Trabalho 2 – Algoritmo e Estrutura de Dados II

Marcelo N. Da Silva

Faculdade de Informática — PUCRS

21 de novembro de 2023

Resumo

Esse relatório descreve a implementação da solução para o segundo trabalho proposto na disciplina de Algoritmo e Estrutura de Dados II, que consiste em desenvolver um algoritmo que realiza o calculo da quantidade de hidrogênios utilizadas para produzir uma unidade de ouro. Para o desenvolvimento, foi utilizada a linguagem Python, e para a representação dos dados foi utilizado um grafo dirigido e busca em profundidade.

Introdução

O objetivo deste estudo é desenvolver um algoritmo capaz de somar a quantidade de hidrogênio necessária para produzir uma unidade de ouro em uma receita específica. As receitas contêm vários elementos químicos que devem ser combinados uns com os outros em diferentes quantidades até produzir uma unidade de ouro. Por exemplo, as receitas começam com hidrogênio e são usadas em diferentes quantidades para produzir outros elementos até chegarem ao objetivo final. A execução incluí realizar a leitura do arquivo de teste e implementar um grafo que inicializa vazio e é construído a partir da leitura do arquivo após a compreensão do desafio.

Cada linha do arquivo é processada e os vértices e arestas são adicionados ao grafo. Implementando também um conjunto de visitantes, que pode rastrear os nodos que já foram visitados durante a execução do algoritmo utilizando uma busca em profundidade. Além, do algoritmo que calculará a quantidade de hidrogênio necessária. Para o desenvolvimento do algoritmo, foi utilizada a linguagem programação Python e está implementado nos arquivos ‘graph.py’ e ‘main.py’ em anexo.

A partir do desenvolvimento, os resultados obtidos com a execução de cada caso de teste serão analisados.

Solução

O algoritmo foi desenvolvido a partir da leitura e processamento das linhas de um arquivo uma a uma. O grafo é atualizado de acordo com os vértices e arestas encontrados, em cada uma dessas linhas divididas. Como um dicionário, o grafo tem vértices para cada chave e uma lista de valores para cada vértice. A busca em profundidade é usada para percorrer todos os caminhos possíveis no grafo para calcular a soma dos produtos de todos os caminhos no grafo. Para explorar todos os caminhos possíveis, a função multiplica os valores das arestas, verificando se o nodo não possui vizinhos, e retornando o produto acumulado até esse ponto.

Dentro da classe ‘Graph’, a função ‘input_string’ é responsável pela construção do grafo. Esta função divide a *string* em partes (list_str e valores_origem), e determina

se a linha do arquivo representa uma relação entre vértices indiretos ou uma aresta direta. Como esboçado no código abaixo.

enquanto teste seja verdadeiro faça

list_str ← string.split("->")

valores_origem ← list_str[0].split(" ")

se o tamanho de valores_origem for igual a 3 então

Graph.add_element(graph, string)

senão

para i de 0 até o tamanho de valores_origem - 2 com passo 2 faça

nova_string ← valores_origem[i] + " " + valores_origem[i + 1] + " ->" + list_str[1]

Graph.add_element(graph, nova_string)

fim

fim

fim

A aresta direta é indicada pelo comprimento de 'valores_origem' de 3 e a função 'add_element' é utilizada para adicionar a aresta ao grafo. Por outro lado, se a linha representa uma relação entre vértices indiretos, um loop é usado para criar uma *string* (nova_string) para cada par de vértices. Além disso, a função 'add_element' é chamada para adicionar cada elemento ao grafo. A função verifica se os vértices de origem e destino já estão presentes no grafo e, se não estiverem, os adiciona. Recebe como argumento o grafo e uma *string* que representa uma linha do arquivo. A função 'add_element' está representada abaixo.

Função estática adicionar_elemento (gráfico, string):

valores ← string.split(" ")

valor_aresta, elem_origem, _, _, elem_destino ← valores

gráfico.definirPadrão(elem_origem, [])

gráfico.definirPadrão(elem_destino, [])

gráfico[elem_origem]. adicionar((elem_destino, valor_aresta))

//.definirPadrão() é usado para garantir que as chaves existam no dicionário antes de acessá-las.

A função ‘calculate_sum’ foi implementada para realizar o cálculo da soma dos produtos de todos os caminhos no grafo. A função recebe um grafo, o nodo para iniciar a busca, o produto acumulado ao longo do caminho até o nodo atual e um conjunto que mantém os nodos que foram visitados durante a busca de profundidade.

Começa marcando o nodo atual como visitado e verificando se ele tem arestas de saída, ou vizinhos. Ela retorna o produto acumulado se não houver vizinhos. A função itera sobre todos os vizinhos se houver. A função faz uma chamada recursiva para atualizar o produto acumulado para o caminho atual e adicioná-lo ao resultado total se o vizinho ainda não foi visitado. A soma total dos produtos de todos os caminhos possíveis a partir do nó inicial é retornada pela função.

Finalmente, um conjunto de visitantes é mantido para evitar ciclos durante a busca, e uma cópia desse conjunto é passada repetidamente para evitar interferências entre chamadas recursivas. Essa implementação esta esboçada no pseudocódigo abaixo.

```
função estática calcular_soma (gráfico, nó, produto_atual=1, visitados=nulo):
    se visitados for nulo, então
        visitados ← conjunto vazio
    fim
    visitados.adicionar(nó)

    se não gráfico[nó] então
        retornar produto_atual
    fim

    soma_total ← 0

    para vizinho, valor em gráfico[nó] faça
        se vizinho não estiver em visitados então
            produto_caminho ← produto_atual * converter_para_inteiro(valor)
            soma_total += calcular_soma(gráfico, vizinho, produto_caminho, copiar(visitados))
        fim
    fim

    retornar soma_total
```

Após isso, foi possível usar o algoritmo para os outros casos de teste. No entanto, como o caso de teste "casoc360.txt" continha mais informações, não foi possível estimar seu tempo de execução. No entanto, os resultados indicarão que não foi possível executar esse caso de teste em particular.

Resultados

A seguir serão apresentados os resultados obtidos com a execução dos casos de testes. Para cada caso, o resultado é exibido no terminal informando a quantidade de hidrogênios necessários para realizar aquela receita e o tempo de execução. Para os casos de testes 40, 80, 120, 180, 240, 280 e 320, foram obtidos os seguintes resultados respectivamente:

casoc40.txt

Hidrogênios: 15192249

Tempo de Execução: 0.002073049545288086

casoc80.txt

Hidrogênios: 27221484395

Tempo de Execução: 0.005009651184082031

casoc120.txt

Hidrogênios: 2257966552765316

Tempo de Execução: 0.012849807739257812

casoc180.txt

Hidrogênios: 1295127372563879647489923

Tempo de Execução: 0.8889884948730469

casoc240.txt

Hidrogênios: 55577785066027869882239842

Tempo de Execução: 4.374760150909424

casoc280.txt

Hidrogênios: 250439366313585404923320 40

Tempo de Execução: 26.99396252632141

casoc320.txt

Hidrogênios: 17986055867306301215241957896478

Tempo de Execução: 71.08144688606262

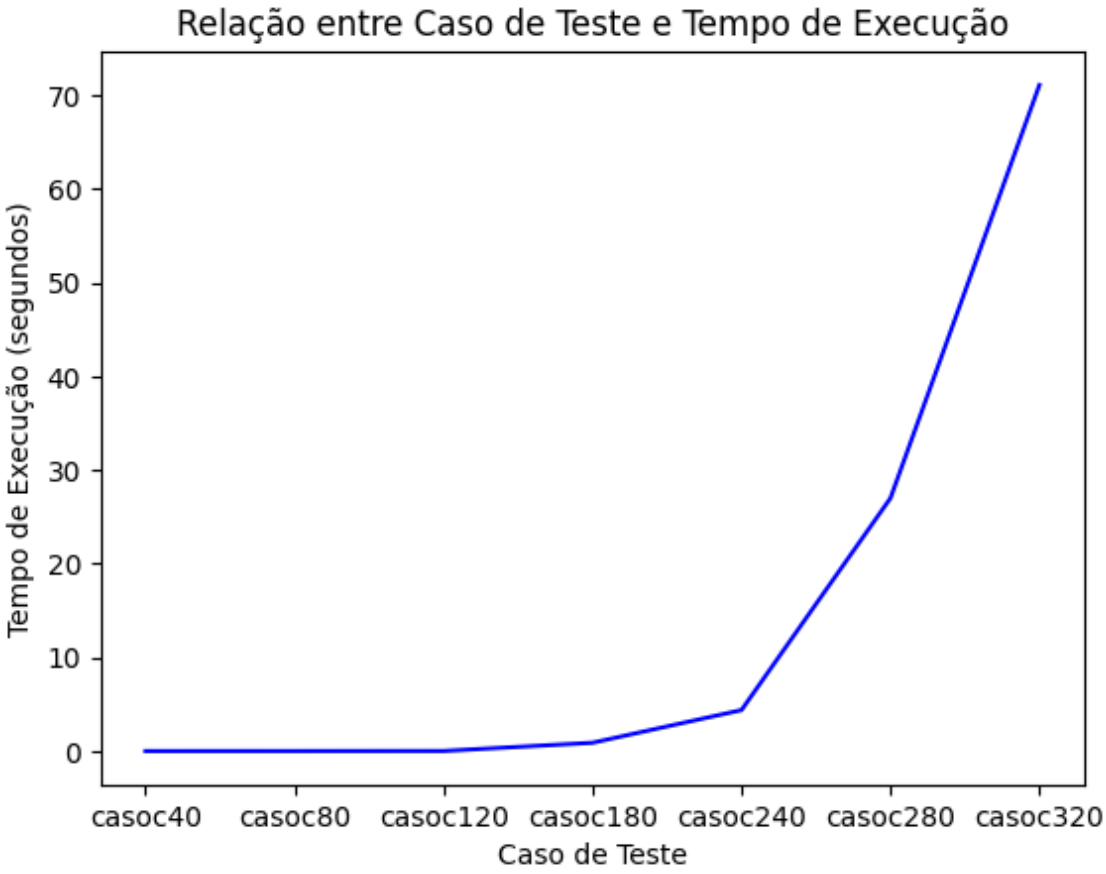
A execução do caso teste “casoc360.txt” não retornou o resultado esperado, mesmo após um período significativo de execução, e não retornou nenhuma mensagem de erro de execução. Até a entrega do trabalho, não foi possível identificar a falha, solucionar esse problema ou otimizar o algoritmo.

Conclusões

Mesmo faltando a execução do último caso de teste, foi possível compreender a aplicação e funcionamento da estrutura de grafos e o problema apresentado. A complexidade do algoritmo depende principalmente do número de arestas e vértices no grafo. Portanto, a complexidade total seria O(V+A), onde V é o número de vértices e A é o número de arestas no grafo.

A fim de verificar a complexidade no programa, com base nos resultados da implementação, foi realizada a computação do tempo de execução de cada teste, e elaborado um gráfico como consta a figura 1 com as informações da evolução de cada teste, relacionando o teste e o tempo de execução.

Figura 1. Tempo de Execução



Ademais, com a implementação desse trabalho, foi possível desenvolver conhecimentos de manipulação de *Strings* e grafos, como realizar cálculo de soma em grafos e adição de elementos ao grafo. Esses conhecimentos podem ser integrados em sistemas mais amplos para realizar tarefas específicas relacionadas a grafos e manipulação de dados em Python.

Referências

[1] A Biblioteca Padrão do Python: “Exceções embutidas”. Disponível em: Acesso em 15 Nov. 2023.

[2] A Biblioteca Padrão do Python: “Parâmetros e funções específicas do sistema”. Disponível em Acesso em 15 Nov. 2023.