# Relatório do Trabalho 2 de Algoritmos e Estrutura de Dados II

Dhruv Babani[*], Bernardo Balzan[†]
Faculdade de Ciência Da Computação — PUCRS

13 de novembro de 2023

**Resumo**

Este artigo descreve a solução para o segundo problema proposto na disciplina de Algoritmos e Estrutura de Dados II, no terceiro semestre, que trata de uma rede de moléculas que se dão a origem ao elemento "Ouro". É apresentado apenas uma solução para o problema e sua eficiência é analisada. Em seguida as representações obtidas são mostradas.

# 1 Introdução

Dentro do escopo da disciplina de Algoritmos e Estrutura de Dados II, o segundo problema "Os alquimistas se reúnem", pode ser resumido assim: Apresenta-se um evento chamado "Grande Convenção dos Alquimistas", que ocorre a cada 100 anos, com o objetivo de conceder o prêmio da Asinha de Morego Dourada para melhor ideia de receita para produzir Ouro. Nesse sentido, a sua formação ocorre através de várias transformações de elementos químicos, mas com um, porém, a primeira mutação deve começar com o hidrogênio e vão usando quantidades diferentes para produzir outros elementos e assim por diante.

O problema a ser resolvido é escrever um algoritmo que apresente o número de hidrogênios utilizados para produzir uma unidade de ouro a partir dos casos de teste(receitas) disponibilizados pelos candidatos do evento.

Para resolver o problema proposto, fizemos o algoritmo na linguagem Java. Além disso, foram utilizados códigos para leitura de arquivo e contagem de tempo na solução resultante, que não afetam o desempenho do algoritmo. Em seguida os resultados obtidos serão apresentados, bem como as conclusões obtidas no decorrer do trabalho.

# 2 Resolução do Problema

## 2.1 Leitura de Arquivo

Primeiramente, tivemos que transformar o conteúdo, que estava nos casos de teste, em um grafo direcionado. Então, o primeiro passo foi criar as classes Edge e Grafo. Na classe Edge, foi inicializado um construtor com os seguintes atributos:

[*]d.babani001@edu.pucrs.br
[†]b.balzan@edu.pucrs.br

- String v -> vértice de origem
- String w -> vertice de destino
- int weigth -> peso

Já na classe Grafo, foi criado um dicionário de String(Key) e List<Edge>(value).Em seguida, foi implementado o método "addToList", onde armazena uma lista de arestas dentro do dicionário, e no final ela retorna essa lista. Depois disso, é elaborado um método do tipo VOID para apenas adicionar as arestas, que na qual, é usado no código de leitura de arquivo apresentado aqui abaixo:

```
Leitura de Arquivo
    função criarGrafo(filename):
    grafo = novoGrafo()
    arquivo = abrirArquivo(filename)

    enquanto houver linhas no arquivo:
        linha = lerLinha(arquivo)
        controlador = 0
        arestas = dividirString(linha , " ")
        verticeOrigem = arestas[últimoElemento(arestas)]

        enquanto arestas[controlador] não for igual a " → ":
            verticeDestino = arestas[controlador + 1]
            peso = converterParaInteiro(arestas[controlador])
            adicionarAresta(grafo , verticeDestino , verticeOrigem , peso)

            controlador = controlador + 2

    fecharArquivo(arquivo)
```

Depois de fazer a leitura de arquivo e ter transformado para um formato de grafo, seguimos para resolução do problema.

## 2.2 O algoritmo

Como estávamos analisando um grafo, tivemos que optar por um tipo de caminhamento. Como o vértice de origem sempre seria o elemento hidrogênio, e deveria percorrer todas as suas conexões(arestas) possíveis, logo optamos pelo caminhamento de busca em profundidade, conhecido também como, DFS. Com isso, implementamos o algoritmo dentro da classe Grafo, para ter acesso direto ao conteúdo armazenado do grafo.

Nesse sentido, o algoritmo para resolver o problema, teve a necessidade apenas de um método. Sabendo disso, o método se divide em cinco partes:

1. Parâmetro de Entrada:
    - O método "calcularHidrogenio"recebe uma String v como parâmetro. Essa String representa a vértice de origem do grafo, que no caso sempre seria o "Hidrogênio"
2. Condição inicial
    - Verifica se a String v é igual a "ouro". Se for, retorna BigInteger.ONE. Isso sugere uma condição de parada ou caso base para a recursão. Ou seja, se o vértice for "ouro", a função retorna 1.

3. Variáveis e Estrutura de Dados

    - Declara uma variável resultado do tipo BigInteger e a inicializa com zero. Esta variável será usada para acumular o resultado final.
    - Obtém a lista de arestas associadas ao vértice v do grafo, armazenadas na variável listaFilhos.

4. Iteração Recursiva

    - Itera sobre as arestas da lista listaFilhos usando um loop for-each. Para cada aresta, obtém o peso da aresta (valorAresta) e o valor calculado recursivamente para o vértice de destino da aresta (valorFilho) chamando recursivamente a função calcularHidrogenio com o vértice de destino (edge.getW()).
    - Atualiza o resultado acumulado multiplicando o valorAresta pelo valorFilho e somando ao resultado.

5. Retorno

    - Retorna o resultado final calculado para o vértice v.

Como pode ser observado, o algoritmo se apresenta a ser bem simples e objetivo. Para o entender melhor, apresentaremos o seu pseudo-código a seguir:

```
1  Algoritmo
2      função calcularHidrogenio(v: String) → BigInteger:
3      se v for igual a "ouro" então
4          retornar BigInteger.ONE
5      fim se
6
7      resultado = BigInteger.ZERO
8      listaFilhos = graph.get(v)
9
10     para cada edge em listaFilhos faça
11         valorAresta = BigInteger.valueOf(edge.getWeight())
12         valorFilho = calcularHidrogenio(edge.getW())
13         resultado = resultado + (valorAresta ∗ valorFilho)
14     fim para
15
16     retornar resultado
17 fim função
```

# 3 Resultados

Depois de implementar o algoritmo acima em linguagem Java e executá-lo com os casos testes, obtivemos os seguintes resultados para cada caso teste, sendo o número de Hidrogênios depois de percorrer todos os caminhos, o tempo em segundos até que o algoritmo gerasse a resposta correta, como apresentado abaixo:

| Caso Teste | Numero de Hidrogênios | Tempo em Segundos |
|---|---|---|
| Casoteste | 16272 | 0.001692 |
| Casoc80 | 27221484395 | 0.006283583 |
| Casoc120 | 2257966552765316 | 0.012662 |
| Casoc180 | 1295127372563879647489923 | 0.179670708 |
| Casoc240 | 55577785066027869882239842 | 0.559684416 |
| Casoc280 | 25043936631358540492332040 | 3.346228833 |
| Casoc320 | 17986055867306301215241957896478 | 6.998456834 |
| Casoc360 | 2351586747225127921295158981670930011 56 | 274.017651208 |

Após a obtenção destes valores, estes foram testados com ferramentas capazes de lidar com grafos de grande escala e todos os resultados foram confirmados corretos.

# 4 Análise de Eficiência

Como pode ser observado na secção de resultados, os últimos casos começaram a levar mais tempo em relação ao seus anteriores. Isso ocorre devido ao uso de objetos BigInteger, que é criado em cada chamada recursiva que é realizada. Sabendo disso, isso pode resultar em uma quantidade significativa de alocação de memória, o que explica o tempo levado para o processamento do resultado. Logo, a complexidade do algoritmo analisado é de O(n), considerando o que foi dito anteriormente.

# 5 Conclusões

Á nossa perspectiva, apresentamos certas dificuldades na implementação desde da leitura de arquivo até o algoritmo em si, devido ao uso de uma estrutura de dados avançada, que no caso foi Grafos. Todavia, após bastante estudo e dedicação, conseguimos elaborar uma solução válida para o problema dito, como foi mostrada na secção 2 do Relatório.

Portanto, concluímos que no mundo em que vivemos, existem séries de problemas que estão sendo resolvidos ou estão pendentes ainda. Então, o que nós quisemos dizer, que esses problemas podem ser resolvidos com algoritmos que fazem o uso de estrutura de dados, como fundamento, para chegar na solução. Dito isso, acreditamos que nós estaríamos aptos a resolver esses problemas mencionados, com a certa autoridade de escolher a melhor solução de estrutura de dados para o cenário problemático. Logo, podemos dizer que atingimos o objetivo da disciplina de "Algoritmos e Estrutura de Dados II".