Relatório T1 – O DNA D/N/A

Gabriel P. Teiga[1]

Escola Politécnica – PUCRS

22 de novembro de 2023

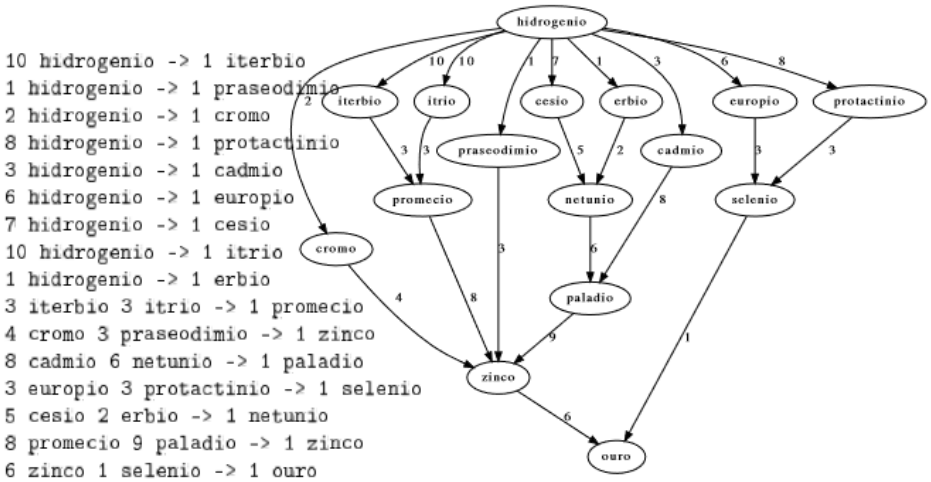
Resumo

O artigo descreve a solução para o trabalho T2 na disciplina de Algoritmos e Estrutura de Dados II. A partir de uma receita de alquimia, partindo sempre do elemento Hidrogênio, propõe transformar várias unidades em elementos diferentes até chegarmos em uma unidade de ouro. Os resultados foram satisfeitos, respeitando os pré-requisitos, explicando detalhadamente o porquê o algoritmo é o mais eficiente possível e com seus respectivos resultados.

Introdução

O desafio chamado de "Os alquimistas se reúnem", é baseado em um cenário fictício. O problema central consiste na otimização do processo alquímico para a produção de ouro, envolvendo a análise de receitas complexas trocadas durante a Grande Convenção dos Alquimistas. A ênfase recai na determinação da quantidade de hidrogênio necessária para transformar uma unidade de ouro, levando em consideração a crescente complexidade das receitas ao longo dos séculos. A complexidade decorre da transformação de elementos químicos, iniciando-se com hidrogênio e variando em quantidades para produzir ouro. Um exemplo pode ser descrito na imagem abaixo:

Exemplo de receita



O desafio é claro: determinar a quantidade exata de hidrogênio necessária para realizar cada receita, proporcionando uma visão clara dos custos associados. Hidrogênio, embora abundante, não é gratuito, e compreender a eficiência na utilização desse

[1] gabrielteiga99@gmail.com

elemento crucial torna-se vital para o sucesso das empreitadas alquímicas. Neste relatório, será apresentada a solução baseada em grafos, que promete simplificar e otimizar a análise das complexas receitas alquímicas. Para a resolução do nosso problema, dividimos o restante do nosso relatório nos seguintes módulos:

- Estrutura de dados – Onde iremos comentar sobre a estrutura que conseguirá realizar as funções que estarão presentes em nosso algoritmo.
- Algoritmo – O algoritmo que irá ser utilizado como base para a implementação da solução em uma linguagem de programação e sua implementação em si.
- Resultados – Demonstração dos resultados, com tabelas e gráficos, do algoritmo implementado na linguagem de programação Java.
- Conclusão – Finalização do nosso relatório, compilando análises feitas nos módulos anteriores.

Estrutura de dados

Considerando o problema, vamos moldar nossa solução. Antes de apresentarmos o algoritmo em si, temos que pensar em nossa estrutura de dados que será utilizada. A escolha tem que ser assertiva, já que teremos inúmeras transformações do hidrogênio em outros elementos, de tamanho *n*. A solução que imaginamos imediatamente é a utilização de grafos. O exemplo de receita já nos mostra um *grafo*, basta implementá-lo e realizar algumas adaptações para a nossa solução.

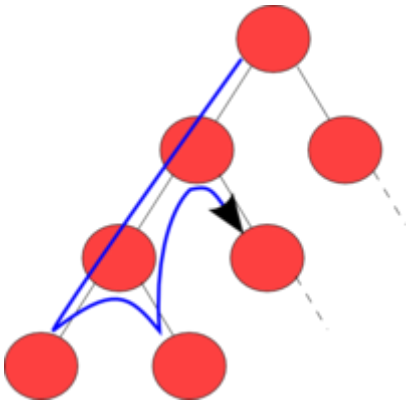
O *grafo* irá armazenar *vértices* e *arestas*, podendo caminhar nele por meio de *vértices* adjacentes ao *vértice* inicial, o hidrogênio. Cada *vértice* terá sua *lista de arestas* que apontarão para um *vértice* destino e seu *peso* de transformação. Os *vértices* ainda irão possuir campos de "*peso*", que irá guardar a quantidade a ser produzida, e um atributo chamado "*cor*", representando o status de visita do nodo. As *arestas* irão possuir um *vértice* de destino e o peso até ele. Importante falar que o *grafo* irá utilizar uma tabela de espalhamento, usando o nome do elemento como chave e seu valor estará guardado o *vértice* com todos seus atributos.

Implementando o algoritmo em Java, utilizaremos a estrutura de dados "*HashMap*" para a classe *Grafo*, presente na biblioteca "*java.util"*. A classe *Vértice* irá possuir uma lista de arestas, então iremos utilizar a estrutura "*ArrayList*". A classe *Aresta* guardará um vértice da classe *Vértice* para referenciar a adjacência e o *peso* até chegar nele.

Algoritmo

Depois de apresentado o problema e identificado as necessidades básicas para resolvê-lo, foi desenvolvido o algoritmo para atingirmos o objetivo do software. A estrutura de dados utilizada são *grafos*, e para caminharmos nele e fazer todas as operações necessárias, iremos utilizar o método de busca em profundidade, conhecido também como *Depth-First Search (dfs)*, como base. Abaixo está uma ilustração de como funciona uma busca em profundidade de um grafo:

Grafo realizando busca em profundidade



Esse algoritmo implementa uma busca a partir de um vértice inicial, que iremos definir como o *Vértice* que representa o elemento *"Hidrogênio"*, até acharmos nosso *Vértice* objetivo, o elemento *"Ouro"*. Quando o *"Ouro"* for achado, iremos retornar ao elemento verificando todas as arestas com vértices de destinos com a cor igual a *"BRANCO"*, caso seja verdadeiro nós realizamos uma nova visita. Caso a cor verificada no *Vértice* de destino seja *"PRETO"* pegamos o peso do vértice com a cor *"PRETO"*, multiplicamos pelo peso da aresta e retornamos o valor necessário a ser produzido. Todos esses resultados serão guardados em uma *variável* que será responsável por guardar o valor e atribuir um valor *peso* ao objeto *Vértice*, antes de ser pintado de *"PRETO"*. Abaixo está o algoritmo desenvolvido para a resolução do problema:

CalculaHidrogênios( )

```
1.  Declara numeroHidrogenios
2.  Para cada Aresta a em Hidrogenio.listaDeArestas{
3.          Declara u = a.verticeDestino()
4.          Se u.cor == BRANCO{
5.              numeroHidrogenios = (visita(u) * a.peso);
6.          }
7.  }
8.  Retorna numeroHidrogenios
```

Visita(Vertice v)

```
1.  Declara resultado = 0
2.      Se v.nome != "ouro"{
3.          v.cor = AMARELO
4.          Para cada Aresta a em v.listaDeArestas{
5.              Declara u = a.verticeDestino()
6.              Se u.cor == BRANCO{
```

```
7.                                  resultado = resultado + (visita(u) * a.peso)
8.                          } Se u.cor == PRETO{
9.                                  resultado = resultado + (u.peso * a.peso)
10.                         }
11.                 }
12.                 v.peso = resultado
13.             } Senão {
14.                 v.peso = 1
15.             }
16.     v.cor = PRETO
17.     Retorna v.peso
```

Implementação em Java:

```
public BigInteger algoritmo(){
    BigInteger numHidrogenios = BigInteger.ZERO;
    for (Aresta a : grafo.get("hidrogenio").getAdjacentes()){
        if (a.getVerticeDestino().getCor() == Cor.BRANCO){
            numHidrogenios =
numHidrogenios.add((visita(a.getVerticeDestino())).multiply(a.getPeso(
)));
        }
    }
    return numHidrogenios;
}

private BigInteger visita(Vertice u){
    BigInteger resultado = BigInteger.ZERO;
    if(!u.getNomeElemento().equals("ouro")){
        u.setCor(Cor.CINZA);
        for (Aresta a: u.getAdjacentes()) {
            Vertice v = a.getVerticeDestino();
            if (v.getCor() == Cor.BRANCO) {
                resultado =
resultado.add(visita(v).multiply(a.getPeso()));
            } else if(v.getCor() == Cor.PRETO){
                resultado =
resultado.add(v.getPeso().multiply(a.getPeso()));
            }
        }
        u.setPeso(resultado);
    } else{
        u.setPeso(BigInteger.ONE);
    }
    u.setCor(Cor.PRETO);
    return u.getPeso();
}
```

O algoritmo resolve o problema apresentado, atendendo todos os requisitos propostos no desafio. Para descobrirmos a quantidade de Hidrogênios necessários para a produção de 1 elemento de *Ouro*, necessitamos passar por todos os *n vértices* das nossas fórmulas. Nesse algoritmo não iremos acessar os elementos mais do que uma vez, já que

estamos atribuindo a quantidade necessária no próprio atributo de peso do Vértice. Então chegamos na conclusão de que temos uma complexidade do nosso algoritmo em **O(n).**

Resultados

Com o algoritmo implementado na linguagem de programação Java, conseguimos realizar os testes e comprovar a sua eficiência. Cada caso foi executado 100 vezes em sequência, gravando seu resultado e tempo em um arquivo de saída. Abaixo podemos verificar o resultado de cada caso:

Tabela de resultados de cada caso

Elementos (Caso)	Resultado (Quantidade de Hidrogênios)
40	4.307.147.029
80	1.837.259.296.514
120	41.319.177.130.862
180	730.891.056.116.294.000
240	2.176.101.481.583.380.000.000.000.000
280	496.528.762.406.562.000.000.000.000.000
320	11.126.892.164.228.500.000.000.000.000
360	110.166.908.492.776.000.000.000.000.000.000.000.000

Os resultados estão de acordo com o gabarito disponibilizado previamente pelo orientador do trabalho. Com os resultados de acordo, foi medido o tempo que foi levado para executar o algoritmo. Após a execução de 100 vezes de cada teste, calculamos a média de tempo para a finalização do cálculo de hidrogênios necessários, de acordo com a fórmula passada por parâmetro:

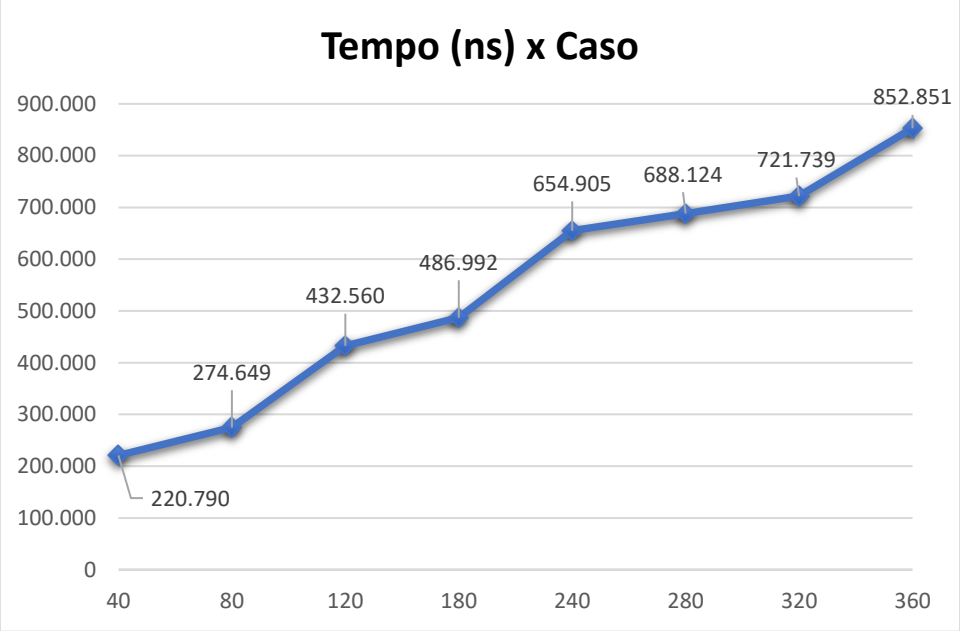
Tabela de tempo de cada caso

Elementos (Caso)	Tempo (ns)
40	220.790
80	274.649
120	432.560
180	486.992
240	654.905
280	688.124
320	721.739
360	852.851

Com os resultados levantados pelo nosso programa, tanto pelo número de hidrogênios esperado, quanto pelo tempo de execução, podemos observar o

comportamento do nosso software de maneira mais ilustrativa. A relação de cada caso e seu respectivo tempo de execução nos gerou o gráfico logo abaixo:

Gráfico relação Caso x Tempo (ns)



O gráfico nos mostra o comportamento esperado pelo algoritmo. Sabendo que é necessário passar por todos os *vértices* (elementos) para achar a quantidade exata de hidrogênios o algoritmo cumpre de maneira eficiente o solicitado nos requisitos do projeto e também é comprovado a complexidade de **O(n)**, mostrando um gráfico linear que acompanha o crescimento de elementos a serem analisados.

Conclusão

O desenvolvimento dos algoritmos conseguiu cumprir com os pré-requisitos solicitados no escopo do projeto e entregou os resultados esperados. A solução consegue entregar de maneira eficiente o objetivo final de calcular a quantidade de hidrogênios necessários para a criação de um elemento de ouro, baseado em uma fórmula criada por alquimistas.

Os resultados se demonstraram satisfatórios, e isso só foi possível pela estrutura de dados criada para a criação de um grafo com pesos e conexões entre vértices e arestas. A adaptação do algoritmo de caminhamento em profundidade conseguiu manter a eficiência e entregar o resultado esperado, de acordo com o gabarito proposto anteriormente.