Leonardo Chou da Rosa[1]

Algoritmos e Estrutura de Dados II - Trabalho 2

Faculdade de Engenharia da Computação — PUCRS

16 de novembro 2023

Resumo

O artigo a seguir apresenta a solução para o enunciado do Trabalho 2 da disciplina Algoritmos e Estrutura de Dados II, onde os alquimistas querem achar a quantidade de Hidrogênio necessária para criar Ouro. A solução proposta utiliza o conceito de grafos como base, e são demonstrados a lógica por trás do algoritmo, exemplos de execução e possíveis melhorias e otimizações.

Introdução

Há cada 100 anos, acontece um evento marcante para os alquimistas: a Grande Convenção dos Alquimistas. O grande prêmio de tal evento é a descoberta da receita mais eficiente de gerar o elemento Ouro. Para achar a melhor receita, a única restrição colocada em cima dos alquimistas é:

- A receita deve partir do elemento Hidrogênio.

Para resolver o problema proposto a utilização de um grafo dirigido (dígrafo)[2] é necessária, onde cada elemento na receita será considerado como um "nodo" no grafo e os seus pesos associados correspondem a quantidade do elemento que é preciso para criar o próximo elemento. Para facilitar o entendimento do processo de soma, a figura 1 demonstra como que é calculado o valor de hidrogênio necessário para chegar no Ouro:

[1] l.chou@edu.pucrs.br

[2] Conjunto de vértices conectados por arestas; grafos dirigidos contém um vértice inicial e um vértice final, determinados pela direção da aresta.

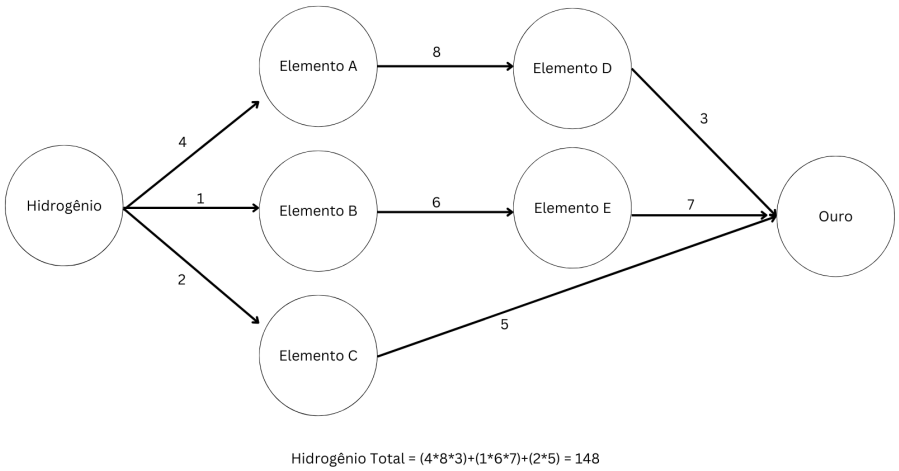

Figura 1: Exemplo de Resolução do Problema Proposto

Como está descrito na imagem, a forma de achar a quantidade de Hidrogênio necessária para construir a receita é pelo seguinte:

- Multiplicar os pesos dos nodos conectados em ordem até chegar no Ouro
- Repetir o processo para todos os nodos de Hidrogênio
- Somar o resultado de todas as multiplicações

A importância de resolver o problema proposto é para desenvolver habilidades de gerenciamento de uma quantidade enorme de dados. Mesmo com o enunciado proposto ter relações com a ficção, uma aplicação similar do problema no mundo real é no desenvolvimento de aplicativos que calculam rotas em um mapa, como o Waze e o Uber. A quantidade de Hidrogênios podem indicar o tempo de viagem entre um ponto e o outro, e ao invés de calcular o tempo de todas as rotas, o programa pode calcular o menor caminho para chegar no destino.

Algoritmo - C + +

"Esqueleto" do Algoritmo:

Na linguagem C + +, a forma mais simples de criar um grafo é criando um *map*[3], com a chave sendo o nodo atual e o conteúdo da chave sendo o nodo seguinte. Considerando que em diversos casos de execução um nodo pode apontar para inúmeros outros nodos, é necessário criar um *vector*[4] que armazena todas as conexões de nodos possíveis a partir do nodo atual.

Considerando que os nodos armazenam duas informações cruciais (nome e peso), foi desenvolvida uma classe interna ao grafo chamada **nodo**, que armazena os nomes e os pesos necessários para a operação. Contrariamente, a chave do mapa não necessitava das duas informações para funcionar, portanto foi utilizado somente o nome de cada nodo para identificar em qual posição do grafo o algoritmo está percorrendo naquele momento.

Leitura dos Arquivos Caso Teste:

[3] Estrutura de dados que armazena uma chave e uma informação somente acessível através da chave. https://cplusplus.com/reference/map/map/

[4] Lista que armazena elementos dinamicamente.

Para ler os arquivos fornecidos para a testagem do algoritmo, foi necessário desenvolver dois métodos internos da classe **grafo** que conseguem ler os arquivos testes e armazenar as informações dentro do grafo:

- add(), que insere o nodo resultante ao nodo inicial
- leitura(), que abre os arquivos de teste e constroem o grafo a partir do método add()

O método add() é bem simples; a chave do grafo (nodo inicial) e o nodo resultante são passados por parâmetro e o nodo resultante é inserido dentro da lista de nodos do nodo inicial.

O método leitura() abre o arquivo, e armazena as informações dentro de dois *vectors*, um que recebe os nomes dos nodos e o outro que recebe os pesos dos nodos. Quando o leitor chega até o caractere "->" (indicador que o próximo elemento será o elemento resultante), o leitor recebe o valor do elemento resultante e executa o método add() até os dois vetores contendo as informações dos nodos iniciais cheguem até o fim.

Cálculo da Quantidade de Hidrogênios:

Após o grafo estiver cheio, a função print() é executada, onde é calculado a quantidade de Hidrogênios necessárias para chegar no Ouro. O algoritmo recebe uma *string* como parâmetro, que indica a chave para acessar o grafo do elemento inicial. No caso do problema proposto, a string inicialmente será "hidrogênio". O método então inicia declarando duas variáveis: a variável soma, que calcula o resultado das multiplicações dos nodos e a variável total, que soma todas as multiplicações calculadas pela soma. A função então entra em um *for-loop*, que só termina após todos os nodos da string passada por parâmetro forem chamados.

Para calcular os valores da soma uma recursão é necessária, pois há duas variáveis necessárias para o cálculo: o peso do nodo sendo acessado e a soma total do nodo após o nodo acessado. Quando a recursão chega até o Ouro, ela retorna o valor de 1, pois não há nodos depois do Ouro. O valor de soma então recebe a multiplicação do peso do nodo sendo acessado com a soma total do nodo seguinte. Após todas as somas de um caminho de Hidrogênio terminarem de ser calculadas, a variável total recebe a soma final. Esse processo se repete até que todos os caminhos de hidrogênio possíveis sejam calculados.

Pseudocódigo:

```
1       print(string elemento) {
2         se o elemento for "ouro" {
3             retorna 1; //chegou ao final
4         }
5         valor de soma = 1;
6         valor total = 0;
7         para todos os nodos dentro do vetor de nodos do elemento {
9               valor de soma recebe a multiplicação do peso do nodo atual
                com a soma do próximo nodo (chamada recursivamente);
10              adiciona o valor total com o valor da soma;
11        }
12       retorna o valor total;
13      }
```

Números Grandes:

A linguagem de programação C++ contém um limite no tamanho de seus inteiros, onde não é possível representar números maiores que $2^{64} - 1$, ou 18, 446, 744, 073, 709, 551, 615. Nos casos de testes maiores, a quantidade de Hidrogênios necessários superam esse limite, portanto é necessário utilizar uma biblioteca externa. A biblioteca utilizada é chamada "Arbitrary-precision integer and rational arithmetic" que resolve esse problema.

Exemplo de Execução:

Para demonstrar o funcionamento do algoritmo, os valores da imagem previamente exibida serão usados:

4 Hidrogênio -> 1 Elemento A
1 Hidrogênio -> 1 Elemento B
2 Hidrogênio -> 1 Elemento C
8 Elemento A -> 1 Elemento D
6 Elemento B -> 1 Elemento E
3 Elemento D 7 Elemento E 5 Elemento C -> 1 Ouro

Leitura do Arquivo:
Mapa do Hidrogênio A armazena vector[Elemento A com peso 4, Elemento B com peso 1, Elemento C com peso 2]
Mapa do Elemento A armazena vector[Elemento D com peso 8]
Mapa do Elemento B armazena vector[Elemento E com peso 6]
Mapa do Elemento C armazena vector[Ouro com peso 5]
Mapa do Elemento D armazena vector[Ouro com peso 3]
Mapa do Elemento E armazena vector[Ouro com peso 7]

Cálculo:

Hidrogênio[0] (Elemento A) = 4 Soma = 4 * 24 (resultado da soma do próximo nodo)	Hidrogênio[1] (Elemento B) = 1 Soma = 1 * 42 (resultado da soma do próximo nodo)	Hidrogênio[2] (Elemento C) = 2 Soma = 2 * 5 (resultado da soma do próximo nodo)
Elemento A[0] (Elemento D) = 8 Soma = 8 * 3 (resultado da soma do próximo nodo)	Elemento B[0] (Elemento E) = 6 Soma = 6 * 7 (resultado da soma do próximo nodo)	Elemento C[0] (Ouro) = 5 Soma = 5 * 1 (chegou ao Ouro; retorna 1)
Elemento D[0] (Ouro) = 3 Soma = 3 * 1 (chegou ao Ouro; retorna 1)	Elemento E[0] (Ouro) = 7 Soma = 7 * 1 (chegou ao Ouro; retorna 1)	

Total = 96 + 42 + 10 = 148

Análise

Para determinar a eficiência do algoritmo desenvolvido, foram coletadas duas informações cruciais:

- A quantidade de tempo que o algoritmo demora para calcular os casos teste
- A quantidade de *loops* que o algoritmo faz para terminar todos os cálculos

Tempo:

O cálculo do tempo foi realizado pela função de clock da biblioteca <chrono> da linguagem C + +. Dentro da biblioteca, há uma função chamada *high_resolution_clock*, que recebe o tempo atual do computador. Para achar o tempo de execução, é necessário coletar e armazenar o valor do tempo antes da

função ser executada e armazenar o valor do tempo depois da função ser executada. Subtraindo o valor final pelo valor inicial, é possível saber o tempo de execução do programa.

Caso Teste	Tempo	Resultado
casoenunciado	4 ms	16272
casoa5	8 ms	102744
casoa20	10 ms	800770
casoa40	13 ms	8256835
casoa60	10 ms	1641700
casoa80	46 ms	69922658719
casoa120	237 ms	721329018682809
casoa180	1079 ms (1.079 segundos)	84287292122047778
casoa240	71922 ms (1.1 minutos)	437216915509229458771143884
casoa280	510966 ms (8.5 minutos)	125827827947272728199292986208801
casoa320	1950346 ms (54 minutos)	958071316502331277458891480 5494
casoa360	16237661 ms (4 horas e 30 minutos)	250822397644304265277554478037890 25
casoa400	N/A	N/A

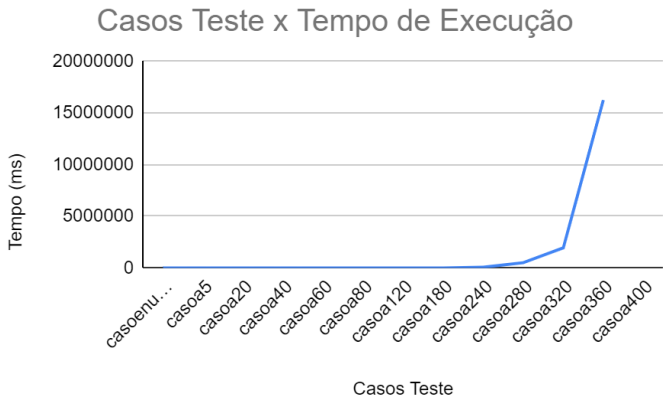

Figura 2: Gráfico mostrando o Tempo de Execução de cada Caso Teste

Conforme os resultados, o tempo de execução dos algoritmos depende do tamanho do grafo sendo utilizado. Como pode se observar na figura 2, após um certo limite o tempo começa a crescer de forma exponencial, estando relacionado diretamente com o tamanho do grafo. Existe um limite de entrada nesse algoritmo para considerar ele aceitável; é possível notar que entre o casoa180 e o casoa240

o tempo de execução aumenta em 6500%, indo de 1 segundo para 1 minuto. Esse padrão continua, pois qualquer caso de teste acima do casoa180 demora mais do que 1 minuto para ser executado.

O crescimento exponencial somente afeta a coleta de dados após o casoa180; mesmo com o crescimento exponencial nos casos anteriores, o tempo de execução ainda era aceitável, pois todos os valores estavam abaixo de 1 segundo facilitando a execução do cálculo. Com base nos resultados coletados, as seguintes conclusões são afirmadas:

1) O algoritmo em questão não está propriamente otimizado; existe um limite no tamanho do grafo que impossibilita a sua viabilidade.
2) O crescimento do tempo de execução é exponencial; após um certo ponto ele chega a demorar múltiplas horas para terminar o cálculo.

Loops:

A quantidade de *loops* foi determinada pela quantidade de vezes que a função print() foi chamada. Devido à recursão utilizada no programa, para achar esse valor uma variável global chamada *loops* foi utilizada, que incrementa por 1 a cada chamada da função. Os valores de *loop* calculados estão na seguinte tabela:

Caso Teste	Loops
casoenunciado	35
casoa5	167
casoa20	588
casoa40	1279
casoa60	675
casoa80	7633
casoa120	48272
casoa180	236949
casoa240	15620729
casoa280	109219294
casoa320	238027912
casoa360	1604777611
casoa400	N/A

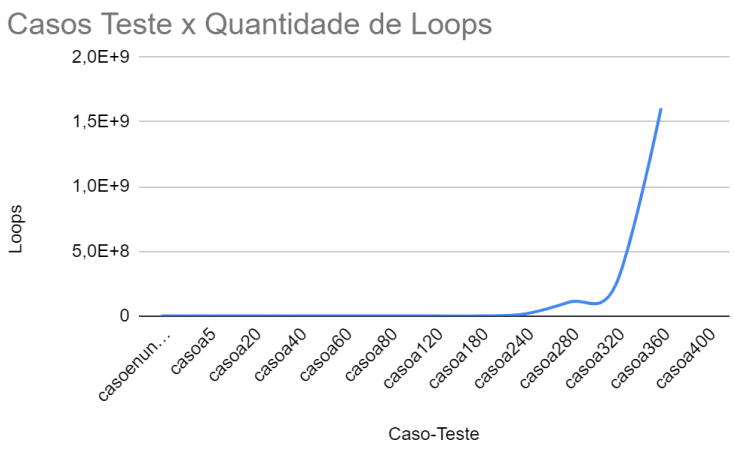

Figura 3: Gráfico mostrando a Quantidade de Loops para cada Caso Teste

Similarmente ao tempo, a quantidade de *loops* aumenta exponencialmente dependendo do tamanho do grafo. A diferença entre os valores adquiridos agora e os valores do tempo é que o crescimento da quantidade de *loops* é aparente até entre o primeiro e o segundo caso, e não somente após um certo ponto. Os dados coletados novamente demonstram que o algoritmo não está otimizado o suficiente para ser utilizado em prática, pois o valor de *loops* é diretamente proporcional com o tempo de execução.

Conclusão

O algoritmo desenvolvido para resolver o problema estabelecido no enunciado é uma solução simples, que consegue alcançar o seu objetivo mas está em um estado primitivo e não otimizado. O algoritmo funciona muito bem para operações em escalas pequenas, como até o casoa180 fornecido, pois ele resolve o problema em uma velocidade muito boa. A partir desse ponto, o algoritmo demora um tempo inviável para resolver o problema, chegando a tempos de execuções de horas para conseguir os resultados dos grafos maiores.

Uma forma possível de otimizar o código seria reduzir a quantidade de *loops* que o algoritmo utiliza quando ele é executado. Como a análise prévia havia descrito, os *loops* estão diretamente proporcionais ao tempo de execução. Sendo o tempo o fator mais importante quando se trata de otimização, se os *loops* diminuem, o tempo de execução também irá diminuir. Utilizando os dados previamente coletados como base, uma quantidade de loops de <200,000 para todos os casos irá torná-lo extremamente eficiente, pois todos os casos testados que contém um valor menor a esse executaram em 1 segundo ou menos.

Referências

[1] “Maximum Value of Unsigned Long Long Int in C++.” GeeksforGeeks, GeeksforGeeks, 3 Dec. 2020, www.geeksforgeeks.org/maximum-value-of-unsigned-long-long-int-in-c/.
[2] Possiblywrong. “Arbitrary-Precision Rational Arithmetic in C++.” Possibly Wrong, 14 Oct. 2018, possiblywrong.wordpress.com/2018/09/02/arbitrary-precision-rational-arithmetic-in-c/.