

Relatório Alquimia de Elementos

Allan Groisman Kusbick

Algoritmo e Estrutura de Dados 2

Ciência da Computação — PUCRS

17 de agosto de 2023

1. Introdução

Este trabalho tem como objetivo a elaboração de um algoritmo que calcula quantas unidades do elemento químico hidrogênio são necessários para produzir uma unidade de ouro. Existem diversas “receitas” de uma alquimia que trazem diversos outros elementos, estes que se transformam, a partir do hidrogênio, para enfim chegar ao ouro. Estas transformações entre elementos tem um custo: podem ser necessários 10 hidrogênios para criar um elemento cesio, e na sequência 10 cesios para transformar-se em um ouro, sendo necessários assim, 50 hidrogênios no total. Para melhor entendimento do problema foi disponibilizado uma receita exemplo que segue na figura seguinte:

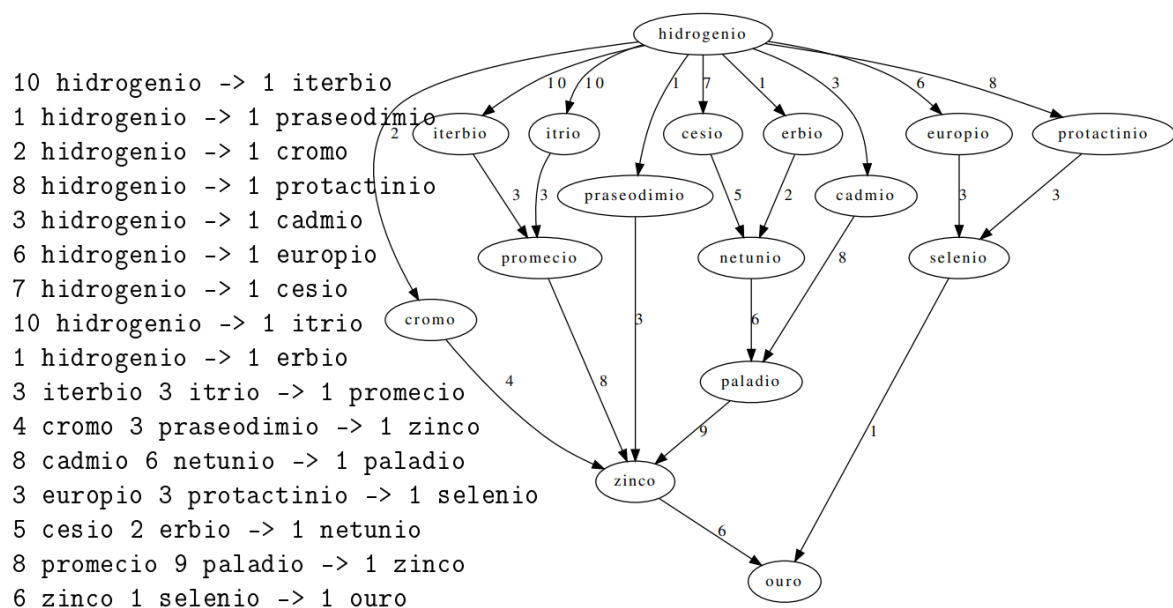
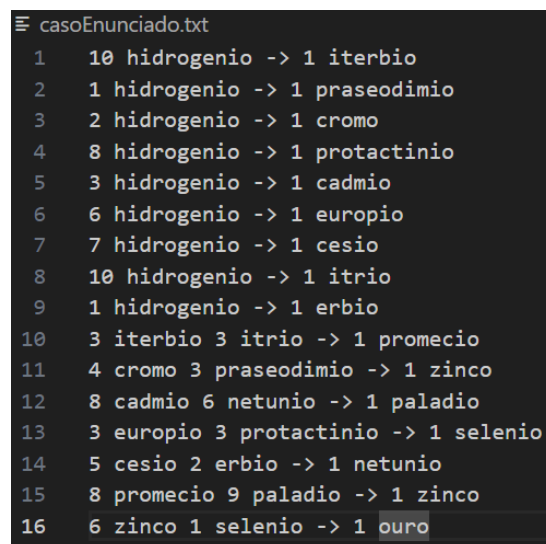


Figura 1: Exemplo de receita padrão fornecido no enunciado do problema.

É possível observar a entrada de texto de todos os elementos e quais são as transformações necessárias, além de uma representação gráfica na forma de um dígrafo valorado. No dígrafo pode-se observar o hidrogênio como origem e o ouro como destino final, além de diversos caminhos que passam por vértices correspondentes aos elementos intermediários. Nas arestas os pesos são as quantidades necessárias do elemento imediatamente anterior, a fim de efetuar uma transformação. A partir desse dígrafo então, o objetivo é implementar um algoritmo que percorra todas as transformações e calcule o valor total de hidrogênios necessários para chegar em um ouro.

2. Dígrafo Valorado e Importação da “Receita”

Primeiramente é criado um dígrafo que importa uma receita através de sua função construtora. As receitas foram disponibilizadas em arquivos texto como é possível observar na figura 2:

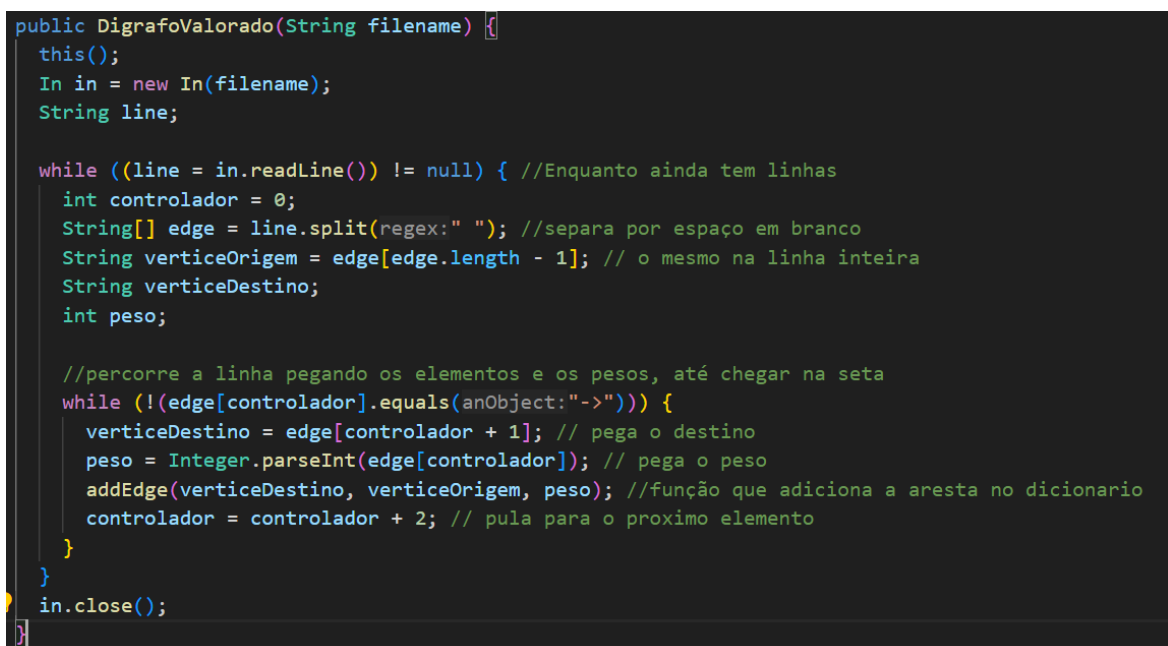


```
casoEnunciado.txt
1 10 hidrogenio -> 1 iterbio
2 1 hidrogenio -> 1 praseodimio
3 2 hidrogenio -> 1 cromo
4 8 hidrogenio -> 1 protactinio
5 3 hidrogenio -> 1 cadmio
6 6 hidrogenio -> 1 europio
7 7 hidrogenio -> 1 cesio
8 10 hidrogenio -> 1 itrio
9 1 hidrogenio -> 1 erbio
10 3 iterbio 3 itrio -> 1 promecio
11 4 cromo 3 praseodimio -> 1 zinco
12 8 cadmio 6 netunio -> 1 paladio
13 3 europio 3 protactinio -> 1 selenio
14 5 cesio 2 erbio -> 1 netunio
15 8 promecio 9 paladio -> 1 zinco
16 6 zinco 1 selenio -> 1 ouro
```

Figura 2: Captura de tela de arquivo texto de uma “receita”.

Em cada linha há informação de uma transformação. Pega-se a linha 10 como exemplo: “3 iterbio 3 itrio -> 1 promecio”. Antes da seta “->” aparecem as quantidades e os elementos necessários para a realizar a transformação que resulta no elemento pós seta. Neste caso são necessários 3 iterbios e 3 itrios para obter 1 promecio.

A função construtora do dígrafo (figura 3) faz o papel de ler esse arquivo texto linha por linha e adicionar estes elementos e transformações como vértices e arestas. Estas informações são armazenadas em um dicionário no formato *Map<String, List<Edge>>* no qual a *String* é o vértice e o *List<Edge>* é a lista de arestas que partem deste vértice. Por fim, a aresta representada pela classe *Edge*, guarda sua origem, o destino e o peso relacionado.



```
public DigrafoValorado(String filename) {
    this();
    In in = new In(filename);
    String line;

    while ((line = in.readLine()) != null) { //Enquanto ainda tem linhas
        int controlador = 0;
        String[] edge = line.split(regex:" "); //separa por espaço em branco
        String verticeOrigem = edge[edge.length - 1]; // o mesmo na linha inteira
        String verticeDestino;
        int peso;

        //percorre a linha pegando os elementos e os pesos, até chegar na seta
        while (!(edge[controlador].equals(anObject:"->"))) {
            verticeDestino = edge[controlador + 1]; // pega o destino
            peso = Integer.parseInt(edge[controlador]); // pega o peso
            addEdge(verticeDestino, verticeOrigem, peso); //função que adiciona a aresta no dicionario
            controlador = controlador + 2; // pula para o próximo elemento
        }
    }
    in.close();
}
```

Figura 3: Captura de tela da função construtora do dígrafo valorado.

3. Algoritmo

Para construir o algoritmo e calcular a quantidade de hidrogênios necessários afim de obter um ouro, foram desenvolvidas duas funções: *calcularHidrogenio(String verticeInicial, String verticeFinal)* e *hidrogenioRecurso(String verticeAtual, Map<String, BigInteger>)*. A solução teve como ideia base o algoritmo de caminhamento DFS (*depth first search*), este o qual utiliza da recursão para descer em profundidade em um grafo.

A função *calcularHidrogenio()* serve como auxiliar para criação de um dicionário *Map <String, BigInteger> memoriaDeValores* que é utilizado para armazenar os valores dos vértices já calculados, afim de evitar repetições desnecessárias. O vértice final (ouro) é adicionado ao dicionário com valor um, pois dessa forma a recursividade o utiliza como ponto de resolução, o que evita a tentativa de acessar suas arestas inexistentes, além de servir como base unitária no cálculo geral.

Por fim, o método retorna o resultado da função *hidrogenioRecurso()*, esta que é chamada com os parâmetros: dicionário *memoriasDeValores* e *String verticeInicial*, pois é a partir deste elemento inicial (hidrogênio) que o cálculo deve ser feito.

```
public BigInteger calcularHidrogenio(String verticeInicial, String verticeFinal) {
    Map<String, BigInteger> memoriaDeValores = new HashMap<>();
    memoriaDeValores.put(verticeFinal, BigInteger.ONE);
    return hidrogenioRecurso(verticeInicial, memoriaDeValores);
}
```

Figura 5: captura de tela da função *calcularHidrogênio()*.

O cálculo, quantidade de *verticeInicial* para uma unidade do *vérticeFinal*, ou hidrogênios para um ouro, realmente acontece na função *hidrogenioRecurso(String verticeAtual, Map<String, BigInteger>)* que sempre recebe como parâmetros o mesmo dicionário de valores *memoriaDeValores*, além do vértice sobre qual está atuando no momento, chamado de *verticeAtual*. Segue na figura 6, uma visão geral desta função.

```
private BigInteger hidrogenioRecurso(String verticeAtual, Map<String, BigInteger> memoriaDeValores) {
    // procura na memoria se o valor do vertice ja foi calculado. Se ja existir so retorna ele.
    if (memoriaDeValores.containsKey(verticeAtual)) {return memoriaDeValores.get(verticeAtual);}
    // inicializa o resultado com zero, acessa os filhos e vai somando o valor deles.
    BigInteger resultado = BigInteger.valueOf(0);
    List<Edge> listaFilhos = graph.get(verticeAtual);
    for (Edge edge : listaFilhos) {
        // pega o valor da aresta do filho atual
        BigInteger valorAresta = BigInteger.valueOf(edge.getWeight());
        // e pega o valor dele recursivamente (caso ja estiver na memoria, no inicio da
        // recursão, já vai retornar)
        BigInteger valorFilho = hidrogenioRecurso(edge.getW(), memoriaDeValores);
        // soma o valor de todos os filhos recursivamente
        resultado = resultado.add(valorAresta.multiply(valorFilho));
    }
    // coloca na memoria o valor do vertice atual para uso futuro
    memoriaDeValores.put(verticeAtual, resultado);
    // retorna o valor
    return resultado;
}
```

Figura 6: Captura de tela da função *hidrogeniorecurso()*.

O primeiro passo é conferir se o valor do vértice já foi calculado no dicionário de valores que, como dito anteriormente, evita calcular repetidamente o valor de um vértice. Caso o vértice esteja no dicionário, retorna o seu valor (Figura 7).

```
// procura na memoria se o valor do vertice ja foi calculado. Se ja existir so retorna ele.  
if (memoriaDeValores.containsKey(verticeAtual)) {return memoriaDeValores.get(verticeAtual);}
```

Figura 7: Segmento da função *hidrogenioRecursivo()*.

Caso o valor do vértice ainda não tenha sido calculado, inicia-se seu cálculo. Este valor é resultado da soma de todas as arestas que partem do vértice, estas que têm seus pesos multiplicados pelo valor dos seus respectivos vértices destino. Sendo n o número de arestas, *valorAresta* o peso da aresta e *valorFilho* o valor do vértice destino, na figura 8 segue uma fórmula que resume este cálculo.

$$= \sum_{i=1}^n \text{valorAresta}_i \times \text{valorFilho}_i$$

Figura 8: Fórmula do cálculo do valor de um vértice.

Para implementação desta fórmula, obtém-se, a partir do grafo, a lista de arestas que partem do vértice atual (*List<Edge> listaFilhos*) e para cada destas arestas, captura-se seu respectivo peso (*valorAresta*) e valor do vértice destino (*valorFilho*). Dentro desta iteração soma-se, em uma variável *resultado*, a multiplicação do *valorAresta* pelo *valorFilho*, *resultado* este que, ao final da iteração, é inserido no dicionário de valores já calculados e então retornado pela função. Segue na figura 9 este trecho de código.

```
// inicializa o resultado com zero, acessa os filhos e vai somando o valor deles.  
BigInteger resultado = BigInteger.valueOf(val:0);  
List<Edge> listaFilhos = graph.get(verticeAtual);  
for (Edge edge : listaFilhos) {  
    // pega o valor da aresta do filho atual  
    BigInteger valorAresta = BigInteger.valueOf(edge.getWeight());  
    // e pega o valor dele recursivamente (caso ja estiver na memoria, no inicio da  
    // recursão, já vai retornar)  
    BigInteger valorFilho = hidrogenioRecursivo(edge.getW(), memoriaDeValores);  
    // soma o valor de todos os filhos recursivamente  
    resultado = resultado.add(valorAresta.multiply(valorFilho));  
}  
// coloca na memoria o valor do vertice atual para uso futuro  
memoriaDeValores.put(verticeAtual, resultado);  
// retorna o valor  
return resultado;
```

Figura 9: Recorte da função *hidrogenioRecursivo()* onde acontece o cálculo do valor do vértice.

A recursão acontece ao utilizar a própria função *hidrogenioRecursivo()* para obter o valor do vértice destino, chamado de *valorFilho*. A partir desta recursividade se tem a característica de profundidade, herdada do DFS, que acontece pelo fato do caminharmento descer tão profundamente quanto possível no ramo antes de retroceder.

A profundidade máxima de cada recursão é definida a partir da verificação do valor requerido no dicionário de valores já calculados *memoriaDeValores*, pois se encontrado, deixa de ser necessário visitar os filhos do nodo destino e descer ainda mais no ramo para calcular seu valor. Por esse motivo

que no início do algoritmo, ainda na função *calcularHidrogenio()*, é inserido o *verticeFinal* desejado (ouro) na *memoriaDeValores*, pois como dito anteriormente, ele serve de limite à recursividade.

4. Testes e resultados

Para testar o algoritmo foram disponibilizados doze casos de testes, com diferentes tamanhos, a fim de testar sua eficiência, além do *casoEnunciado.txt* que é o mesmo apresentado como exemplo base do problema (figura 1 e figura 2). Na classe *AppAlquimia.java*, esta que serve como base para a aplicação do algoritmo, em sua *main* há um vetor de *Strings* com os nomes dos testes, e para cada teste, é chamada a função auxiliar *algoritmoHidrogenio(String teste)*.



```
public class AppAlquimia {
    Run | Debug
    public static void main(String[] args) {

        // Lista de arquivos na ordem desejada
        String[] arquivos = {
            "casoa5.txt",
            "casoa20.txt",
            "casoa40.txt",
            "casoa60.txt",
            "casoa80.txt",
            "casoa120.txt",
            "casoa180.txt",
            "casoa240.txt",
            "casoa280.txt",
            "casoa320.txt",
            "casoa360.txt",
            "casoa400.txt",
            "casoEnunciado.txt"
        };

        // Percorre a lista de arquivos
        for (String teste : arquivos) {
            // Imprime e manda rodar o algoritmo de hidrogênio
            System.out.println("Teste: " + teste);
            algoritmoHidrogenio(teste);
        }
    }
}
```

Figura 10: Captura do método *main()* da classe *AppAlquimia.java*.

No método *algoritmoHidrogenio(String teste)* há a medição de tempo de execução da criação do dígrafo com o carregamento do arquivo texto (*new DigrafoValorado("caminho/arquivoTexto")*), além da medição da aplicação do algoritmo de hidrogênios correspondentes (*.calcularHidrogenio(verticeInicial,verticeFinal)*). Dessa forma é possível analisar e comparar os diferentes resultados. Segue este método na figura 11.

```

private static void algoritmoHidrogenio(String teste) {

    long startTime;
    long endTime;
    long tempoFinal;

    // Calcula o tempo de carregamento do grafo
    startTime = System.currentTimeMillis(); // Captura o tempo inicial

    // cria um novo grafo para cada arquivo
    DigrafoValorado grafoAlquimia = new DigrafoValorado("casosTeste\\" + teste);

    endTime = System.currentTimeMillis(); // Captura o tempo final
    tempoFinal = endTime - startTime; // Calcula o tempo decorrido
    System.out.println("Tempo de carregamento: " + tempoFinal + " milissegundos");

    // Executa o algoritmo sobre o arquivo
    String verticeInicial = "hidrogenio";
    String verticeFinal = "ouro";

    startTime = System.currentTimeMillis(); // Captura o tempo inicial

    // Calcula e printa na tela o numero de hidrogenios
    System.out.println("Valor até o ouro: " + grafoAlquimia.calcularHidrogenio(verticeInicial, verticeFinal));

    endTime = System.currentTimeMillis(); // Captura o tempo final
    tempoFinal = endTime - startTime; // Calcula o tempo decorrido
    System.out.println("Tempo de execução: " + tempoFinal + " milissegundos");
    System.out.println();
    // grafoAlquimia.imprimirGrafo();
}

```

Figura 11: Captura de tela do método *algoritmoHidrogenio()*.

Após a execução de todos os casos testes obteve-se a seguinte saída (tabela 1) que contém tempo de carregamento dos arquivos, tempo de execução do algoritmo de hidrogênios e o resultado de quantos hidrogênios são necessários. Os tempos de carregamento e de execução em cada caso foram insignificantes, tendo máximo de 33 milissegundos de carregamento no “casoa5.txt” e de 6 milissegundos na execução do “casoa320.txt”.

	Carregamento	Execução	Resultado
casoa5.txt	33ms	1ms	102744
casoa20.txt	8ms	1ms	800770
casoa40.txt	7ms	3ms	8256835
casoa60.txt	4ms	1ms	1641700
casoa80.txt	3ms	3ms	69922658719
casoa120.txt	4ms	4ms	721329018682809
casoa180.txt	6ms	0ms	8428729212204778
casoa240.txt	0ms	0ms	437216915509229458771143884
casoa280.txt	0ms	0ms	12582782794727272819929298620801
casoa320.txt	10ms	6ms	9580713165023312774588914805494
casoa360.txt	4ms	5ms	25082239764430426527755447803789025
casoa400.txt	10ms	2ms	63033055628032755906131628407108501716
casoEnunciado.txt	0ms	0ms	16272

Tabela 1: Tempos e resultados dos testes na execução do programa.

Porém, observou-se um tempo excessivo no primeiro teste executado de 33 milissegundos de carregamento, este que estava fora da média dos testes subsequentes. Por esse motivo foram executados, de maneira independente, dez vezes cada teste, a fim de obter uma média isolada, sem nenhuma interferência da execução dos demais. A seguir na tabela 2 é possível observar os resultados já com suas calculadas.

Valores a partir da média de 10 execuções	Carregamento	Execução	Resultado (número de hidrogênios)
casoa5.txt	66ms	5,8ms	102744
casoa20.txt	72,8ms	3,6ms	800770
casoa40.txt	53,1ms	4,2ms	8256835
casoa60.txt	53,7ms	1ms	1641700
casoa80.txt	62,5ms	10,5ms	69922658719
casoa120.txt	56,5ms	13,7ms	721329018682809
casoa180.txt	70ms	8,3ms	8428729212204778
casoa240.txt	58ms	10,8ms	437216915509229458771143884
casoa280.txt	77,3ms	4,5ms	12582782794727272819929298620801
casoa320.txt	70,5ms	12,7ms	9580713165023312774588914805494
casoa360.txt	85,3ms	11,7ms	25082239764430426527755447803789025
casoa400.txt	62,3ms	7,5ms	63033055628032755906131628407108501716
casoEnunciado.txt	42,2ms	2ms	16272

Tabela 2: Tempos médios e resultados dos testes executados individualmente no programa.

A complexidade do algoritmo DFS utilizado como base do algoritmo de hidrogênios é $O(V + E)$, onde E são arestas e V os vértices, ou seja, é a soma dos dois, pois é necessário passar por todos os caminhos possíveis e visitar todos os vértices. A diferença para o algoritmo desenvolvido neste trabalho é que o programa não apenas marca os vértices como visitados, mas associa um valor respectivo em um dicionário.

O dicionário é quem garante a eficiência do algoritmo, ele evita que a recursão aconteça toda vez até o último nível onde está o ouro, pois se o vértice já foi visitado, já teve seu valor calculado, e por isso não há a necessidade de continuar com a recursão. Desta forma, pode-se dizer que a complexidade continua $O(V + E)$ onde é necessário passar por todos vértices e arestas, porém a visitação tem apenas o custo extra de calcular o valor do vértice.