Relatório T2 Algoritmos e Estrutura de Dados II

Augusto Baldino, Vinícius Petersen,
Larissa Laier
Escola Politécnica — PUCRS

22 de novembro de 2023

Resumo

Este relatório aborda a aplicação de estruturas de grafos na resolução do T2 da disciplina Algoritmos e Estrutura de Dados II, visando a prática da implementação de Grafos. O problema consiste em determinar a quantidade de hidrogênio necessária para criar uma unidade de ouro, seguindo regras de transformação entre elementos químicos. As soluções propostas envolvem a representação do problema como um grafo e a aplicação de algoritmos para calcular as quantidades necessárias.

Introdução

É chegada a época da Grande Convenção dos Alquimistas, um evento que ocorre a cada 100 anos e reúne os mais renomados alquimistas para celebrar com festividades, palestras educativas, brindes e a troca de receitas mágicas. O ápice desse encontro é o jantar de gala, onde se entrega o cobiçado prêmio da Asinha de Morcego Dourada para a melhor ideia de receita capaz de transmutar elementos químicos em ouro.

Nesse contexto de fervor alquímico, as receitas, aparentemente simples, revelam-se cada vez mais complexas com o passar dos séculos. O Conselho dos Alquimistas, em busca da receita perfeita, contratou você para automatizar a tarefa de analisar essas fórmulas e responder a uma pergunta crucial: quanto hidrogênio é necessário para realizar cada receita? Essa informação torna-se vital, uma vez que a quantidade de hidrogênio impacta diretamente nos custos, e, afinal, o hidrogênio não é um recurso ilimitado.

A figura abaixo ilustra uma dessas receitas, em que os nodos representam elementos e as quantidades necessárias para a transmutação estão anotadas nas arestas.

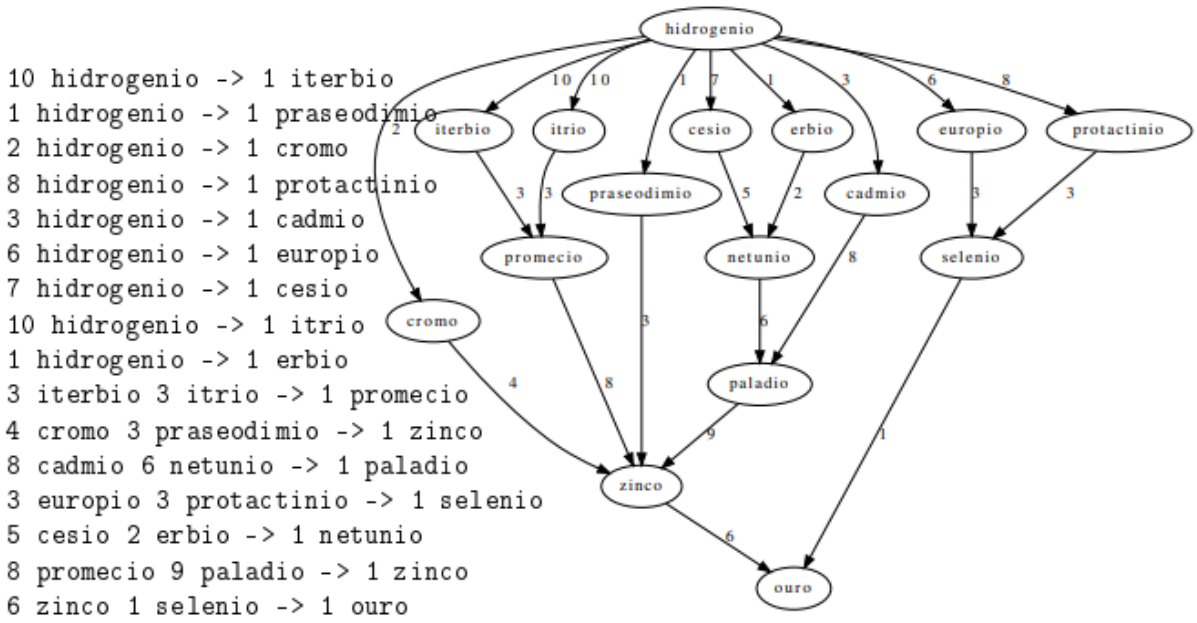


Foto retirada do enunciado do Trabalho 2

O desafio proposto consiste em desenvolver um algoritmo capaz de calcular e apresentar a quantidade de hidrogênio utilizada para produzir uma unidade de ouro. Este relatório descreverá o problema em detalhes, a solução adotada, os resultados obtidos nos casos de teste, os tempos de execução, as dificuldades enfrentadas e, por fim, as conclusões alcançadas ao longo dessa jornada.

Primeira solução:

Para a nossa primeira solução, optamos por utilizar a linguagem Java, pelo maior domínio do grupo nesta linguagem. Começamos a desenvolver a nossa solução com duas classes: classe Graph e elementCompositionReader.

A classe **Graph** representa um grafo direcionado, onde os elementos químicos são os nodos e as arestas indicam a relação entre esses elementos nas receitas alquímicas. Essa classe é utilizada para calcular a quantidade total de hidrogênio necessária para produzir uma unidade de ouro, seguindo as receitas dadas.

A classe **ElementCompositionReader** é responsável por ler as composições químicas a partir de um arquivo de texto, construir um grafo usando a classe **Graph**, e calcular a quantidade total de hidrogênio necessária para criar uma unidade de um elemento específico.

```java
import java.util.*;
public class Graph {
    private Map<String, Integer> elementToHydrogen = new HashMap<>();
    private Map<String, List<String>> elementToNeighbors = new HashMap<>();

    public void addEdge(String element, int hydrogenCount) {
        elementToHydrogen.put(element, hydrogenCount);
        elementToNeighbors.put(element, new ArrayList<>());
    }

    public int findHydrogenRequired(String element) {
        return dfs(element, new HashSet<>());
    }

    private int dfs(String element, Set<String> visited) {
        if (!elementToHydrogen.containsKey(element)) {
            return 0;
        }

        visited.add(element);
        int hydrogenCount = elementToHydrogen.get(element);

        for (String neighbor : elementToNeighbors.get(element)) {
            if (!visited.contains(neighbor)) {
                hydrogenCount += dfs(neighbor, visited);
            }
        }

        visited.remove(element);
        return hydrogenCount;
    }
}
```

Representação do código da solução 1 classe Graph.

Estrutura de Dados Utilizadas:

- **Map<String, Integer> elementToHydrogen:** Este map associa cada elemento químico a uma quantidade específica de hidrogênio. A chave é o nome do elemento e o valor é o número de átomos de hidrogênio associados a esse elemento.
- **Map<String, List<String>> elementToNeighbors:** Este map mantém as relações entre elementos químicos. A chave é o nome do elemento e o valor é uma lista de elementos vizinhos, representando as transformações químicas possíveis.

```java
import java.io.*;

public class ElementCompositionReader {
    public static void main(String[] args) throws IOException {
        Graph graph = new Graph();
        BufferedReader br = new BufferedReader(new FileReader("Caso1.txt"));
        String line;
        while ((line = br.readLine()) != null) {
            String[] parts = line.split(" -> ");
            String[] lhs = parts[0].split(" ");
            String[] rhs = parts[1].split(" ");
            int hydrogenCount = Integer.parseInt(lhs[0]);
            String element = rhs[1];
            graph.addEdge(element, hydrogenCount);
        }
        br.close();

        String targetElement = "ouro";

        int hydrogenCount = graph.findHydrogenRequired(targetElement);

        System.out.println("Hidrogênio necessário para criar " + targetElement + ": " + hydrogenCount);
    }
}
```

Representação do código da solução 1 da classe ElementCompositionReader.

Caso1.txt

```
8 hidrogenio -> 1 prata
1 prata -> 1 ouro
```

Arquivo de texto simplificado que utilizamos para testes rápidos.

O problema deste código é que ao compilá-lo encontramos um valor inesperado:

```
PS C:\Users\augusto.baldino\Downloads\T2_Alest> & 'C:\Program Files\Java\jre-1.8\bin\java.exe' '-cp' 'C:\Users\augusto.baldino\AppData\Roaming\Code\User\workspaceStorage\20b91a1016360a0b0e770c06a9536df1\redhat.java\jdt_ws\T2_Alest_381fca90\bin' 'ElementCompositionReader'
Hidrogênio necessário para criar ouro: 1
PS C:\Users\augusto.baldino\Downloads\T2_Alest>
```

Ele está retornando o valor 1, que está equivocado. O caso correto corresponderia a 8 hidrogênios.

O problema estava na lógica implementada na classe **Graph, especificamente no addEdge** que adiciona uma aresta ao grafo, mas a representação do grafo. Como o objetivo é construir um grafo de composições químicas, é necessário conectar os elementos com suas respectivas composições, atribuindo os valores em hidrogênios aos respectivos elementos.

Segunda solução:

Para a segunda solução, resolvemos mudar a lógica da classe Graph:

Welcome | Graph.java

Graph.java > Graph > dfs(String, Set<String>)

```java
import java.util.*;
public class Graph {
    private Map<String, Integer> elementToHydrogen = new HashMap<>();
    private Map<String, List<String>> elementToNeighbors = new HashMap<>();

    public void addEdge(String element, int hydrogenCount) {
        elementToHydrogen.put(element, hydrogenCount);
        elementToNeighbors.put(element, new ArrayList<>());
    }

    public int findHydrogenRequired(String element) {
        return dfs(element, new HashSet<>());
    }

    private int dfs(String element, Set<String> visited) {
        if (!elementToHydrogen.containsKey(element)) {
            return 0;
        }

        visited.add(element);
        int hydrogenCount = elementToHydrogen.get(element);

        for (String neighbor : elementToNeighbors.get(element)) {
            if (!visited.contains(neighbor)) {
                hydrogenCount += dfs(neighbor, visited);
            }
        }

        visited.remove(element);
        return hydrogenCount;
    }
}
```

Métodos Principais:

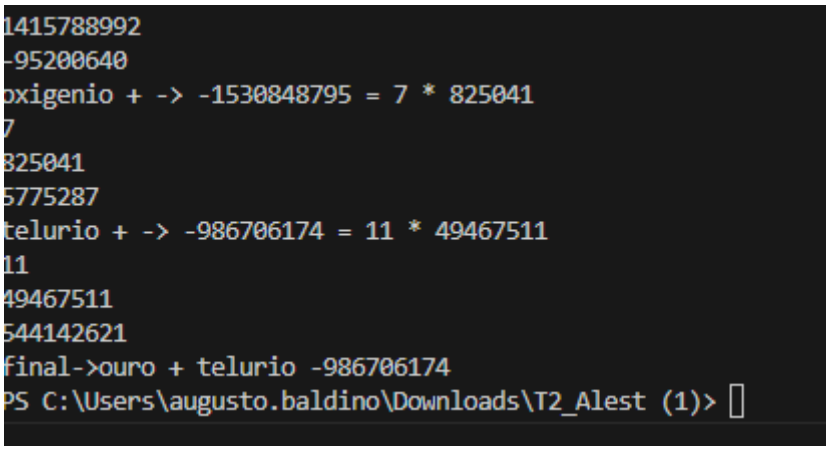
- **addEdge(String elementL, String elementR):** Este método adiciona uma aresta ao grafo. Se o elementoL contiver a palavra "hidrogenio", ele adiciona uma relação de composição de hidrogênio entre o elementoR e a quantidade de hidrogênio especificada. Caso contrário, ele adiciona uma aresta entre os elementos, representando uma transformação química.
- **calculateHydrogen():** Este método percorre o grafo e calcula a quantidade total de hidrogênio necessária para produzir cada elemento, considerando as relações de composição de hidrogênio e as transformações químicas definidas no grafo. Caso ele encontre um elemento não especificado por uma quantidade de hidrogênios, ele chamará o método **calculateHydrogenForElement(String element)** para definir a quantidade de hidrogênios dos elementos.
- **calculateHydrogenForElement(String element):** Este método calcula a quantidade de hidrogênio necessária para produzir um elemento específico. Ele é chamado recursivamente para calcular as quantidades de hidrogênio para os elementos vizinhos, assim como é o caso de ele encontrar um elemento sem hidrogênios.

```java
import java.io.*;

public class ElementCompositionReader {
    Run | Debug
    public static void main(String[] args) throws IOException {
        Graph graph = new Graph();
        String elementL = "";
        BufferedReader br = new BufferedReader(new FileReader(".\\casos_t2_2023_2\\casoj360.txt"));
        String line;
        while ((line = br.readLine()) != null) {
            String[] parts = line.split(" -> ");
            String[] lhs = parts[0].split(" ");
            String[] rhs = parts[1].split(" ");
            for(int i=0; i<lhs.length ; i=i+2){
            elementL = lhs[i]+" "+lhs[i+1];}
            String elementR = rhs[1];
            System.out.println("ReaderR:"+elementR);
            for(int i=0; i<lhs.length ; i=i+2){
            elementL = lhs[i]+" "+lhs[i+1];
            graph.addEdge(elementL, elementR);
            System.out.println("ReaderL:"+elementL);
            }
            elementL = "";
        }
        br.close();

        graph.calculateHydrogen();



    }
}
```

ElementCompositionReader da segunda solução

Nesta classe, fizemos algumas modificações visam aprimorar a leitura e a adição de relações químicas do arquivo de entrada ao grafo. O loop **for** foi introduzido para percorrer todos os elementos na parte esquerda de cada seta (**->**). Dessa forma, cada elemento é adicionado individualmente ao grafo, permitindo uma representação mais precisa das relações químicas. Além disso, foram incluídas impressões para facilitar o acompanhamento do processo. O cálculo das quantidades de hidrogênio é realizado após a leitura do arquivo, proporcionando uma análise eficiente das relações químicas estabelecidas. Essas adaptações buscam otimizar a manipulação dos dados químicos e melhorar a compreensão do funcionamento do programa.

Contudo, o código não estava perfeito, pois existiam casos em que dava um problema chamado overflow. O problema de overflow ocorre quando o resultado de uma operação aritmética ultrapassa o valor máximo que pode ser representado por um determinado tipo de dado, no caso, o tipo **int** em Java. O tipo **int** é de 32 bits, o que significa que pode representar valores no intervalo de aproximadamente -2 bilhões a +2 bilhões.
Se durante as operações matemáticas o resultado exceder esse intervalo, ocorre overflow, e o valor excedente é "enrolado" de volta para o início do intervalo, causando resultados incorretos e imprevisíveis.

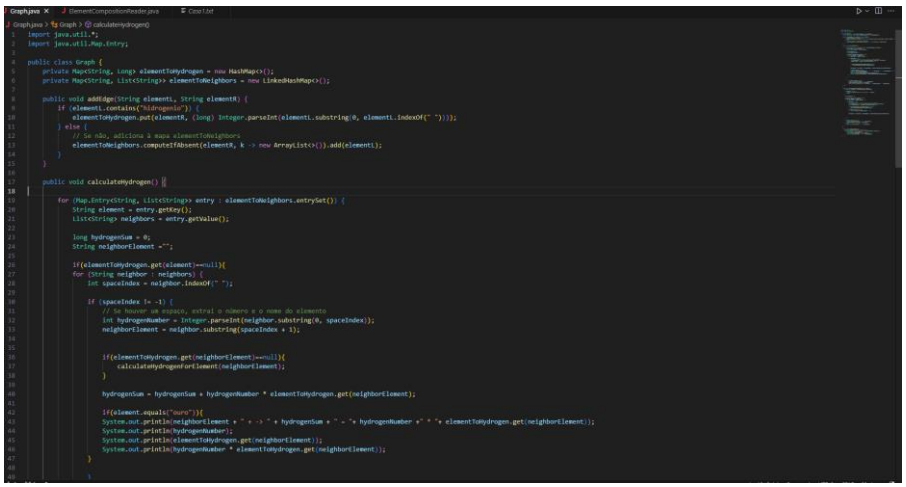


Este é o resultado ao rodar o Casoj80.txt

Terceira Solução:

Para a terceira solução, resolvemos utilizar o tipo de dado Long, que tem uma capacidade maior de armazenamento, utilizando 64 bits em comparação com os 32 bits do tipo **int**.

A faixa de representação de um **long** vai de aproximadamente -9 quintilhões a +9 quintilhões. Em contraste, o tipo **int** cobre uma faixa de aproximadamente -2 bilhões a +2 bilhões.

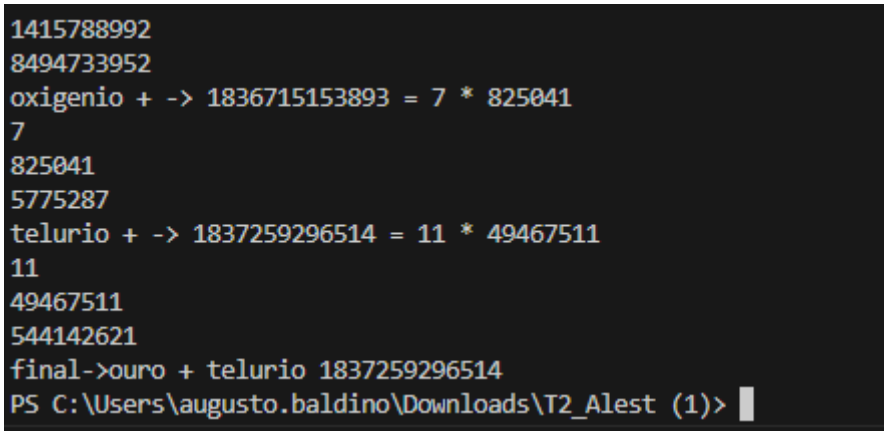


Parte 1 do código da classe Graph utilizando long

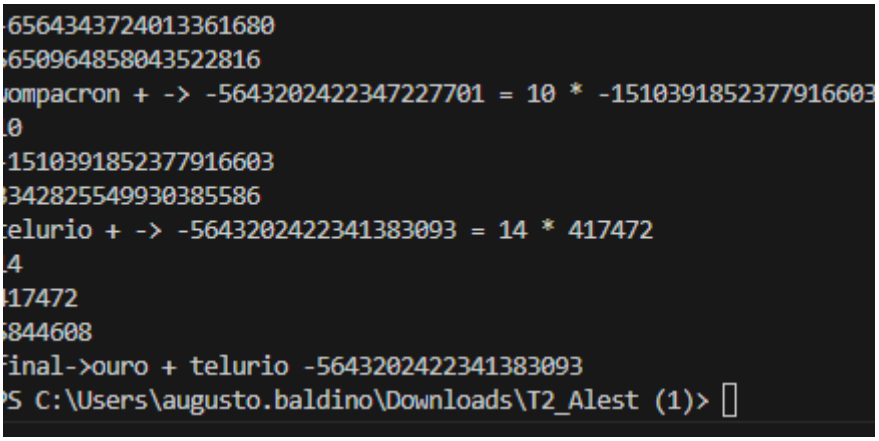
```java
public void calculateHydrogenForElement(String element) {
    List<String> neighbors = elementToNeighbors.get(element);

    if (neighbors == null) {
        System.out.println("Elemento não encontrado: " + element);
        return;
    }

    long hydrogenSum = 0;

    if (elementToHydrogen.get(element) == null) {
        for (String neighbor : neighbors) {
            int spaceIndex = neighbor.indexOf(" ");

            if (spaceIndex != -1) {
                // Se houver um espaço, extrai o número e o nome do elemento
                int hydrogenNumber = Integer.parseInt(neighbor.substring(0, spaceIndex));
                String neighborElement = neighbor.substring(spaceIndex + 1);

                if (elementToHydrogen.get(neighborElement) == null) {
                    calculateHydrogenForElement(neighborElement);
                }
                if (element.equals("ouro")) {
                    System.out.println(neighborElement + " +");
                }

                hydrogenSum = hydrogenSum + hydrogenNumber * elementToHydrogen.get(neighborElement);

            }
        }

        if (!element.equals("ouro")) {
            System.out.println(element + " -> " + hydrogenSum);
            elementToHydrogen.put(element, hydrogenSum);
        } else {
            System.out.println(element + " " + hydrogenSum);
        }
    }
}
```

Segunda parte das modificações feitas na classe Graph

Utilizando esta solução, o código casoj80.txt já estava funcionando corretamente.



Por sua vez, o código ainda dava Overflow, desta vez em casos ainda mais rigorosos, mais especificamente a partir do casoj240.



como pode ser visto no casot240.txt

Quarta solução:

Desta vez, tivemos que recorrer a documentação do java para encontrarmos um tipo de dado que comportasse o número de hidrogênios necessários a partir do “casoj240” e para isso utilizamos o BigInteger.

BigInteger é uma classe em Java que está incluída no pacote **java.math**. Ela é projetada para lidar com números inteiros arbitrariamente grandes, não limitados pelo tamanho dos tipos de dados primitivos, como **int** ou **long**. Por isso a principal razão para utilizar **BigInteger** em vez de **long** está relacionada com a capacidade de representação. Enquanto **long** tem um limite superior em torno de 9 quintilhões, **BigInteger** não possui um limite teórico superior. Isso torna a classe adequada para lidar com cálculos envolvendo números muito grandes, onde **long** pode sofrer overflow.

No contexto do código, que lida com a contagem de hidrogênio em processos de alquimia, o uso de **BigInteger** é apropriado, pois você está acumulando valores que podem crescer significativamente e ultrapassar os limites dos tipos primitivos.

```java
    for (Map.Entry<String, List<String>> entry : elementToNeighbors.entrySet()) {
        String element = entry.getKey();
public void calculateHydrogenForElement(String element) {
    List<String> neighbors = elementToNeighbors.get(element);

    if (neighbors == null) {
        System.out.println("Elemento não encontrado: " + element);
        return;
    }

    BigInteger hydrogenSum = BigInteger.ZERO;

    if (elementToHydrogen.get(element) == null) {
        for (String neighbor : neighbors) {
            int spaceIndex = neighbor.indexOf(" ");

            if (spaceIndex != -1) {
                // Se houver um espaço, extrai o número e o nome do elemento
                int hydrogenNumber = Integer.parseInt(neighbor.substring(0, spaceIndex));
                String neighborElement = neighbor.substring(spaceIndex + 1);

                if (elementToHydrogen.get(neighborElement) == null) {
                    calculateHydrogenForElement(neighborElement);
                }
                if (element.equals("ouro")) {
                    System.out.println(neighborElement + " +");
                }

                hydrogenSum = hydrogenSum.add(BigInteger.valueOf(hydrogenNumber).multiply(elementToHydrogen.get(neighborElement)));
            }
        }

        if (!element.equals("ouro")) {
            System.out.println(element + " -> " + hydrogenSum);
            elementToHydrogen.put(element, hydrogenSum);
        } else {
            System.out.println(element + " " + hydrogenSum);
        }
    }
}
```

Metodo calculateHydrogenForElement Resultado

Ao solucionarmos o problema proposto, utilizamos o método currentTimeMills para observarmos o tempo de execução da nossa solução. Para chegarmos a um valor mais aproximado, rodamos 3 vezes cada um dos casos e encontramos os seguintes resultados:

casoj40.txt: (50ms + 50ms + 53ms)/3 = 51ms

casoj80.txt: (57 + 57 + 59)/3 = 57ms

casoj120.txt: (69 + 63 + 66)/3 = 66ms

casoj180.txt: (77 + 65 + 68)/3 = 70ms

casoj240.txt: (67 + 74 + 75)/3 = 72ms

casoj280.txt: (71 + 71 + 68)/3 = 70ms

casoj320.txt: (77 + 74 + 71)/3 = 74ms

casoj360.txt: (82 + 78 + 80)/3 = 80ms

Conclusões:

Concluímos que a representação de reações químicas por meio de grafos acíclicos, aliada ao uso de **BigInteger**, proporciona uma solução robusta e flexível para o desafio da alquimia moderna. A estrutura de dados e as escolhas técnicas realizadas foram fundamentais para alcançar resultados precisos e eficientes na determinação da quantidade de hidrogênio para a produção de ouro.

A escolha de um grafo dirigido possibilitou a representação natural das relações de dependência entre os elementos, refletindo fielmente as regras específicas das reações. A estrutura de vizinhança entre nodos facilitou a navegação, garantindo uma execução eficiente dos cálculos.

Referências:

[1] Araújo Rogério: “os-tipos-primitivos-da-linguagem-java”. 2022
https://blog.grancursosonline.com.br/os-tipos-primitivos-da-linguagem-java/

[2] Oracle;“ClassBiginteger”
https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html