# A tribo bárbara

Lucas Dimer Justo

Engenharia de Software — PUCRS

3 de abril de 2019

**Resumo**

Este relatório apresenta o processo de desenvolvimento de um algoritmo para resolver o primeiro problema proposto na disciplina Algoritmos e Estruturas de Dados II do 3º semestre do curso de Engenharia de Software. O problema trata-se de uma equipe de antropólogos que solicitou ajuda para, através de pergaminhos antigos, entender o comportamento de uma tribo bárbara e assim responder à pergunta: "Qual dos guerreiros bárbaros que não tem filhos, tem o maior número de terras?". Então, após entender que os guerreiros podiam conquistar terras em vida e assim que morressem transferiam as suas terras igualmente entre seus filhos, montamos uma árvore genérica para associar pais com filhos, e dividir suas terras. Assim, obtemos a informação precisa de quem tem filho, quem não tem e quantas terras exatamente cada indivíduo possui, para então percorrer a árvore e descobrir qual dos guerreiros satisfaz a condição de não ter filhos e ter o maior número de terras nesta categoria (categoria de ser nodo folha).

# Introdução

Antropólogos encontraram pergaminhos que descrevem o comportamento de uma antiga tribo bárbara. Com estes pergaminhos, conseguimos descobrir que eles contabilizam suas terras de acordo com as seguintes regras:

1. As terras de um guerreiro são divididas igualmente entre os filhos quando ele morre;

2. Naturalmente, um filho pode acumular mais terras através de conquista, compra, casamento, etc , durante sua vida e com isso ele deixará mais (ou menos) terras para seus filhos.

A equipe nos enviou um arquivo de texto com dados no seguinte formato:

- Primeira linha: número de terras do primeiro guerreiro (aquele que deu origem a toda família);
- Todas linhas seguintes: NomeDoPai NomeDoFilho TerrasDesteFilho.

Exemplo de formato do arquivo:

103787

Thorgestax Deldriralex 4626

Thorgestax Delrenmax 6080

Delrenmax Diormanlox 4736

Delrenmax Neppanpix 2249

Com este arquivo ao nosso dispor, devemos fazer um algoritmo, que leia os dados do arquivo e distribua as terras para cada guerreiro, de acordo com os comportamentos da tribo. Assim podemos percorrer esses números de terra de cada indivíduo e ver quais dos guerreiros sem filho, tem o maior número de terras.

# Desenvolvendo a solução

A primeira abordagem para resolver o problema, foi tentar percorrer o arquivo linha por linha e montar uma árvore com estas informações da seguinte forma:

```
void montandoAvore() {
      terrasInicial = scanner.proximoDado()
      enquanto(houverDadosNoArquivo) {
            nomeDoPai = scanner.proximoDado()
            nomeDoFilho = scanner.proximoDado()
            terrasDoFilho = scanner.proximoDado()
            arvore.add(nomeDoPai,nomeDoFilho,terrasDoFilho)
      }
}
void arvore.add(String paiN, String filhoN, double terras){
      Nodo filho = novo Nodo(filhoN,terras)
```

```
        Nodo pai = arvore.procura(paiN)
        pai.filhos.adicionaNaLista(filho)
    }
```

Porém este método de montar árvore lançava uma exceção (NullPointerException) em alguns casos, pois não é garantido que, se na linha N (do arquivo) adicionamos fulano, qualquer linha antes de N o pai de fulano já foi adicionado. Com isso as vezes ao usarmos o método procura(nomeDoPai) ele retornava um pai nulo, pois não achava ninguém. Então ao tentarmos adicionar um filho em uma lista de filhos inexistente de um pai também inexistente obtínhamos um erro.

Nisto reparamos que a abordagem para montar a árvore genérica não pode ser tão simples, e optamos por usar estruturas auxiliares para ajudar no processo de montagem da mesma. Então ao pensarmos muito para tentar escolher estas estruturas auxiliares e não chegarmos à uma conclusão muito boa, optamos por criar um caso de teste muito simples, com uma árvore genérica que representa uma família de apenas 5 membros. Em seguida realizamos o processo de montagem manualmente (em uma folha de papel), para notar quais os passos necessários para montar uma árvore dessas e consequentemente escolher as melhores estruturas para realização destas ações.

As 5 linhas de entrada escolhidas para ajudar nesta etapa foram as mesmas mostradas anteriormente na introdução para ajudar no entendimento do modelo de arquivo recebido:

103787

Thorgestax Deldriralex 4626

Thorgestax Delrenmax 6080

Delrenmax Diormanlox 4736

Delrenmax Neppanpix 2249

A resposta desse caso de teste criado é Deldriralex com 56519.5 terras (foi utilizado um valor real para armazenar terras, para lidar com as divisões de terras entre pais e filhos sem perda de valor). Este resultado foi encontrado primeiramente manualmente em uma folha de papel e depois confirmado pelo computador ao rodá-lo com o algoritmo final.

Ao realizar o processo de montagem manualmente ficou claro que a melhor abordagem seria, ao em vez de usar o nomeDoPai, nomeDoFilho e terrasDoFilho diretamente no método adicionar da árvore, utilizaríamos estas variáveis para criar uma Lista de Strings com o nome de cada pai (lista foi escolhida aqui por que é uma estrutura de dados com fácil iteração), um HashMap cujo a chave é o nomeDoPai e o valor é uma Lista de Filhos (Filho foi utilizado aqui

como um objeto que guarda os atributos: nomeDoFilho, nomeDoPai, terras) e outro HashMap cujo a chave é o nomeDoFilho e o valor é o objeto Filho (os dois HashMaps foram escolhidos para suas respectivas tarefas por que as únicas operações efetuadas sobre eles neste algoritmo seriam adição e pesquisa, as duas muito boas neste tipo de estrutura).

# Concluindo a solução

Então após repetir aquele laço do método montandoArvore() só que ao em vez de usarmos as variáveis no arvore.adicionar() usarmo-las para montar as estruturas definidas no parágrafo anterior, criamos um objeto ArvoreDeGuerreiros que recebe como parâmetros no construtor as seguintes variáveis:

- HashMap Filhos: chave = nome do filho; valor = objeto filho
- ListaPais: nome de todos os pais
- HashMap PaisEFilhos: chave = nome do pai; valor = lista de filhos deste pai
- Nome do pai que gerou a árvore
- Número de terras do pai que gerou a árvore

Segue abaixo o método para achar o pai que gerou a árvore e todos os outros métodos de ArvoreDeGuerreiros (que tem um objeto Nodo com os atributos: String nome, double terras, ListaNodo filhos):

```
//se um pai não existir no hashmap de filhos é por que ele é o pai gerador (todos os outros pais são filhos de alguém presente na árvore também), se não houver nenhum pai que respeite essa condição a árvore não tem apenas um ou algum gerador (o que não acontece nos nossos casos de teste).
      String acheOPaiGerador(ListaPais lp, HashMapFilhos hmf) {
      Para cada nomeDePai em lp {
            Se (hmf.get(nomeDePai)==null) {
                  Return nomeDePai
            }
      }
      Return null
}

//converte para lista de nodos é um método para converter as listas de filhos para listas de nodos (usado para montar a arvore).
ListaDeNodos converteParaListaDeNodos(ListaDeFilhos l) {
      ListaDeNodos resposta = nova ListaDeNodos()
```

```
      Para cada Filho f em l {
            resposta.add(novo Nodo(f.nome,f.terras))
      }
      Return resposta
}
```

//montarArvore() é um método recursivo que é chamado na main com a raiz de parâmetro (o filho da raiz é o pai gerador da árvore). Neste método lembre-se que PaisEFilhos é o HashMap cujo a chave é o nomeDoPai e o valor é a ListaDeFilhos deste pai.

```
void montarArvore(Nodo n) {
      Para cada Nodo p em n.filhos {
            Se (PaisEFilhos.get(p.nome)!=null) {
                  p.filhos =
                  converteParaListaDeNodos(PaisEFilhos.get(p.nome))
            }
            montarArvore(p)
      }
}
```

//atribuirTerras() é um método recursivo que é chamado na main com o pai gerador da árvore como parâmetro e distribui as terras entre os guerreiros, seguindo as regras citadas na introdução deste relatório. Neste método lembre-se que PaisEFilhos é o HashMap cujo a chave é o nomeDoPai e o valor é a ListaDeFilhos deste pai. Este método é chamado após o método montarArvore() ja ter sido chamado.

```
void atribuirTerras(Nodo n){
      Para cada Nodo p em n.filhos {
            p.terras += n.terras/PaisEFilhos.get(n.nome).size()
            atribuirTerras(p)
      }
}
```

//acharResposta() é o método recursivo chamado na main (com a raiz de parâmetro) após montar a árvore e atribuir as terras aos guerreiros, ele serve para achar qual dos guerreiros sem filhos (nodos folhas) tem o maior número de terras. Este método trabalha com duas variáveis

externas (variáveis globais da classe ArvoreDeGuerreiros) que são Nodo guerreiroResposta e double maiorTerras. Estas duas variáveis são globais por que variáveis de controle em métodos recursivos não dão muito certo, pois são recriadas com valor novo a cada chamada recursiva.

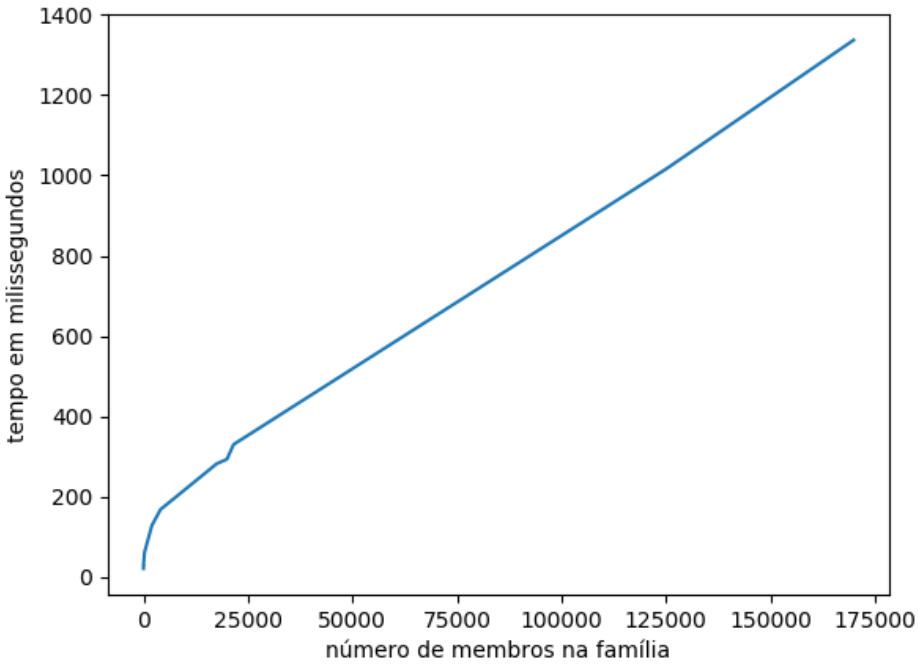
```
Nodo acharResposta(Nodo r){
      Para cada Nodo n em r.filhos{
            Se (n.filhos.size()==0){
                  Se (n.terras > maiorTerras){
                        maiorTerras = n.terras
                        guerreiroResposta = n
                  }
            }
            acharResposta(n)
      }
      Return guerreiroResposta
}
```

# Resultados

Ao terminar de elaborar o algoritmo e implementa-lo em Java, obtivemos os resultados da seguinte tabela (ordenados crescentemente por tamanho da família):

| Nome do caso teste | Nome do guerreiro | Número de terras | Tempo em milissegundos | Tamanho da família |
|---|---|---|---|---|
| enunciado | Alteetoflex | 26799.4 | 27 | 16 |
| casoJB4b | Nepulacritrikix | 12629.9 | 31 | 30 |
| casoJB4a | Rimeemicpox | 25307.0 | 31 | 31 |
| casoJB8a | Cristiumimonlax | 21531.0 | 61 | 239 |
| casoJB8b | Stropunngemonstax | 16457.42 | 128 | 2004 |
| casoJB10b | Delrenvax | 11624.84 | 168 | 4071 |
| casoJB10a | Nepstitrigax | 13964.85 | 282 | 17524 |
| casoJB12b | Prepeeblepscapox | 14002.88 | 293 | 19954 |
| casoJB12a | Gruvervix | 12579.92 | 330 | 21569 |
| casoJB14b | Gruumimontex | 11857.98 | 1017 | 125197 |
| casoJB14a | Carmanclox | 12181.94 | 1337 | 169908 |

Com as colunas de tempo em milissegundos e tamanho da família (número de pessoas), montamos um gráfico para mostrar o desempenho do algoritmo relacionado ao tamanho da entrada (tamanho da família). Para montar o gráfico foi utilizado a classe pyplot da biblioteca matplotlib na linguagem de programação Python.




Este gráfico nos permite constatar que mesmo quando o número de membros na família aumenta drasticamente, o algoritmo continua achando o resultado rapidamente. Tanto que, a diferença de tempo de resposta de um caso com 16 membros na família para um caso com 169908 membros na família é de apenas 1310 milissegundos. Considerando que um computador não demora sempre o mesmo tempo para achar a resposta do mesmo problema com o mesmo algoritmo, continua sendo um tempo aceitável.

# Conclusões

O processo de início do desenvolvimento da solução, mostra que embora a maneira de acha-la nem sempre seja a mais simples possível, ainda pode-se achar métodos simples que o ajude a montar uma solução mais complexa (em relação à solução anterior que não havia funcionado). Isto se demonstra na hora em que a tentativa de solução de simplesmente achar o pai na árvore e ligar o filho a ele falhou graças ao formato da entrada, e a saída para isto foi criar um caso de teste simples para analisar como realmente resolver o problema.

Acreditamos ter desenvolvido uma solução de fácil compreensão e implementação, que não demanda de grande espera para obtenção de resultados. Isto é observável na seção de resultados ao analisarmos a coluna de tempo em milissegundos da tabela de casos testes e na seção concluindo a solução onde mostra-se os métodos curtos e simples que juntos compõem o algoritmo.