetor de ponteiros para seus filhos. Para facilitar as implementações, ao construir a árvore é feita uma busca em profundidade para preencher vetores que servirão como funções globais `pai()` e `profundidade()`; nos códigos abaixo, N é o tamanho da árvore.

Programa 2.1 Criação da árvore.

```
1   FUNCAO cria_arvore(N, nó)
2       raiz ← nó
3       pai ← vetor de tamanho N
4       profundidade ← vetor de tamanho N
5       travessia(raiz, 0)
6   fim
7
8   ROTINA travessia(nó, prof)
9       profundidade[nó] = prof
10      para filho em nó.filhos() faca
11          pai[filho] = nó
12          travessia(filho, prof+1)
13      fim
14  fim
```

Será analisado, para cada algoritmo, tanto sua complexidade de tempo de execução quanto a de espaço adicional, também levando em consideração quão simples é sua im-

plementação. Para tornar a discussão da complexidade de tempo mais clara, será dito que um algoritmo tem complexidade de tempo $\langle f(x), g(x) \rangle$ se a complexidade de seu preprocessamento é $f(x)$ e a de suas consultas é $g(x)$.

2.2.1 Algoritmo Trivial

O primeiro algoritmo a ser estudado é um em que simplesmente não há nenhum preprocessamento e, para cada consulta, apenas subimos pelos pais a partir do nó em questão até encontrarmos seu ancestral com profundidade igual à profundidade requisitada.

Para esta implementação, o tempo de execução de seu preprocessamento não depende do tamanho da árvore e também não utiliza nenhuma memória adicional, portanto o preprocessamento do Algoritmo Trivial tem complexidade O(1) tanto para tempo quanto espaço.

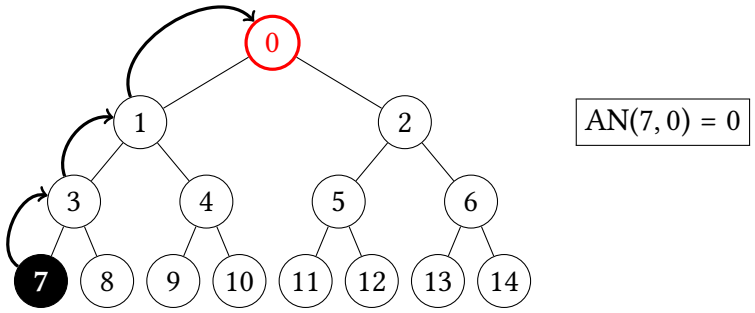


Figura 2.3: *Exemplo de consulta do Algoritmo Trivial.*

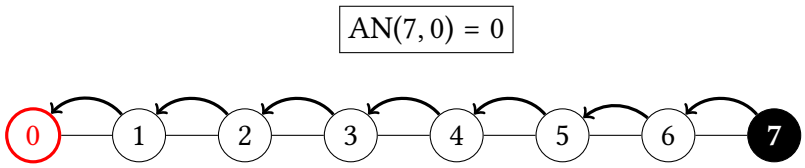


Figura 2.4: *Exemplo de consulta do Algoritmo Trivial.*

No que diz respeito a cada consulta, precisamos chamar a função $pai()$ partindo do nó inicial até chegar no nó com a profundidade desejada. Essa quantidade é exatamente a diferença entre esta profundidade e a profundidade do nó inicial.

A consulta mais lenta para uma árvore qualquer é uma que parte do nó mais profundo e precisa subir até a raiz, levando tempo $O(d)$, onde d é a profundidade da árvore. No pior caso isso é equivalente a $O(n)$ onde n é a quantidade de nós e para uma árvore balanceada cujos nós têm k filhos, é equivalente a $O(\log_k n)$, fazendo com que a performance da consulta seja muito dependente da forma da árvore.

Após essa análise, podemos concluir que o Algoritmo Trivial tem complexidade de tempo $\langle O(1), O(d) \rangle$ e $O(1)$ de espaço adicional, sendo particularmente eficiente para árvores balanceadas e com fator de ramificação maior.

Programa 2.2 Consulta do Algoritmo Trivial.

```
1  FUNCAO consulta(nó, prof)
2      x ⟵ nó
3      enquanto profundidade(x) ≠ prof
4          x ⟵ pai(x)
5      fim
6      devolva x
7  fim
```

2.2.2 Algoritmo da Tabela

Ao contrário do algoritmo trivial, a ideia é precalcular o resultado de todas as consultas possíveis durante a fase de preprocessamento para otimizar o desempenho da fase de consultas.

Cada nó u tem associado a si *profundidade(u) + 1* possíveis consultas. Assim, para uma árvore com n nós, existem $n + \sum_{i=0}^{n-1}$ *profundidade(i)* = O(nd) consultas, onde d é a profundidade da árvore. O preprocessamento se dá de maneira simples, preenchendo uma tabela de tamanho O(nd) com dois laços encadeados, utilizando a função *pai*() para subir a partir de cada nó até a raiz e salvar a resposta para cada profundidade.

Programa 2.3 Preprocessamento do Algoritmo da Tabela.

```
1  ROTINA preprocessa(árvore)
2      para u de 0 ate N–1 faca
3          v ⟵ u
4          tabela[u][profundidade(u)] = u
5          para w de profundidade(u)–1 ate 0 faca
6              tabela[u][w] = pai(v)
7              v = pai(v)
8          fim
9      fim
10 fim
```

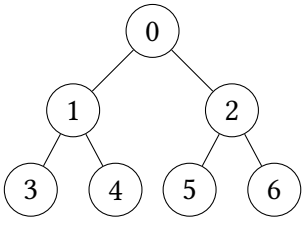


Figura 2.5: *Árvore exemplo.*

	p = 0	p = 1	p = 2
0:	0		
1:	0	1	
2:	0	2	
3:	0	1	3
4:	0	1	4
5:	0	2	5
6:	0	2	6

(i, j) = AN(i, j)

Tabela 2.1: *Tabela de resultados das consultas da árvore da figura 2.5.*

Com a tabela devidamente preenchida, as consultas se tornam tão simples quanto

acessar a sua entrada correspondente, ou seja, a resposta da consulta que pede pelo ancestral do nó u com profundidade p é dada por $tabela[u][p]$.

Programa 2.4 Consulta do Algoritmo da Tabela.

```
1  FUNCAO consulta(nó, prof)
2      devolva tabela[nó][prof]
3  fim
```

A complexidade de espaço e de tempo do preprocessamento dependem fortemente da forma da árvore, podendo ser $O(n^2)$ no caso de uma árvore linear e para o caso de uma árvore k-ária balanceada é $O(n \log_k n)$, muito mais eficiente. Já a consulta é feita em tempo $O(1)$, usando apenas a tabela previamente calculada. Assim, o Algoritmo da Tabela tem complexidade de tempo $\langle O(nd), O(1) \rangle$ e $O(nd)$ de espaço, sendo uma boa opção para casos de árvores balanceadas em que o preprocessamento se torna barato frente à quantidade de consultas que serão feitas e o espaço utilizado não é uma restrição.

2.2.3 Algoritmo dos Ponteiros

Aqui a ideia é subir do nó inicial até a resposta em cada consulta de forma mais inteligente; para isso, a cada nó u da árvore é associado um vetor com ponteiros onde o elemento na posição j aponta para o 2^j-ésimo ancestral de u. Para preencher estes vetores, note que o 2^0-ésimo ancestral de um nó é seu pai e que $2^{j-1} + 2^{j-1} = 2^j$, ou seja, o 2^{j-1}-ésimo ancestral do 2^{j-1}-ésimo ancestral de um nó é seu 2^j-ésimo ancestral:

Programa 2.5 Preprocessamento do Algoritmo dos Ponteiros.

```
1  ROTINA preprocessa(árvore)
2      M ← ⌈log2 d⌉   ▷ d é a profundidade da árvore
3      ponteiros ← matriz N x (M+1)
4      para i de 0 ate N–1 faca
5          ponteiros[i][0] ← pai(i)
6      fim
7      para j de 1 ate M faca
8          para i de 0 ate N–1 faca
9              x ← ponteiros[i][j–1]
10             ponteiros[i][j] ← ponteiros[x][j–1]
11         fim
12     fim
13 fim
```

Uma vez que a tabela de ponteiros está preenchida a consulta se dá, de forma bastante elegante, através da representação binária dos números. Para um número de m bits X, este número pode ser escrito como uma sequência $s_{m-1}s_{m-2}\ldots s_1 s_0$ e $\sum_{i=0}^{m-1} s_i \cdot 2^i = X$. Seja X a diferença entre a profundidade do nó inicial da consulta e a profundidade da resposta, os pulos necessários são dados exatamente pelos bits ligados em X.

Exemplo: seja $X = 85 = 01010101_2$ a distância do nó inicial para a resposta, pode-se pular $2^6 = 64$ nós, depois $2^4 = 16$, $2^2 = 4$ e finalmente apenas $2^0 = 1$ para terminar a consulta.

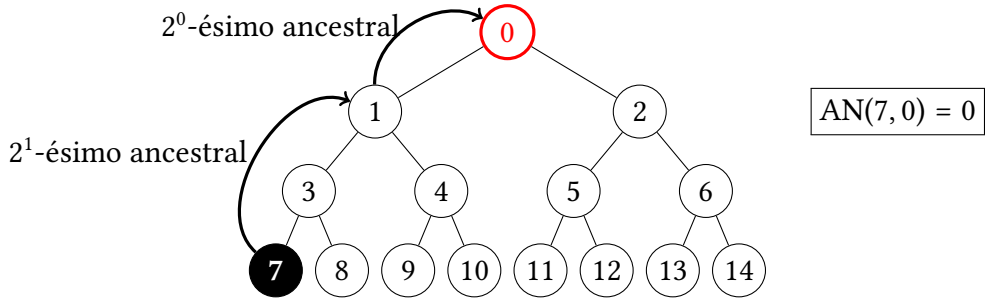


Figura 2.6: ***Exemplo de consulta do Algoritmo dos Ponteiros.*** *A diferença de profundidade é 3 = $011_2 = 2^1 + 2^0$.*

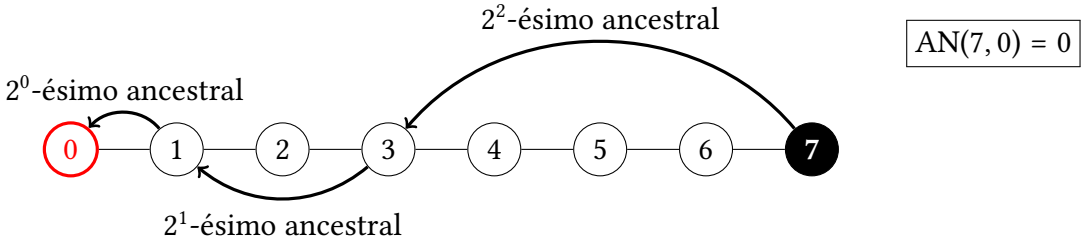


Figura 2.7: ***Exemplo de consulta do Algoritmo dos Ponteiros.*** *A diferença de profundidade é 7 = $0111_2 = 2^2 + 2^1 + 2^0$.*

Programa 2.6 Consulta do Algoritmo dos Ponteiros.

```
1  FUNCAO consulta(nó, prof)
2      x ← nó
3      enquanto profundidade(x) ≠ prof faca
4          d = profundidade(x) − prof
5          pulo ← ⌊log2 d⌋   ▷ seleciona o bit mais significante
6          x ← ponteiros[x][pulo]
7      fim
8      devolva x
9  fim
```

Como a quantidade de passos da consulta é exatamente o número de bits ligados em X, são necessários $O(\log d)$ passos, onde d é a profundidade da árvore. Portanto, a complexidade do Algoritmo dos Ponteiros é $\langle O(n \log d), O(\log d)\rangle$ de tempo e $O(n \log d)$ de espaço, que no caso de uma árvore k-ária balanceada se tornam $\langle O(n \log_k(\log_k n)), O(\log_k(\log_k n))\rangle$ e $O(n \log_k(\log_k n))$, respectivamente.

2.2.4 Algoritmo da Pré-ordem

O último algoritmo considerado neste texto faz uso das propriedades da pré-ordem de uma árvore para trazer um preprocessamento mais eficiente. Enquanto é feita a travessia da árvore, cada nó é associado ao seu índice na pré-ordem, que é inserido numa lista de nós de sua respectiva profundidade.

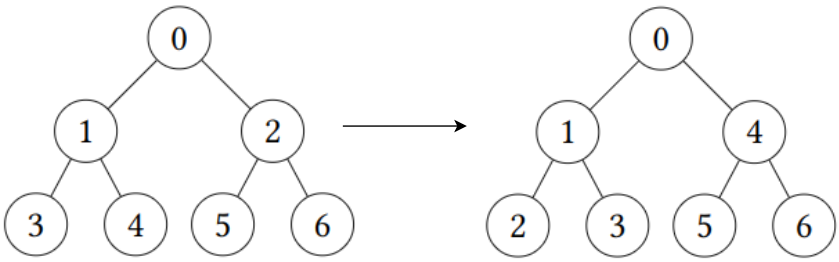


Figura 2.8: *Associação entre os índices originais e os índices da pré-ordem.*

Para tal, a ideia é guardar a ordem em que os nós de cada nível foram visitados durante a travessia em pré-ordem em vetores e então uma consulta se resume a procurar o ancestral do nó inicial que está no vetor correspondente à profundidade buscada, tendo performance proporcional à quantidade de elementos por nível da árvore.

Programa 2.7 Preprocessamento do Algoritmo da Pré-ordem.

```
1   global id = 0
2
3   ROTINA preprocessa(árvore)
4       dfs(árvore.raiz, prof = 0)
5   fim
6
7   ROTINA dfs(nó, prof)
8       preordem[nó] = id
9       nome[id] = nó
10      travessia[prof].insere(id)
11      id ⟵ id + 1
12      para u em nó.filhos() faca
13          dfs(u, prof + 1)
14      fim
15  fim
```

Para entendermos o funcionamento do algoritmo para as consultas, primeiro precisamos nos convencer de que, se um nó v é ancestral de u, então $preordem(v) \leq preordem(u)$, onde $preordem(x)$ é o índice associado à posição de x na travessia em pré-ordem. Isso é verdade já que sempre descobrimos os filhos de um nó depois dele próprio, fazendo com que um nó com índice maior que outro não possa ser seu ancestral. Além disso, no caso em que existam vários nós em determinada profundidade, o ancestral que buscamos é aquele cujo índice na pré-ordem é o mais próximo do índice do nó inicial, porém ainda menor

profundidade 0:	0
profundidade 1:	1 | 4
profundidade 2:	2 | 3 | 5 | 6

Tabela 2.2: ***Vetores contendo os nós processados em cada nível.*** *Os índices guardados nos vetores são referentes à pré-ordem e não os da árvore original.*

que tal. Para realizar esta consulta de forma eficiente e obter complexidade logarítmica, pode-se usar a técnica de busca binária para encontrar o último elemento de um vetor menor ou igual ao parâmetro passado em O(log m) passos, onde m é o tamanho desse vetor, lembrando que a busca deve ser feita com base nos índices da pré-ordem e depois esse índice deve ser convertido para a resposta.

Programa 2.8 Consulta do Algoritmo da Pré-ordem.

```
1  FUNCAO consulta(nó, prof)
2      idx ← travessia[prof].busca_binaria(preordem[nó])
3      devolva nome[idx]
4  fim
```

profundidade 1:	1 | 4

Figura 2.9: ***Exemplo de consulta do Algoritmo da Pré-ordem.*** *Para uma consulta AN(5, 1) na árvore da figura 2.8, a resposta seria o nó com índice de pré-ordem igual a 4, ou seja, o nó 2.*

Finalmente, podemos então concluir que a complexidade de tempo do Algoritmo da Pré-ordem é $\langle O(n), O(\log k^d) \rangle$ já que a quantidade de nós por nível de uma árvore k-ária é $O(k^d)$ e a de espaço é $O(n)$.

2.3 Resumo das implementações

No próximo capítulo estaremos interessados em avaliar a performance de cada algoritmo para alguns casos de teste, levando em consideração o preprocessamento necessário, a realização de consultas e a quantidade de memória utilizada.

Entretanto, analisando as tabelas 2.3, 2.4 pode-se perceber de imediato que restringir a entrada para árvores balanceadas permite até mesmo que os algoritmos mais simples apresentem complexidades boas, o que espera-se que seja refletido nos resultados dos testes. Vale lembrar que dizer que um algoritmo tem complexidade de tempo $\langle f(x), g(x) \rangle$ quer dizer que a complexidade de seu preprocessamento é $f(x)$ e a de suas consultas é $g(x)$.

	Linear	k-ária
Trivial	$\langle O(1), O(n) \rangle$	$\langle O(1), O(\log_k n) \rangle$
Tabela	$\langle O(n^2), O(1) \rangle$	$\langle O(n \log_k n), O(1) \rangle$
Ponteiros	$\langle O(n \log_2 n), O(\log_2 n) \rangle$	$\langle O(n \log_k(\log_k n)), O(\log_k(\log_k n)) \rangle$
Pré-ordem	$\langle O(n), O(1) \rangle$	$\langle O(n), O(\log_k n) \rangle$

Tabela 2.3: *Comparação da complexidade de tempo dos algoritmos.*

	Linear	k-ária
Trivial	$O(1)$	$O(1)$
Tabela	$O(n^2)$	$O(n \log_k n)$
Ponteiros	$O(n \log_2 n)$	$O(n \log_k(\log_k n))$
Pré-ordem	$O(n)$	$O(n)$

Tabela 2.4: *Comparação da complexidade de espaço dos algoritmos.*

Capítulo 3

Benchmarks

3.1 Metodologia

Os algoritmos foram implementados na linguagem C++ usando apenas as bibliotecas padrão e a STL. Todos os programas foram compilados com a versão 7.4.0 do compilador g++, num computador que possui um processador Intel Core i5-8250U com clock base de 1.60GHz e turbo boost até 3.40GHz em uma única thread. As medições de tempo foram feitas com o `steady_clock` da biblioteca `<chrono>`, utilizando uma precisão de nanosegundos, tomando o devido cuidado de cronometrar apenas as partes relevantes do código. Todos os programas foram compilados com a flag `-O2` para permitir otimizações por parte do compilador.

Para cada algoritmo apresentado neste trabalho, foram medidos os tempos de execução tanto da parte de preprocessamento quanto da parte de consultas, separadamente. Cada teste também foi realizado com árvores lineares, binárias e quaternárias para evidenciar possíveis diferenças de performance de acordo com o formato da árvore.

O número de nós das árvores testadas variou entre 150K e 21.6M e as árvores binárias e quaternárias geradas para fins destes testes são completas (portanto balanceadas). Tanto os testes de preprocessamento quanto os de consultas foram executados dez vezes com cada tamanho de árvore para tomar então suas médias como resultado.

Os testes de preprocessamento consistem em um programa que cria uma árvore completa com a quantidade de nós e o fator de ramificação desejados e então cria um objeto da classe associada ao algoritmo a ser testado, o que equivale à fase de preprocessar a árvore de entrada. Já os testes de consultas consistem em um programa que cria também uma árvore completa com os mesmos parâmetros e então executam um conjunto de 10M de consultas gerado previamente. Estes arquivos foram gerados aleatoriamente de forma que toda consulta seja composta por um nó válido (entre 0 e $N - 1$) onde N é o tamanho do experimento e uma profundidade válida (entre 0 e *profundidade(u)*, onde u é o nó da consulta.

3.2 Análise dos resultados

As árvores que surgiriam em aplicações reais possivelmente não seriam exatamente como as árvores aqui testadas, que são todas completas, porém ainda podemos ter uma boa noção de como as diferentes implementações se comportam no pior caso possível e em casos mais favoráveis.

3.2.1 Preprocessamento

Árvores lineares

A primeira coisa a ser notada é que, para árvores lineares, o Algoritmo da Tabela mal pode ser comparado com os outros, já que para este caso sua complexidade de espaço de preprocessamento é $O(n^2)$, se tornando impossível manter o programa na memória até mesmo para o menor tamanho de árvore dos testes padrão, 150K. Apesar disso, o tamanho das árvores foi mantido para que os testes não se tornassem facilmente influenciáveis por fatores do sistema como trocas de contexto, por exemplo. Como evidenciado na figura 3.1, o Algoritmo da Tabela tem desempenho muito pior do que todos os outros até mesmo para os testes reduzidos, o que nos dá segurança para afirmar, juntamente com sua análise de complexidade, que ele também levaria muito mais tempo para completar os testes padrão.

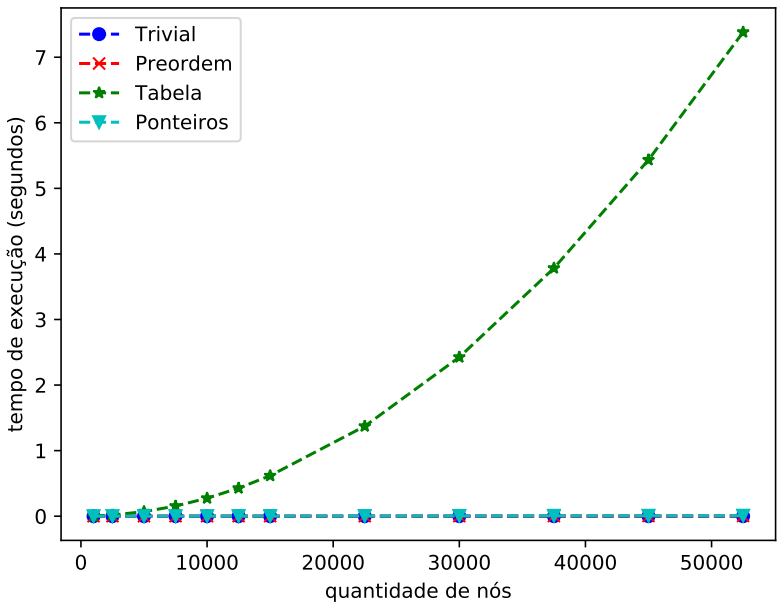


Figura 3.1: *Resultados para o preprocessamento de árvores lineares com uma versão reduzida dos testes.*

Seguindo para a bateria normal de testes, o Algoritmo Trivial, como esperado, é definitivamente o mais rápido, se mantendo essencialmente constante a menos de pequenas

variações, seguido pelo Algoritmo da Pré-ordem que leva uma vantagem grande sobre Algoritmo dos Ponteiros já que este é O($n \log n$) enquanto o primeiro é O(n).

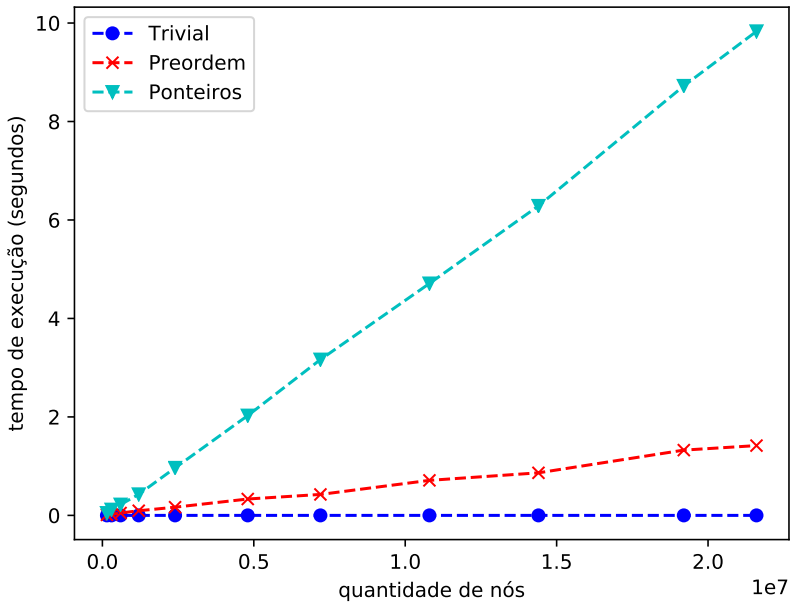


Figura 3.2: *Resultados para o preprocessamento de árvores lineares com os testes padrão.*

Árvores binárias

Para estes testes estamos considerando árvores binárias completas com profundidade esperada $\log_2 n$ onde n é o tamanho do teste, o que torna possível por exemplo rodar o Algoritmo da Tabela para todos os tamanhos já que sua complexidade de espaço se torna $O(n \log n)$, sendo comparável com o Algoritmo dos Ponteiros, que ainda se mostra mais eficiente pela constante menor. O Algoritmo da Pré-ordem se tornou ainda mais rápido se comparado aos testes com árvores lineares apesar de sua complexidade não depender da profundidade da árvore, provavelmente por serem necessárias menos alocações de memória.

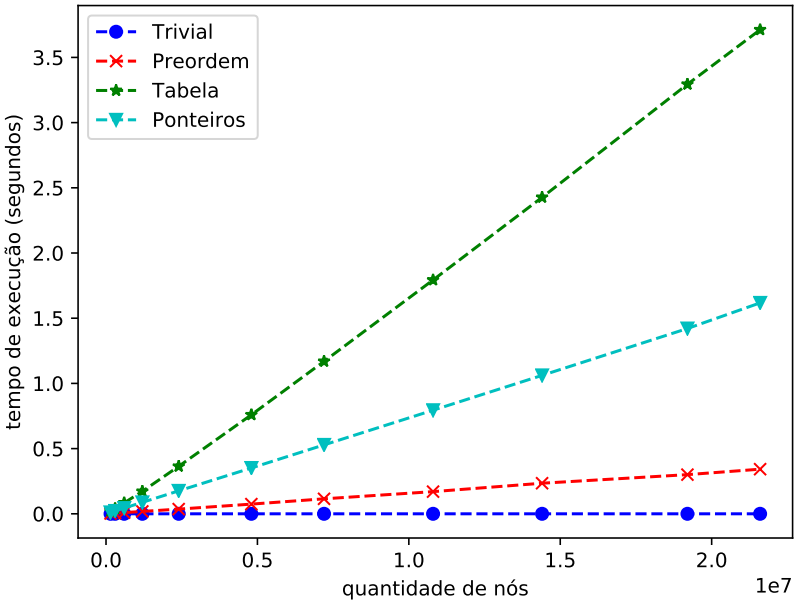


Figura 3.3: *Resultados para o preprocessamento de árvores binárias.*

Árvores quaternárias

Para estas árvores é interessante notar que o desempenho do Algoritmo da Tabela e do Algoritmo dos Ponteiros foram melhores do que para árvores binárias e isso se deve à relação entre suas complexidades de tempo e o fator de ramificação da árvore. O Algoritmo dos Ponteiros se manteve estável já que sua complexidade não depende da forma da árvore, o que o torna pouco adaptável ao formato da entrada.

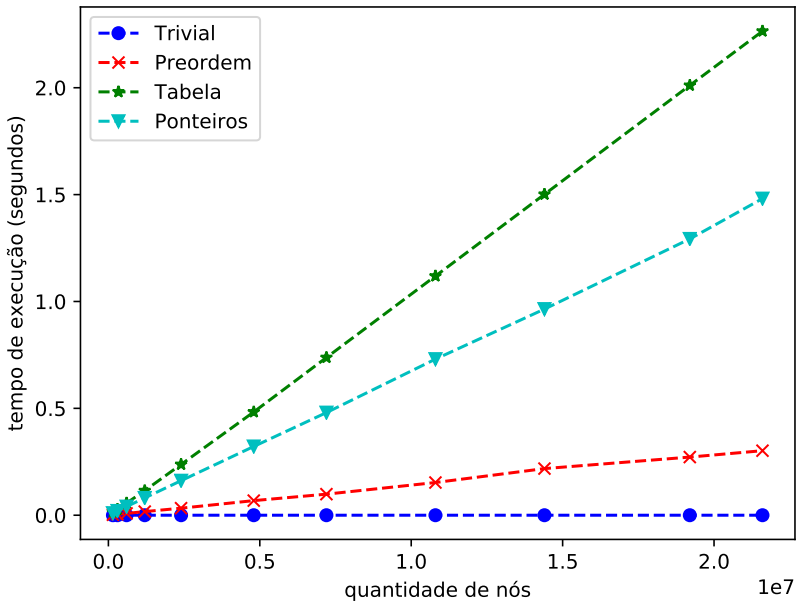


Figura 3.4: *Resultados para o preprocessamento de árvores quaternárias.*

3.2.2 Consultas

Árvores lineares

Aqui, assim como no caso dos testes de preprocessamento, não são todos os algoritmos que conseguiram passar pela bateria de testes padrão. O Algoritmo da Tabela segue com sua limitação de memória, porém agora o Algoritmo Trivial fica bastante lento, chegando perto dos 20 segundos para 1K consultas no maior tamanho de árvore e a metodologia escolhida para os testes de consulta fez com que o Algoritmo dos Ponteiros também não conseguisse ter memória suficiente já que o arquivo de teste era carregado simultaneamente na memória do computador. Todavia, como espera-se que o tempo necessário para responder uma quantidade m de consultas seja uma função linear em m, é possível obter resultados ainda significativos a partir do teste reduzido.

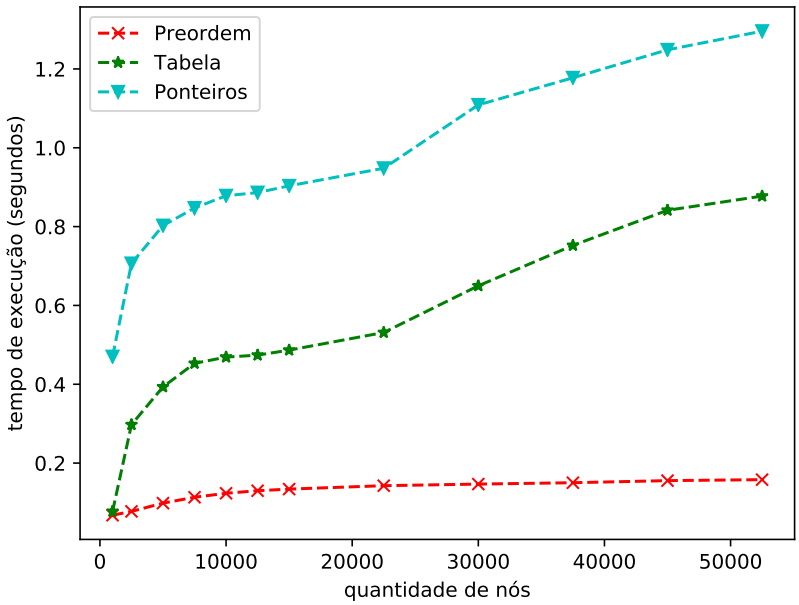


Figura 3.5: ***Resultados para o teste reduzido de consultas em árvores lineares.*** *Para efeito de comparação, o tempo esperado para o Algoritmo Trivial executar 10M consultas para o maior tamanho de árvore deste teste é de aproximadamente 7 minutos e por isso nem está no gráfico.*

Árvores binárias

Os testes de consultas em árvores binárias evidenciaram a grande melhora que o Algoritmo Trivial consegue obter, já que, ao limitar a profundidade das árvores testadas em $O(\log_2 n)$, cada uma de suas consultas também é realizada em tempo logarítmico. Já o Algoritmo da Pré-ordem e o Algoritmo dos Ponteiros têm performances semelhantes, porém, o segundo leva a melhor. O Algoritmo da Tabela, como esperado, se mantém constante e extremamente rápido.

Vale lembrar também que esses resultados correspondem à realização de 10 milhões de consultas, ou seja, até mesmo o Algoritmo Trivial é capaz de realizar uma única consulta em tempos quase insignificantes (na ordem de 9.25e−07 segundos).

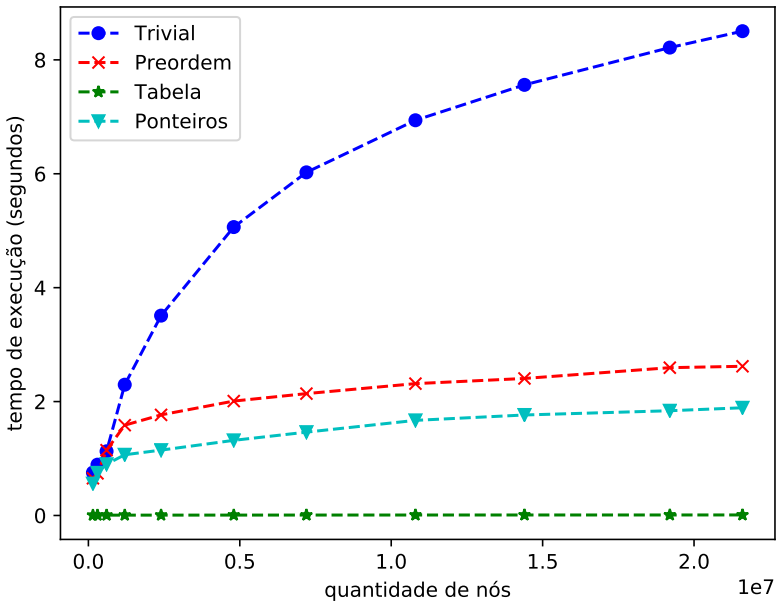


Figura 3.6: *Resultados para as consultas em árvores binárias.*

Árvores quaternárias

Aqui, o Algoritmo Trivial se torna ainda mais eficiente devido à limitação da profundidade das árvores em $O(\log_4 n)$, ficando muito mais próximo dos seus concorrentes e ficando abaixo de de 5 segundos para 10 milhões de consultas no maior tamanho de árvore testado. Tanto o Algoritmo da Pré-ordem e o Algoritmo dos Ponteiros se mantiveram praticamente iguais ao teste anterior, provavelmente devido às suas constantes já serem suficientemente pequenas para árvores binárias.

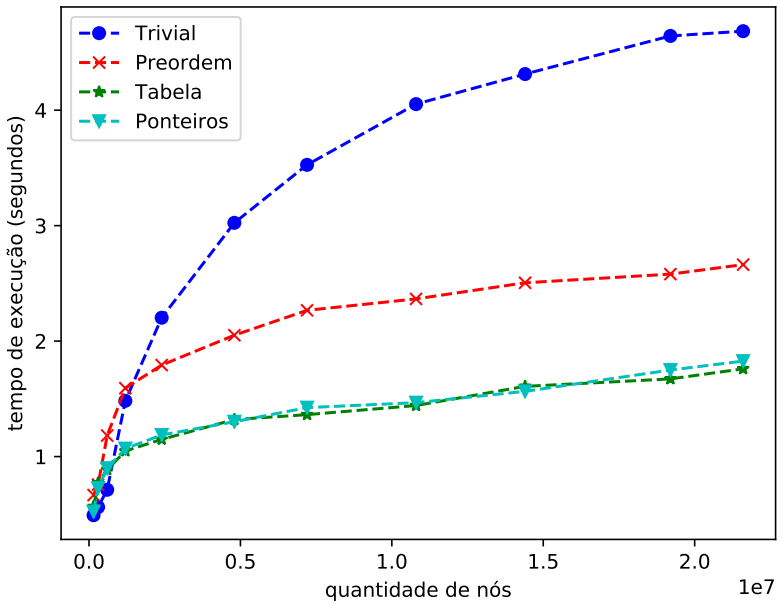


Figura 3.7: *Resultados para as consultas em árvores quaternárias.*

3.3 Uso de memória

Uma parte muito importante da comparação dos algoritmos implementados é analisar seu consumo de memória, já que além de ser um fator proibitivo para muitos computadores pessoais também pode influenciar a performance de um algoritmo caso fique no limite da memória disponível. Os valores presentes nas tabelas abaixo foram obtidos através do programa `/usr/bin/time -v` do Linux, exceto para o Algoritmo da Tabela no caso das árvores lineares.

3.3.1 Árvores lineares

Observando a quantidade necessária de memória para preprocessar árvores lineares com cada um dos algoritmos pode-se ver porque o Algoritmo da Tabela é inviável para este caso do problema, já que atingiria quantidades absurdas devido à sua complexidade quadrática. O Algoritmo dos Ponteiros apresenta uma quantidade de memória utilizada maior que o Algoritmo da Pré-ordem, como esperado, já que o primeiro tem uma constante que multiplica sua complexidade.

Nós	Trivial	Tabela *	Ponteiros	Pré-ordem
150K	0.017	44	0.034	0.028
300K	0.031	176	0.066	0.053
600K	0.059	704	0.129	0.104
1.2M	0.11	2816	0.256	0.206
2.4M	0.22	11264	0.546	0.410
4.8M	0.45	45056	1.09	0.818
7.2M	0.67	101376	1.63	1.12
9.6M	0.90	228096	2.17	1.63
10.8M	1.01	513216	2.44	1.75
14.4M	1.35	1154736	3.26	2.25
19.2M	1.80	2598156	4.35	3.26
21.6M	2.02	3259126	4.89	3.50

Tabela 3.1: ***Uso de memória (em GB) de cada algoritmo para preprocessar árvores lineares.*** *Os valores do Algoritmo da Tabela foram estimados, já que não é possível rodá-lo para estes tamanhos. Além disso, é importante notar que está incluso nestes valores a árvore em si, já que mesmo o Algoritmo Trivial, que não realiza preprocessamento nenhum, ainda precisa acessar a estrutura da árvore.*

3.3.2 Árvores binárias

Com os testes restringidos a árvores binárias, o Algoritmo da Tabela consegue realizar o preprocessamento sem estourar a memória da máquina utilizada, mas ainda apresenta uma quantidade significativamente maior que os outros algoritmos, o que pode se tornar um problema eventualmente para casos extremos. Além disso, todos os outros algoritmos também apresentaram um uso reduzido de memória, como esperado.

Nós	Trivial	Tabela	Ponteiros	Pré-ordem
150K	0.014	0.029	0.029	0.016
300K	0.026	0.057	0.042	0.030
600K	0.050	0.115	0.082	0.057
1.2M	0.096	0.232	0.162	0.112
2.4M	0.190	0.467	0.322	0.221
4.8M	0.377	0.945	0.640	0.439
7.2M	0.565	1.45	0.959	0.670
9.6M	0.753	1.95	1.27	0.877
10.8M	0.846	2.21	1.43	0.989
14.4M	1.12	2.97	1.91	1.33
19.2M	1.50	3.98	2.55	1.75
21.6M	1.69	4.48	2.87	1.97

Tabela 3.2: ***Uso de memória (em GB) de cada algoritmo para preprocessar árvores binárias.*** *Note que está incluso nestes valores a árvore em si, já que mesmo o Algoritmo Trivial, que não realiza preprocessamento nenhum, ainda precisa acessar a estrutura da árvore.*

3.3.3 Árvores quaternárias

Nestes testes, é possível observar uma pequena redução da memória utilizada pelo Algoritmo Trivial e pelo Algoritmo da Pré-ordem, apesar de seu preprocessamento não depender do fator de ramificação da árvore e o Algoritmo dos Ponteiros viu um ganho também pequeno, já que a mudança de $n \log_2(\log_2 n)$ para $n \log_4(\log_4 n)$ não é tão expressiva. Isto indica que, para fatores de ramificação maiores ainda, a quantidade de memória utilizada pelos algoritmos deve se estabilizar não muito longe dos valores desta tabela.

Nós	Trivial	Tabela	Ponteiros	Pré-ordem
150K	0.014	0.024	0.022	0.016
300K	0.025	0.046	0.041	0.029
600K	0.047	0.093	0.080	0.055
1.2M	0.092	0.189	0.157	0.109
2.4M	0.181	0.381	0.312	0.214
4.8M	0.354	0.766	0.622	0.429
7.2M	0.537	1.15	0.931	0.632
9.6M	0.715	1.53	1.24	0.847
10.8M	0.804	1.72	1.39	0.966
14.4M	1.07	2.30	1.85	1.31
19.2M	1.42	3.07	2.47	1.69
21.6M	1.60	3.45	2.78	1.87

Tabela 3.3: ***Uso de memória (em GB) de cada algoritmo para preprocessar árvores quaternárias.*** *Note que está incluso nestes valores a árvore em si, já que mesmo o Algoritmo Trivial, que não realiza preprocessamento nenhum, ainda precisa acessar a estrutura da árvore.*

3.4 Conclusões e trabalho futuro

Analisando os resultados fica evidente o quão importante é conhecer tanto a aplicação em questão quanto os recursos disponíveis, já que cada algoritmo tem méritos e defeitos que dependem do formato das árvores. O Algoritmo da Tabela é uma ótima escolha para aplicações em que a quantidade de consultas pesa muito mais do que o tempo de preprocessamento, já que este abre mão de performance *a priori* para responder as consultas o mais rápido possível; entretanto, também possui um custo proibitivo de memória conforme o fator de ramificação diminui, que pode torná-lo inviável até mesmo para *workstations* com grandes quantidades de memória. O Algoritmo Trivial se torna mais viável conforme o fator de ramificação aumenta e tem custo zero de memória e tempo de preprocessamento, porém não lida muito bem com seu pior caso de árvores lineares, levando tempos que tornariam qualquer aplicação ineficaz conforme o tamanho das árvores aumentam. Tanto o Algoritmo dos Ponteiros quanto o Algoritmo da Pré-ordem trazem soluções elegantes com boas complexidades tanto para o preprocessamento quanto para as consultas, porém de mais difícil compreensão; o primeiro leva a melhor nas consultas (exceto no caso de árvores lineares) ao passo que o segundo conta com o melhor preprocessamento de todos (a menos do Trivial, que não faz nada).

Numa continuação deste trabalho, seria interessante expandir o estudo para o caso do Problema do Ancestral de Nível **dinâmico**, no qual a árvore do problema pode ser modificada em tempo de execução. Logo de cara é fácil perceber que o Algoritmo Trivial funcionaria sem nenhuma adaptação, enquanto os outros teriam que gastar algum tempo reprocessando a árvore, o que leva a mais discussões interessantes. Além disso, seria bom testar os algoritmos com árvores não completas, talvez obtendo comparações mais condizentes com casos reais das aplicações do problema.

Apêndice A

Testes de unidade com a biblioteca Catch2

Para garantir a corretude dos algoritmos implementados neste trabalho, já estava decidido desde o início a utilizar alguma ferramenta para realizar testes que pudessem facilmente identificar erros nas implementações ao longo do ano. Depois de pesquisar bastante a biblioteca escolhida foi a **Catch2** por ser *header-only* e facilitar o processo de rodar em outras máquinas sem precisar instalar nada. A sintaxe do *framework* fornecido pela biblioteca é bem organizada e foi de fácil utilização. No programa A.1 consta uma parte do código real dos testes do Algoritmo da Tabela, exemplificando quão simples é escrever testes de unidade bem separados por classe e tipo de teste, podendo criar diversos casos de teste, cada um com um *setup* diferente de variáveis e objetos.

Para este estudo, cada algoritmo teve seus testes encapsulados dentro de casos de teste (`TEST_CASE`) enquanto cada tipo de teste dentro de cada algoritmo (árvores lineares, binárias e quaternárias) foram colocados dentro de suas próprias seções (`SECTION`). As asserções, que fazem o papel de garantir que algo realmente vale é feito através das macros `REQUIRE`, que checa igualdade entre dois valores e `REQUIRE_THROW_AS`, que verifica se uma função levantou a exceção que era esperada.

Programa A.1 Parte dos testes para o Algoritmo da Tabela.

```
1  #include "../include/catch.hpp"
2  #include "../../generators/generators.hpp"
3  #include "../include/TestUtils.hpp"
4  #include "../../code/include/TableAlgorithm.hpp"
5
6  TEST_CASE ("Table algorithm", "[table]") {
7      SECTION ("Binary Tree") {
8          int n = 2178;
9          vector<Node*> nodes;
10         build_balanced_kary_tree(n, 2, nodes);
11         Tree *tree = new Tree(nodes.size(), nodes[0]);
12         TableAlgorithm *table = new TableAlgorithm(tree);
13
14         SECTION ("Has a query function") {
15             for (int node = 0; node < n; node++) {
16                 for (int depth = 0; depth <= tree->depth(node); depth++) {
17                     REQUIRE(table->query(node, depth) == naive_check(tree, node, depth
                           ));
18                 }
19             }
20             SECTION ("Query function returns -1 if there is no answer") {
21                 REQUIRE(table->query(1, tree->size()) == -1);
22                 REQUIRE(table->query(1, tree->size()) == -1);
23             }
24
25             SECTION ("Query function throws if negative depth") {
26                 REQUIRE_THROWS_AS(table->query(1, -1), std::invalid_argument);
27             }
28
29             SECTION ("Query function throws if invalid node") {
30                 REQUIRE_THROWS_AS(table->query(-1, 0), std::invalid_argument);
31                 REQUIRE_THROWS_AS(table->query(tree->size(), 0), std::invalid_argument)
                       ;
32             }
33         }
34     }
35 }
```

Referências

[*Catch2* s.d.] *Catch2*. URL: https://github.com/catchorg/Catch2.

[Bender e Farach-Colton 2002] Michael A. Bender e Martin Farach-Colton. "The level ancestor problem simplified". Em: *LATIN*. 2002 (citado na pg. 8).

[Euler 1741] Leonhard Euler. "Solutio problematis ad geometriam situs pertinentis". Em: *Commentarii Academiae Scientiarum Imperialis Petropolitanae* 8 (1741), pgs. 128–140 (citado na pg. 2).

[Farach e Muthukrishnan 1996] Martin Farach e S. Muthukrishnan. "Perfect hashing for strings: formalization and algorithms". Em: *Combinatorial Pattern Matching*. Ed. por Dan Hirschberg e Gene Myers. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pgs. 130–140. ISBN: 978-3-540-68390-2 (citado na pg. 8).

[Geary *et al.* 2006] Richard F. Geary, Rajeev Raman e Venkatesh Raman. "Succinct ordinal trees with level-ancestor queries". Em: *ACM Trans. Algorithms* 2.4 (out. de 2006), pgs. 510–534. ISSN: 1549-6325. DOI: 10.1145/1198513.1198516. URL: http://doi.acm.org/10.1145/1198513.1198516 (citado na pg. 8).

[Hopkins e Wilson 2004] Brian Hopkins e Robin J. Wilson. "The truth about königsberg". Em: *The College Mathematics Journal* 35.3 (2004), pgs. 198–207. DOI: 10.1080/07468342.2004.11922073. eprint: https://doi.org/10.1080/07468342.2004.11922073. URL: https://doi.org/10.1080/07468342.2004.11922073 (citado na pg. 2).

[Menghani e Matani 2019] Gaurav Menghani e Dhruv Matani. *A Simple Solution to the Level-Ancestor Problem*. 2019. arXiv: 1903.01387 [cs.DS] (citado na pg. 8).

[Papamichail *et al.* 2014] Dimitris Papamichail, Thomas Caputi e Georgios Papamichail. "The level ancestor problem in practice". Em: (fev. de 2014) (citado na pg. 8).

[Sadakane e Grossi 2006] Kunihiko Sadakane e Roberto Grossi. "Squeezing succinct data structures into entropy bounds". Em: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*. SODA '06. Miami, Florida: Society for Industrial e Applied Mathematics, 2006, pgs. 1230–1239. ISBN: 0-89871-605-5. URL: http://dl.acm.org/citation.cfm?id=1109557.1109693 (citado na pg. 8).

[Yuan e Atallah 2009] Hao Yuan e Mikhail J. Atallah. "Efficient data structures for range-aggregate queries on trees". Em: *Proceedings of the 12th International Conference on Database Theory*. ICDT '09. St. Petersburg, Russia: ACM, 2009, pgs. 111–120. ISBN: 978-1-60558-423-2. DOI: 10.1145/1514894.1514908. URL: http://doi.acm.org/10.1145/1514894.1514908 (citado na pg. 8).