Trabalho 2: Os alquimistas se reúnem

Marcelo A. F. Martins[*], Arthur R. S. Ferreira[†]
Pontifícia Universidade Católica do Rio Grande do Sul

20 de novembro de 2023

Resumo

Este artigo descreve uma alternativa de solução para a segunda avaliação proposta na disciplina de Algoritmos e Estruturas de Dados II no semestre 2023/2, as regras do problema e solução para o mesmo, estarão descritas detalhadamente no artigo, contendo a explicação da estrutura de dados utilizada e o motivo da escolha do modelo.

Introdução/Descrição

Na rara ocasião da Grande Convenção dos Alquimistas, os participantes se reúnem em uma celebração única, onde ocorre uma competição que consiste em obter a melhor ideia de receita para chegar ao ouro, as receitas são simples: elementos químicos devem ser transformados uns nos outros em várias quantidades até chegar a ouro, porém todas as receitas começam com hidrogênio. O trabalho consiste em desenvolver um algoritmo capaz de determinar a quantidade de hidrogênio necessário para alcançar uma unidade de ouro passando por todas as transformações fornecidas.

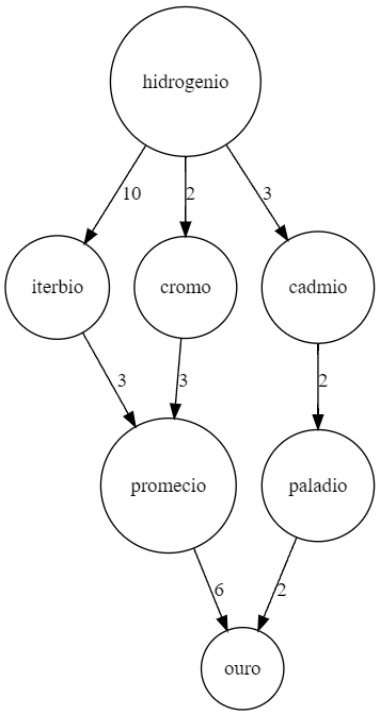
Dentro do escopo da disciplina de Algoritmos e Estruturas de Dados II, o primeiro problema a ser discutido, seria a estrutura de dados ideal para solucionar o problema, buscando uma otimização boa e um tempo de execução razoável, tendo em vista que a ideia do algoritmo, é basicamente realizar uma sequência de operações com os elementos partindo da ideia de que temos um elemento inicial, que seria o hidrogênio e a partir dele seriam geradas camadas até chegar ao ouro, como no exemplo da figura 1.

A conclusão do exemplo mostrado seria que são necessárias 228 unidades de hidrogênios para produzir uma única unidade de ouro. Esse resultado foi obtido considerando as diferentes etapas de formação, começando com o hidrogênio e passando por itérbio, cromo, cádmio, promécio, paládio e, finalmente, ouro. Cada uma dessas transições envolve uma combinação específica de elementos e quantidades, levando à conclusão de que 228 átomos de hidrogênio são requeridos para formar uma unidade de ouro.

processo de raciocínio:
10 Hidrogênios = 1 itérbio
2 Hidrogênios = 1 cromo
3 Hidrogênios = 1 cadmio
3 itérbios e 3 cromo = 1 promécio -> 36 hidrogênios
2 cadmio = 1 paládio -> 6 hidrogênios
6 promécio e 2 paládio = 1 ouro -> 228 hidrogênios

[*]marcelo.felisberto@edu.pucrs.br
[†]arthur.real@edu.pucrs.br

Figura 1: Graphviz Online



Solução desenvolvida

Após uma reflexão profunda sobre o desafio em questão, tornou-se evidente que, para alcançar uma otimização verdadeiramente eficaz, é importante eleger uma estrutura de dados que proporcione esta ideia de relação entre strings com um valor (arresta) direcionado entre elas, esta ideia é simples de raciocinar ao olharmos para a estrutura de grafos direcionados e valorados. Utilizando a ideia do algoritmo *DepthFirstSearch* (dfs) visto em aula, se tornaria relativamente mais simples realizar as operações necessárias para chegarmos ao ouro. Uma vez que o objetivo central reside na constante multiplicação e acumulação desses valores partindo de nossa raiz (hidrogênio) presente no nosso grafo. Um *EdgeWeightedDigraph* se revelou uma solução verdadeiramente notável, por conta de seus atributos e algoritmos vistos em aula que facilitariam a manipulação de nossas informações, como optamos por realizar o trabalho a partir da linguagem de programação *java*, utilizamos a classe *BigInteger* para podermos trabalhar com números muito grandes, uma vez que a classe *Integer* não era capaz de resolver todos os casos propostos.

Alogoritmo presente na EdgeWeightedDigraph

```
1  public BigInteger hidrogenioToGold(){
2      Map< String ,BigInteger>  dic = new HashMap≠ () ;
3      BigInteger totalHidrogenios = calculateHidroNecssary("hidrogenio" ,dic) ;
4      return totalHidrogenios ;
5  }
6   private BigInteger calculateHidroNecssary(String elemento ,Map< String ,BigInteger>  dic){
7      BigInteger totalH = BigInteger.valueOf(0) ;
8
9      if(dic.containsKey(elemento)){
10        return dic.get(elemento) ;
11     }
12
```

```
13      if(!elemento.equals("ouro")){
14        for(Edge e : this.getAdj(elemento)){
15          BigInteger hidrogenios = e.getWeight().multiply(calculateHidroNecssary(e.getW() ,dic)) ;
16          totalH = totalH.add(hidrogenios) ;
17        }
18      }
19      else{
20        return BigInteger.valueOf(1) ;
21      }
22
23      dic.put(elemento , totalH) ;
24      return totalH ;
25    }
```

O algoritmo utiliza uma abordagem recursiva para calcular a quantidade total de hidrogênio necessária para produzir ouro. Representamos as relações entre elementos por meio de um grafo, onde cada nó representa um elemento químico e as arestas indicam a quantidade de hidrogênio ou de um elemento necessária para a transformação. A função principal, *hidrogenioToGold*, inicia o processo chamando a função recursiva *calculateHidroNecssary* com o elemento inicial "hidrogênio". Esta última função percorre as arestas do grafo, multiplicando a quantidade de hidrogênio em cada aresta pelo resultado da chamada recursiva para o próximo elemento. O processo continua até que o elemento seja "ouro", momento em que a função retorna 1, indicando a quantidade de hidrogênio necessária para produzir ouro. Em resumo, o algoritmo calcula de forma eficiente a quantidade de hidrogênio requerida para a produção de ouro com base nas relações químicas estabelecidas no grafo.

Primeiramente ao chamarmos a função publica *hidrogenioToGold*, é criado um dicionario para podermos armazenar um par <*Sring,BigInteger*> que será utilizado para otimizar o processo, pois evitará repetições de calculo e de busca, pois nele armazenaremos os elementos e a quantidade de hidrogênios relativa para a criação deles, ou seja quando formos utilizar um elemento que já foi previamente calculado, teremos o valor de hidrogênios guardado para uso.

Por consequinte, é chamada a função privada *calculateHidroNecssary*, passando o dicionario criado e uma string que representa o elemento em que estamos trabalhando. O primeiro passo que ocorre no método, é a criação da variável *totalH*, que representa a quantidade final de hidrogênios que será retornada ao fim da execução de nossa função, logo após utilizamos um *if* para vermos se o elemento a ser trabalhado já consta no nosso dicionario(este if é necessário para a otimização do código, evita *loops* desnecessários).

Toda via, agora que passamos pela parte inicial do método, começa de fato o verdadeiro processo para o calculo da quantidade necessária de hidrogênios, utilizamos um *if* para perguntarmos se o elemento de trabalho atual é o ouro, pois neste caso, pulamos o *loop* e retornamos 1, caso este não venha a ser o caso, entramos em um *for* que passa por todas as arrestas adjacentes do elemento sendo trabalhado e faz o calculo chamando a si próprio (recursão), e acumulando o valor em *totalH*, por fim retornamos o valor de hidrogênio necessário que está armazenado na variável.

Resultados

Foi criada uma tabela que demonstra os resultados obtidos do algoritmo criado dentro da *EdgeWeighTedDigraph*, foram disponibilizados um total de 13 arquivos para testes,a tabela mostra a quantidade de hidrogênios necessários, tempo de execução(apenas o tempo gasto dentro do algoritmo) e a quantidade de operações, sendo realizado 3 testes e colocando um intervalo de tempo entre o resultado mais rápido e o resultado mais demorado.

Foi feito uma contagem das operações para demonstrar pelo que o algoritmo está passando, a

complexidade de tempo do algoritmo é dominada pelo número de arestas no grafo, ou seja,**O(n)**, sendo **n** o número de arestas no nosso grafo. Podemos observar que na grande maioria dos casos o número de operações aumenta de acordo com a quantidade de arrestas presentes no grafo. Ele possui uma complexidade muito baixa, pois o for irá rodar o número de vezes necessários para percorrer o grafo todo(O(n)) e os "ifs e elses"seriam constantes(**O(1)**) para cada vez que passar pelo *for*

Caso de Teste	Qtd de hidrogênio necessária	Tempo	Operações
casoTeste	16272	1ms < Tempo < 2ms	23
casoa5	102744	2ms < Tempo < 3ms	110
casoa20	800770	2ms < Tempo < 4ms	355
casoa40	8256835	4ms < Tempo < 5ms	635
casoa60	1641700	2ms < Tempo < 3ms	345
casoa80	69922658719	4ms < Tempo < 6ms	1138
casoa120	721329018682809	4ms < Tempo < 5ms	1541
casoa180	8428729212204778	5ms < Tempo < 6ms	2111
casoa240	437216915509229458771143884	5ms < Tempo < 7ms	2741
casoa280	1258278279472727281992929862080l	6ms < Tempo < 8ms	3098
casoa320	958071316502331277458891480549 4	7ms < Tempo < 9ms	3539
casoa360	250822397644304265277554478037890 25	7ms < Tempo < 9ms	3799
casoa400	630330556280327559061316284071085017 16	7ms < Tempo < 10ms	4195

Os testes foram feitos em um computador com o processador: "9th Gen Intel(R) Core(TM) i5-9600K @ 3.70GHz, e demonstram que o algoritmo é super eficaz, tanto com pequenas quantidades de elementos, quanto com grandes volumes de dados pois ele é de uma complexidade O(n).

Conclusões

O algoritmo foi exposto a testes extensivos com diversas entradas de diferentes tamanhos de combinações de elementos até chegar ao ouro e chegamos a conclusão de que o algoritmo demonstrou um desempenho excelente, mesmo com grandes volumes de dados. O número de operações executadas foi praticamente linear, e o tempo de execução foi muito rápido, mostrando a eficiência da nossa solução. Porém, isto só ocorre pois utilizamos uma estrutura mais eficiente de dados para execução do trabalho, e fizemos um algoritmo previamente planejado, o uso de Hashmap certamente foi crucial para a otimização, pois corta em grande parte o número de loops que aconteceriam dentro do *for* do algoritmo. Antes da optimização, os casos mais complexos podiam levar até 5 minutos para finalizar todos os cálculos e chamadas de método, isto ocorreu por conta da falta de um dicionário que armazenava propriamente os valores de hidrogênio necessário para qualquer elemento específico, ou seja, anteriormente o programa estava funcional mas muito ineficiente por conta de necessitar voltar e fazer muitos loops no programa.