Thiago Zilberknop[1]

Algoritmos e Estrutura de Dados II - Trabalho 2

Faculdade de Engenharia da Computação — PUCRS

11 de Novembro 2023

Resumo

O seguinte artigo descreve o processo para solução do problema proposto pelo Trabalho 2 da disciplina de Algoritmos e Estruturas de Dados II, ministrada pelo professor Alexandre Agustini na Pontifícia Universidade Católica do Rio Grande do Sul. O trabalho trata de formular e analisar um algoritmo capaz de processar grafos direcionados e valorados de diferentes tamanhos que descrevem uma "receita alquímica" para transformar hidrogênio em ouro, retornando o número necessário de átomos de hidrogênio para, através da receita, conseguir um átomo de ouro.

Introdução

Alquimistas se reúnem de tempos em tempos para discutirem as suas mais novas receitas para aquisição de ouro através da transmutação contínua de hidrogênio em outra matéria, e essa outra matéria em outra, e assim em diante até, finalmente, a transmutação desta em ouro. É um plano de fundo cartunesco e divertido para o assunto que nos é interessante nesse trabalho: como tratar um grafo (as receitas, no caso) direcionado, valorado, e possivelmente muito grande?

Primeiramente, há de ser entendida a estrutura genérica de qualquer receita que possa vir a existir: todos os alquimistas partem do hidrogênio e tentam, passando por vários outros átomos, alcançarem o ouro. Isto é, no grafo em si, o vértice que contém hidrogênio sempre terá um caminho para o vértice que contém ouro, e vice-versa. Porém, um átomo de ouro só poderá ser criado se todos os ingredientes necessários tiverem sido produzidos através das reações descritas na receita. Aproveitando o jargão de grafos, uma *visita*[2] a um nodo qualquer só é permissível se os *pais* — nodos que apontam para aquele nodo — já foram eles mesmos visitados. Logo, uma vez que o nodo de ouro sempre é o "último" nodo de um grafo, todos os outros nodos precisam ser visitados antes dele.

Em vista disso, e de algumas particularidades das receitas usadas como casos de teste para o algoritmo, as regras de criação e de consulta do grafo a ser construído podem ser resumidas da seguinte maneira:

1) O vértice de hidrogênio sempre é o vértice de partida, pois nenhum outro vértice aponta para ele;
2) O vértice de ouro sempre é o último vértice a ser visitado, pois ele aponta para nenhum outro vértice;
3) Sempre existe um caminho entre os vértices de hidrogênio e de ouro;

[1] thiago.zilberknop@edu.pucrs.br

[2] Uma visita a um nodo indica o momento no qual um algoritmo executa as operações que precisam ser feitas naquele nodo, podendo estas operações serem de qualquer tipo. No caso em questão, uma visita consiste em descobrir o custo em hidrogênio de percorrer o grafo até o vértice sendo visitado

4) Todo vértice só pode ser visitado se os outros que apontam para ele já houverem sido visitados;
5) Todo vértice tem um custo de "produção" igual ao valor das arestas que apontam para ele;
6) Todo custo indica o número de vezes que um vértice "catalisador" precisa ser visitado para que o vértice do "produto" seja visitado;
7) Toda "produção" sempre resulta em um único produto e em uma única unidade daquele produto, não havendo possibilidade de uma "reação" resultar em múltiplos produtos e múltiplas unidades de produtos;
8) Todas as receitas são acíclicas[3];

Com estas regras em mente, foi possível dar início ao processo de resolução do problema, começando pela escolha de língua — Python, pela facilidade de escrita — e pela estruturação em código dos grafos em si.

Classes

No momento inicial, foi de destaque a escolha de como estruturar os grafos e seus nodos. Python, tendo suporte ao paradigma de Programação Orientada a Objetos (POO), possibilitou a construção de uma classe **`Graph`** e de uma classe interna **`Node`**, a primeira contendo apenas uma lista[4] de objetos **`Node`**, e a segunda contendo todas as informações de um vértice qualquer do grafo, como visto no pseudo-código abaixo:

```
1       class Graph:
2         class Node:
3           Na construção de uma instância de Node:
4             Uma string chamada data é criada
5             Um dict de objetos da classe Node chamada edges é criada
6             Uma list de objetos da classe Node chamada fathers é criada
7         Na construção de uma instância de Graph:
8           uma list de objetos da classe Node chamada vertices é criada
```

Enquanto que é plenamente possível criar um algoritmo que não siga o paradigma de POO — especialmente considerando que a complexidade de dados não é grande, uma vez que o grafo possui apenas um atributo, e os nodos três — é conveniente segui-lo devido a compartimentalização de todos os elementos do grafo, facilitando a compreensão de cara parte e como também a soma de todas as partes.

Observa-se a decisão de usar *dicts*[5] na classe **`Node`** para guardar tanto um nodo (a chave) como o custo (o valor) associado a ele para visitá-lo a partir do vértice que o tem dentro do seu próprio dicionário **`edges`**. Já a outra estrutura, **`fathers`**, contém apenas os nodos que apontam para aquele vértice, não importando a ordem de vértices na lista nem o peso das arestas destes vértices. É importante salientar aqui, porém, que o custo das arestas dos vértices "pai" podem ser desconsideradas apenas por que uma única quantidade de produto é produzida em qualquer caso, como de acordo com as particularidades dos casos de teste.

Em seguida, foi dada atenção a quais métodos seriam interessantes para as classes. Naturalmente, é necessário criar métodos que adicionem vértices ao grafo e outros que adicionem ou modifiquem as relações dos vértices. Adicionando e refinando o pseudo-código, temos o seguinte esqueleto de classe:

---

[3] "Um ciclo de um grafo 'G', também chamado de *circuito* se o primeiro vértice não for especificado, é um subconjunto do conjunto de vértices de 'G' que formam um caminho tal que o primeiro nodo corresponde ao último nodo" — Wolfram MathWorld
[4] Vale salientar que, em Python, toda lista pode funcionar como um vetor. Tanto uma lista vazia quanto um vetor são declarados usando colchetes fechados (e.g.: vetor = [])
[5] Em Python, um *dict* é uma estrutura de dados do tipo hashmap, que também pode ser chamada de dicionário. Dicionários são declarados atribuindo chaves fechadas a um nome (e.g.: dicionário = {})

```
class Graph:
  class Node:
    def __init__(string nome = “”): # Inicialização de um Nodo
      string data = nome
      dict edges = {}
      list fathers = []

    def __str__():
      Retorna uma string que contém as informações da instância, para fins de verificação

    def Adiciona_Borda(Node nodo, int peso):
      Se a instância que chamou o método não tiver nodo como chave em edges:
        Adiciona nodo como chave para o valor peso no dict edges da instância
        Adiciona a própria instância que chamou o método no final da list fathers de node

  def __init__(): # Inicialização de um Graph
    list vertices = []

  def __str__():
       Para cada vértice em vertices, chama o método __str__() para aquele vértice

  def Adiciona_Vértice(Node nodo):
    Adiciona o nodo no fim de vertices

  def Constrói_e_Adiciona_Vértice(string data):
    Cria uma nova instância de Node usando data e adiciona ela no fim de vertices
    reaproveitando o método Adiciona_Vértice()

  def Adiciona_Aresta_de_Nodo1_ao_Nodo2(Node nodo1, Node nodo2, int peso):
    Encontra a posição do nodo1 em vertices e guarda em index
    vertices[index] chama o método Adiciona_Borda() usando nodo2 e peso como parâmetros

  def Encontra_Nodo(string data):
    Procura e retorna o vértice guardado em vertices cujo nome ou dado é data
    Se nenhum vértice tiver data, retorna nada

  def Hidrogenio_até_Ouro():
    Percorre o grafo inteiro começando no vértice que contém hidrogênio como data
    e terminando no vértice que contém ouro como data, acumulando em uma variável os
    pesos de cada aresta e eventualmente retornando esse valor como resposta
```

Implementação do Algoritmo

Primeira Versão

As classes assim destrinchadas, chegamos no segundo momento do processo de elaboração do algoritmo, cujo funcionamento já está tentativamente descrito no método `Hidrogenio_até_Ouro()`, embora de forma lacônica: o detalhamento compreensivo do algoritmo, passo a passo.

```
def Hidrogenio_até_Ouro(node = None):
  Se node for nulo, significa que esta é a primeira vez que a função é chamada, então:
```

```
3          node = Encontra_Nodo("hidrogenio")
```

Já percebe-se no início a escolha por uma solução recursiva, evidenciada pela necessidade de uma condicional onde é, em essência, averiguada se a função já foi chamada antes ou não. A recursividade é interessante para este algoritmo, uma vez que é necessário percorrer o grafo de um vértice qualquer até o vértice final de ouro para achar o custo de "transformar" aquele elemento em ouro.

```
4          Se node ainda for nulo, o grafo não possui um vértice com hidrogênio, logo:
5            Retorna -1
```

"Menos um", no caso, sendo um valor inválido, comunicando para o resto do programa que o grafo que foi construído é incompleto, poupando o algoritmo de uma execução errônea.

```
6          hidro = 0
```

Uma variável **`hidro`** é declarada para conter o somatório total do custo de transformar hidrogênio em outros elementos; este será o valor final retornado após a execução completa do algoritmo.

```
7          Se node não aponta para outros vértices:
8            Retorna 1
```

Só existe uma possibilidade quando um vértice aponta para nada: o nodo de ouro. Como veremos nas próximas linhas, este custo é utilizado como um fator multiplicativo; se o valor de retorno fosse zero ao invés de um, o custo de reação de qualquer vértice que aponta para o ouro seria zero também.

```
9          Para cada edge em node.edges:
10           hidro += ( o peso de node até edge ) * Hidrogenio_até_Ouro(edge)
11         Retorna hidro
```

Eis onde o verdadeiro trabalho do algoritmo começa: partindo do hidrogênio até um vértice ou elemento qualquer apontado por ele e multiplicando o custo desta "reação" pelo custo total das reações indo deste mesmo vértice qualquer até o ouro, descobrimos o custo absoluto de percorrer o grafo inteiro do hidrogênio até o ouro, com uma ressalva: partindo daquele primeiro vértice aleatório apontado pelo hidrogênio.

Com o uso de um laço, todos os vértices apontados pelo hidrogênio e seus custos totais até o ouro — calculados através da recursão — são adicionados a **`hidro`**, desta forma obtendo o custo de produção de uma única unidade de ouro em unidades de hidrogênio. Com esta forma atual, o algoritmo já pode ser considerado funcional e talvez até adequado ao problema, porém, há trabalho ainda a ser feito, pois como denotado na Figura 1 e na Figura 2, o algoritmo começa a demorar significativamente mais tempo para processar o grafo conforme o número de vértices aumenta. Em especial, a Figura 2 revela a potencial

Figura 1

Tempo de execução em milissegundos em função do tamanho do grafo

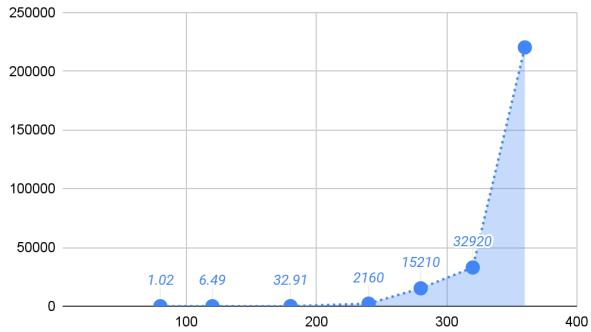
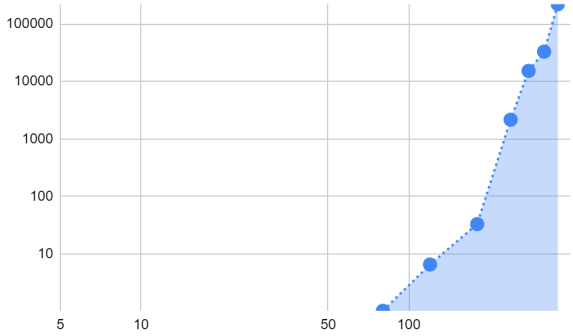


Figura 2

Tempo de execução em milissegundos em função do tamanho do grafo (escala logaritmica)



natureza exponencial do algoritmo, comportamento que deve ser evitado seja qual for o problema sendo resolvido, e como pode ser observado, o caso com uma entrada de um grafo com 400 vértices sequer foi computado, tendo o penúltimo caso de 360 vértices demorado 220.310 milissegundos, ou um pouco mais que 3 minutos e meio, valor que é estarrecedor considerando que o caso anterior de 320 vértices demorou apenas 32 segundos.

Deu-se início a uma busca para um método de otimização que acelerasse o algoritmo tal que seu tempo de execução fosse aceitável para grafos grandes, e a ideia que veio a ser utilizada foi a de reaproveitar cálculos já feitos sobre a estrutura. Se, por exemplo, um número grande de vértices apontam para um mesmo nodo, e este nodo aponta para vários outros vértices, seria interessante guardar o custo total que todo vértice que aponta para aquele nodo tem para chegar nele, desta forma poupando o algoritmo de fazer este mesmo cálculo para todos os "filhos" daquele nodo. Bastaria uma consulta a um dicionário cujas chaves são vértices e cujos valores atrelados são o custo total de percorrer o grafo até eles.

Segunda Versão

Com algumas pequenas mudanças no algoritmo e na classe **`Graph`**, a ideia do dicionário que guarda valores já calculados pode ser implementada:

```
1       class Graph:
2         ...
15        ...
16        def __init__(): # Inicialização de um Graph
17          list vertices = []
18          dict values = {}

1       def Hidrogenio_até_Ouro(node = None):
2         ...
8         ...
9         Se node não for uma chave em values:
10          values[node] = 0
11        Se values[node] for diferente de 0:
12          Retorna values[node]
13        ...
14        ...
15        values[node] = hidro
16        Retorna hidro
```

Figura 3

Tempo de execução em nanossegundos em função do tamanho do grafo

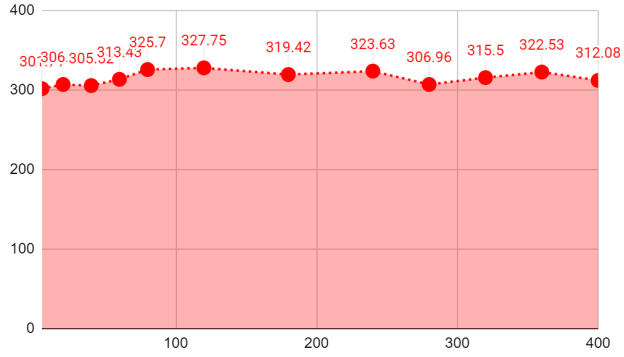
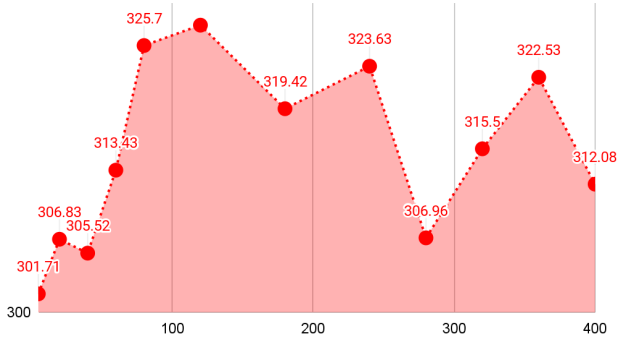


Figura 4

Tempo de execução em nanossegundos em função do tamanho do grafo (escala vertical 300 a 330 ns)



Observando a figura 3 a olho nu, pode aparentar que essas pequenas mudanças fizeram com que a complexidade de tempo do algoritmo virasse muito próxima de constante, o tamanho do grafo não sendo mais determinante para o tempo de execução. De fato, o agora-computado caso de 400 vértices sequer teve o maior tempo, como evidente na figura 4. Vale ressaltar, também, que a escala de tempo caiu em seis ordens de magnitude (10^{-6}), indo de milissegundos ($10^{-3} seg$) para nanossegundos ($10^{-9} seg$), sendo o algoritmo agora tão rápido que a captação do tempo só é possível através da média de um plural de execuções, em vista de que algumas vezes o processador simplesmente não detecta diferença de tempo e acusa zero segundos entre o início e o fim do algoritmo. Para as figuras 3 e 4, todos os casos foram executados 1 milhão de vezes.

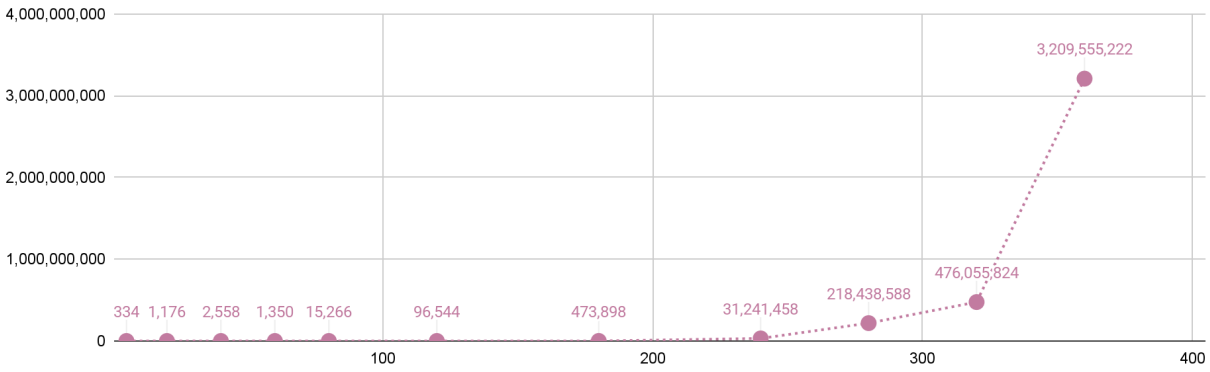
Análise

Embora seja evidente a superioridade da segunda versão, não seria certo não fazer uma análise completa de ambas as versões: utilizando uma série de equações que usam o número de chamadas do algoritmo atreladas ao tamanho das entradas, será determinada por aproximação a complexidade de tempo de ambas.

Versão 1

Figura 5.1

Número de chamadas de Hidrogenio_até_Ouro() (versão 1) em função do tamanho do grafo



$$f(5) = 334$$

$$f(360) = 3,209,555,222$$

$$r \approx \frac{log(3,209,555,222) - log(334)}{360 - 5} \approx 0.01966957...$$

$$b = 10^{r} \approx 10^{0.01966957...} \approx 1.04633215$$

Sendo b a possível base do expoente n no caso de funções exponenciais, as equações acima são utilizadas para determinar b, e sendo este valor muito próximo de 1, a probabilidade da função regente ser exponencial é muito baixa. Logo, a função é na verdade polinomial, e deve ser descoberto o grau dela, que pode ser determinado por:

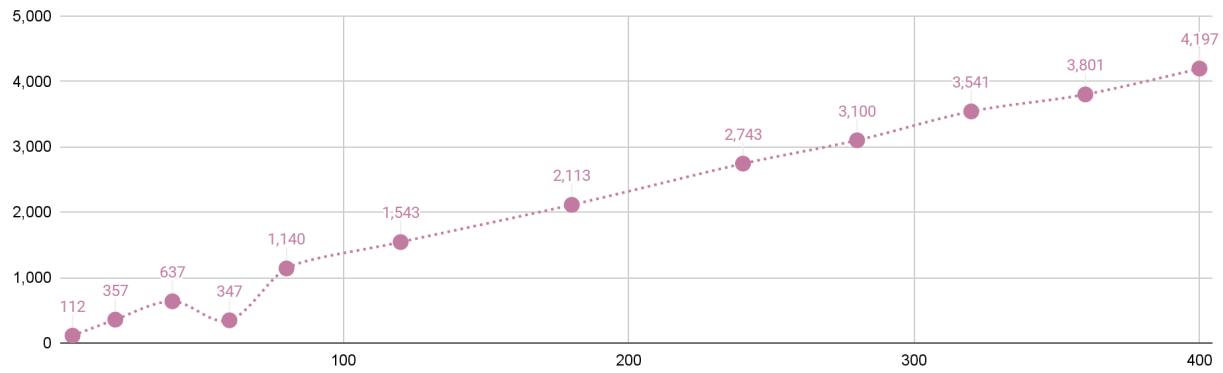
$$g \approx \frac{log(3{,}209{,}555{,}222) - log(334)}{log(360) - log(5)} \approx 3.75953061...$$

Sendo g o possível grau da função que rege o algoritmo, com um valor de aproximadamente 3.76, pode-se afirmar que a primeira versão do algoritmo tem complexidade de tempo $O(n^3)$, sendo esta classe de complexidade uma das piores. Ademais, o comportamento cúbico fica claro na figura 5.1, onde, a partir do caso de 320 vértices, o número de chamadas aumenta subitamente e, no próximo caso de 360, aumenta ainda mais drasticamente.

Versão 2

Figura 5.2

Número de chamadas do Hidrogenio_até_Ouro() (versão 2) em função do tamanho do grafo



$$f(5) = 112$$

$$f(400) = 4,197$$

$$r \approx \frac{log(4{,}197) - log(112)}{400 - 5} \approx 0.00398410...$$

$$b = 10^r \approx 10^{0.00398410...} \approx 1.00921593$$

$$g \approx \frac{log(4{,}197) - log(112)}{log(400) - log(5)} \approx 0.82692933...$$

Com um g menor que 1, e não por pouco, a probabilidade é de que a segunda versão do algoritmo tenha complexidade de tempo $O(log\ n)$, que é a segunda melhor classe de tempo, sendo a melhor $O(1)$, ou constante.

Nota-se que na figura 5.2, o número de chamadas aumenta junto do tamanho de entrada, contudo, como visto na figura 4, o tamanho do grafo não parece ser determinante para o tempo de execução. O fator decisivo para o tempo, por um *educated guess* ou palpite fundamentado, deve ser a convergência ou não de nodos para um único nodo que fica entre os vértices de hidrogênio e de ouro. Isto é, se muitos vértices apontam para um mesmo vértice, todos os nodos apontados por este vértice podem consultar em **`values`** o custo total de percorrer o grafo até aquele mesmo vértice, que é uma operação de complexidade

de tempo constante[6], ao invés de fazer todo o cálculo necessário para descobrir este valor, processo que consome muito mais tempo do que uma operação $O(1)$.

Por conseguinte, pode-se concluir que as diferenças de tempo entre os casos de teste da segunda versão se devem a certos "nodos funís" que, como consequência das suas conexões, levam a um maior número de acessos em `values` e a um decréscimo da quantidade de cálculos necessários.

Conclusão

Durante a elaboração do algoritmo, uma série de questões foram levantadas, em principal a otimização do próprio. As receitas dos alquimistas são apenas um caso onde grafos e métodos de tratá-los são interessantes; no mundo real, as aplicações de grafos são múltiplas, e quantidade de dados ou vértices sendo manipulados podem ser avassaladores.

Uma aplicação da teoria de grafos é na busca de *Minimum Delay Scheduling*[7] (MDS) ou *Agendamento com Atraso Mínimo* para aviões, onde o espaço aéreo é segmentado em zonas transitam. Como definir o menor atraso possível para todos os aviões sem alterar as agendas de forma significativa, levando em conta a capacidade de cada aeroporto e zona, como também a proporção adequada de chegadas e partidas? Dado um país grande como o Brasil ou os Estados Unidos, o grafo que comporta todas as zonas, todos os aeroportos e todos os aviões teria uma quantidade tremenda de dados, dados que precisam ser processados a todo momento para a garantia do menor atraso possível para os aviões.

Enquanto que os alquimistas podem esperar alguns minutos para descobrirem quantos átomos de hidrogênio serão gastos na busca por ouro, uma torre de controle aérea não tem tempo para perder, e o quanto antes os resultados forem emitidos por um algoritmo, mais seguro e confiável será o serviço feito pela torre. As redes de tráfego aéreo que existem hoje, entre vários outros sistemas modernos e extremamente interligados, não seriam possíveis com um algoritmo cuja performance é tão devagar quanto a primeira versão do algoritmo proposto. A teoria de grafos torna possível tratar problemas como esses; a otimização e aperfeiçoamento dos algoritmos tornam os tratamentos exequíveis.

---

[6] "Na vida real, o tempo de complexidade de um *HashMap* não é *O(1)*, sendo na verdade *Big O(tempo médio de colisão da estrutura)*, também conhecido como *O(1) amortizado*. Um verdadeiro *O(1)* só é atingido através de *hashing* sem colisões (também conhecido como *hashing* 'perfeito')." — Medium.com | Internal Working of HashMap in Java. Does it really maintain an O(1) time complexity? What is the Difference between TreeMap, HashMap, LinkedHashMap, and HashTable?

[7] "O problema de agendamento de linhas aéreas é como um todo muito complexo. O vasto número de regras e regulamentos associados com aeroportos, aviões e tripulação, combinados com a expansão global das redes de tráfego aéreo, requerem que o problema seja quebrado em peças menores e gerenciáveis para manter algum grau de tratabilidade. Consequentemente, o problema de agendamento de linhas aéreas tradicional é tipicamente decomposto em quatro estágios, com a saída de um estágio usada como entrada para os subsequente(s). O primeiríssimo estágio é conhecido como o problema de geração de agendas. Nele, uma linha aérea busca construir uma agenda de voos onde cada voo é especificado por uma 'origem, destino, data de partida, tempo, e duração.'" — Robust Airline Schedule Planning: Minimizing Propagated Delay in an Integrated Routing and Crewing Framework; Seção 1.1; Autores: Dunbar, Michelle; Froyland, Gary; Wu, Cheng-Lung.

Dados Experimentais Obtidos

Os resultados foram obtidos usando a biblioteca “time” de Python em uma nova classe **`Handler`** que, como o nome sugere, manuseia e popula os grafos, junto de uma nova função-membro de **`Graph`** chamada `assess()` que averigua a execução do algoritmo e determina se ela foi feita com êxito, com erros, ou com resultados diferentes dos que foram obtidos antes. O número de chamadas, por sua vez, foi obtido com a introdução de uma variável nova como atributo dos grafos, cujo valor é incrementado toda vez que a função `Hidrogenio_até_Ouro()` é chamada.

```
1       class Handler:
2         def __init__(self, filename: str = “exemplo.txt”) -> None:
3           self.graph = read_data(filename)
4
5         def time_delta(self) -> int:
6           Usando a biblioteca time, retorna a diferença de tempo entre antes de chamar
7           Hidrogenio_até_Ouro() e depois da execução deste
8
9         def handle(self, limit, filename) -> str:
10          Executa Hidrogenio_até_Ouro() limit vezes, retornando uma string que diz o nome do
11          caso de teste (filename) e, através do método assess() e de time_delta(), o tempo
12          médio de execução e se a execução foi correta ou não
```

Versão 1

Caso de Teste	Tempo (ms)	# Chamadas	Resultado
5 vértices	< 1	334	102,744
20 vértices	< 1	1,176	800,770
40 vértices	< 1	2,558	8,256,835
60 vértices	< 1	1,350	1,641,700
80 vértices	1.02	15,266	69,922,658,719
120 vértices	6.49	96,544	721,329,018,682,809
180 vértices	32.91	473,898	8,428,729,212,204,778
240 vértices	2,160	31,241,458	437,216,915,509,229,458,771,143,884
280 vértices	15,210	218,438,588	12,582,782,794,727,272,819,929,298,620,801
320 vértices	32,920	476,055,824	9,580,713,165,023,312,774,588,914,805,494
360 vértices	220,310	3,209,555,222	25,082,239,764,430,426,527,755,447,803,789,025
400 vértices	----------	--------------	----------

Versão 2

Caso de Teste	Tempo (ns)	# Chamadas	Resultado
5 vértices	301.71	112	102,744
20 vértices	306.83	357	800,770
40 vértices	305.52	637	8,256,835
60 vértices	313.43	347	1,641,700
80 vértices	325.7	1,140	69,922,658,719
120 vértices	327.75	1,543	721,329,018,682,809
180 vértices	319.42	2,113	8,428,729,212,204,778
240 vértices	323.63	2,743	437,216,915,509,229,458,771,143,884
280 vértices	306.96	3,100	12,582,782,794,727,272,819,929,298,620,801
320 vértices	315.5	3,541	9,580,713,165,023,312,774,588,914,805,494
360 vértices	322.53	3,801	25,082,239,764,430,426,527,755,447,803,789,025
400 vértices	312.08	4,197	63,033,055,628,032,755,906,131,628,407,108,501,716