Trabalho 2 - Algoritmo e Estrutura de Dados II

Daniel Campos Martins[*], Gabriel de Cezaro Tomaz[†]
Ciência da computação - PUCRS

11 de setembro de 2024

Resumo

Este artigo apresenta alternativas de solução para o Trabalho II de Algoritmos e Estruturas de Dados II, que trata da implementação de um algoritmo capaz de descobrir o maior caminho em um grafo. Este deve ser gerado a partir de dimensões fornecidas pelo enunciado, sendo os caminhos do grafo feitos com base em qual dimensão cabe dentro da outra.

É apresentada uma possibilidade de solução, esta é comentada e explicada e depois apresentada de forma gráfica afim de medir sua eficiência. Desse modo, é possível chegar aos resultados do problema e definir a complexidade do algoritmo.

Introdução

Este relatório descreve o desenvolvimento de um algoritmo destinado a resolver um desafio específico. O problema em questão envolve uma inspiração em um filme, onde um jovem decide presentear sua sogra de um modo bem criativo, basicamente colocar uma caixa dentro da outra, e dentro da outra, e dentro da outra... até ela achar o presente na última caixa. Para isso o jovem baixa um catálogo completo de caixas de papelão. Este catálogo possui uma lista com os tamanhos de cada caixa, dando suas três dimensões, infelizmente largura e comprimento não estão definidos e se misturam dentro do arquivo. Portanto, coube a nós implementar uma maneira automatizada de saber qual caixa cabe dentro da outra, e assim descobrir a maior sequência possível de caixas que podem ser colocadas uma dentro da outra. Foram dados catálogos de várias empresas e o algoritmo deve ser capaz de executar esta busca com todos eles.

Desse modo usamos nossos conhecimentos em grafos para resolver o problema, visando também uma boa eficiência do programa. Ao concluir a implementação, dedicamos esforços à coleta de informações sobre o desempenho do algoritmo. Esses dados são fundamentais para uma análise aprofundada da complexidade algorítmica e da eficiência do código. No decorrer do artigo, exploraremos mais essa parte.

Planejamento

Para concluir a implementação do algoritmo, estabelecemos uma série de passos organizacionais que também visavam facilitar a replicação do projeto. Esses passos incluíram:

1. Criação do objeto caixa, cada dimensão do catálogo vira um objeto caixa.

[*] martins.daniel23@edu.pucrs.br
[†] gabriel.tomaz@edu.pucrs.br

2. Desenvolvimento do algoritmo de comparação das caixas.
3. Criação do gráfico dirigido (digrafo), usando o padrão visto em aula.
4. Implementação da função de caminho maximo, encontrar o caminho máximo pra cada vertice que possuir grau de entrada 0.
5. Análises de complexidade do algoritmo.

Posteriormente, avançamos para a fase de testes, coleta e análise de dados, utilizando tabelas e gráficos para uma melhor compreensão dos resultados.

Descrição da solução

Neste tópico buscamos mostrar como implementamos o nosso programa, qual foram nossas ideias, quais estruturas usamos, e os pseudo-códigos dos algoritmos utilizados. Assim é possível visualizar o processo utilizado e quais linhas de raciocínio foram seguidas.

Classe Box

A classe Box modela uma caixa com três dimensões (lados) e permite armazenar informações sobre outras caixas que ela pode conter ou nas quais pode ser contida. Ela inclui métodos get() para acessar as informações. E métodos para adicionar uma caixa que contem ou que está contida, além de uma representação em string para fácil visualização. Segue o pseudo-código da classe:

Algorithm 1 Classe Box

Classe Box
Atributos:
  name: Identificação da caixa
  bigger_side: Tamanho do maior lado
  medium_side: Tamanho do lado médio
  small_side: Tamanho do menor lado
  contain: Lista de caixas que cabem dentro dessa caixa
  contained: Lista de caixas que esta caixa cabe dentro

Métodos:
Construtor *Box(name, bigger_side, medium_side, small_side)*
  inicializa *name*, *bigger_side*, *medium_side*, *small_side*
  inicializa *contain* como lista vazia
  inicializa *contained* como lista vazia

Métodos *gets() e toString()*

Método *add_contain(num)*
  adiciona *num* à lista *contain*

Método *add_contained(num)*
  adiciona *num* à lista *contained*

Método *contain_IsEmpty()*
  retorna verdadeiro se *contain* estiver vazio, falso caso contrário

Classe App

É na classe app onde estão as funções para ler caixas, comparar caixas e descobrir o maior caminho. Inicialmente com a função read_boxes, que é responsável por ler os dados de um arquivo de texto específico e processá-los para criar instâncias de caixas, que são então adicionadas a uma lista global chamada list_boxes. Primeiramente ela lê as linhas do arquivo, e para cada linha do arquivo, converte os 3 valores da linha para uma lista de inteiros, ordena os valores do menor para o maior, cria o objeto Box com o nome sendo a concatenação dos valores ordenados, por fim adiciona na list_boxes.

Algorithm 2 Ler Caixas

```
Função read_boxes(chosen)
    Abrir arquivo "Casos/{chosen}.txt"
    Para cada linha em arquivo
        Converter linha em lista de inteiros
        Ordenar a lista de inteiros
        Atribuir menor, médio e maior lado
        Criar nome da caixa
        Criar objeto Box com os lados e nome
        Adicionar objeto Box a list_boxes
```

A função compare_boxes é encarregada de comparar os tamanhos das caixas, e assim modificar os valores das listas contain e contained de cada objeto Box. Para isso ela lê a váriavel global list_boxes com 2 for para comparar todas com todas, se a comparação for da própria caixa com ela mesma pula a comparação, se não compara se as três dimensões cabem dentro da outra, se couber então devemos alterar contained da primeira caixa, e contain da segunda.

Algorithm 3 Comparar Caixas

```
Função compare_boxes()
    Para cada i em list_boxes
        Para cada j em list_boxes
            Se i == j, continuar
            Se lados da caixa i < lados da caixa j
                Adicionar caixa j a contained de i
                Adicionar caixa i a contain de j
```

A função longest_path inicialmente pega todos vertices com grau de entrada igual a 0, calcula o maior caminho de cada vertice e adiciona no dicionario global longest_paths com a chave sendo o nome da caixa.

Algorithm 4 Maior Caminho

```
Função longest_path(d)
    Para cada caixa em list_boxes
        Se caixa.contain_IsEmpty()
            Criar LongestPathDAG com caixa
            Encontrar longest path
            Adicionar longest path a longest_paths
```

O algoritmo da função calc_longest_path verifica se cada caixa na list_boxes está no dicionario global longest_paths, ou seja se tem um caminho máximo, se estiver path recebe o caminho. Se path não for vazio adiciona na variável global paths.

Algorithm 5 Cálculo de Maior Caminho

Função calc_longest_path()
  Para cada source_box **em** list_boxes
    Para cada target_box **em** list_boxes
      Se source_box em longest_paths
        Calcular path de source_box a target_box
        Adicionar path a paths

A função print_longest_path passa por todos os caminhos na variável global paths e testa para descobrir qual o maior e depois imprime o comprimento e o caminho em si.

Algorithm 6 Imprimir Maior Caminho

Função print_longest_path()
  Inicializar max_length e max_path_info
  Para cada (source, target, path, length) **em** paths
    Se length > max_length
      Atualizar max_length e max_path_info
  Se max_path_info não **for** None
    Imprimir max_length e max_path_info

É na função main onde tudo é executado passo a passo. Primeiro obtém a resposta do usuário de qual catálogo será usado, executa read_boxes com o catálogo escolhido, depois o compare_boxes e assim escreve o .txt do digrafo. Cria um objeto digrafo com esse .txt (A API do dígrafo é a que foi passada em aula), escreve um .dot com o mesmo, e depois executa longest_path com o mesmo. Assim com a variável global longest_paths setada, se torna possível chamar a função calc_longest_path para setar os paths, e por fim com a função print_longest_path descobrir e imprimir o maior caminho.

Algorithm 7 Principal

Função main()
  Obter opção do usuário
  Iniciar cronômetro
  Chamar read_boxes com a opção
  Chamar compare_boxes
  Chamar write_txt
  Criar Digraph
  Chamar dot com Digraph
  Chamar longest_path com Digraph
  Chamar calc_longest_path
  Chamar print_longest_path
  Parar cronômetro
  Imprimir tempo de execução

Classe Maior Caminho

A classe LongestPathDAG é projetada para encontrar o caminho mais longo em um Grafo Acíclico Direcionado (DAG). Essa classe é inicializada com um grafo e um vértice de origem, e usa uma ordenação topológica do grafo para determinar o caminho mais longo a partir do vértice de origem para todos os outros vértices do grafo. O construtor da classe inicializa as estruturas de dados

1. self.graph: Armazena o grafo passado como argumento.

2. self.s: Armazena o vértice de origem.

3. self.dist_to: Um dicionário que mapeia cada vértice para a distância mais longa conhecida a partir do vértice de origem.

Inicialmente, todas as distâncias são definidas como menos infinito (-float('inf')), exceto para o vértice de origem, que é definido como 0. self.edge_to: Um dicionário que mapeia cada vértice para o vértice anterior no caminho mais longo conhecido a partir do vértice de origem. Inicialmente, todos os valores são definidos como None.

O Método find_longest_path encontra o caminho mais longo a partir do vértice de origem para todos os outros vértices do grafo: Ele obtém a ordenação topológica do grafo usando a classe Topological (A API da Ordenação Topológica é a que foi passada em aula). Para cada vértice na ordenação topológica, se a distância mais longa conhecida a partir do vértice de origem para esse vértice não for menos infinito, ele verifica todos os vértices adjacentes. Para cada vértice adjacente, ele atualiza a distância mais longa e o vértice anterior se encontrar um caminho mais longo.

O Método path_to retorna o caminho mais longo do vértice de origem para um vértice específico v: Se a distância para v não estiver no dicionário self.dist_to ou for menos infinito, significa que v não é alcançável a partir do vértice de origem, e o método retorna None. Ele constrói o caminho mais longo começando em v e seguindo os vértices anteriores no dicionário self.edge_to até chegar ao vértice de origem. Se, durante a construção do caminho, encontrar um vértice None antes de alcançar o vértice de origem, isso indica que v não é alcançável, e o método retorna None. O caminho é construído na ordem inversa (do destino para a origem) e depois é invertido para a ordem correta (da origem para o destino). Segue o pseudo-código da classe:

Algorithm 8 Classe LongestPathDAG

```
Classe LongestPathDAG:
  Método __init__(graph, s):
    Entrada: graph (grafo), s (vértice de origem)
    self.graph ← graph
    self.s ← s
    self.dist_to ← {v : −∞ para cada v ∈ graph.getVerts()}
    self.edge_to ← {v : None para cada v ∈ graph.getVerts()}
    self.dist_to[s] ← 0
  Método find_longest_path():
    topo_sort ← Topological(self.graph).getTopological()
    Para cada v em topo_sort faça:
      Se self.dist_to[v] ≠ −∞ então:
        Para cada w em self.graph.getAdj(v) faça:
          Se w não está em self.dist_to então:
            self.dist_to[w] ← −∞
          Se self.dist_to[w] < self.dist_to[v] +1 então:
            self.dist_to[w] ← self.dist_to[v] +1
            self.edge_to[w] ← v
  Método path_to(v):
    Entrada: v (vértice de destino)
    Se v não está em self.dist_to ou self.dist_to[v] == −∞ então:
      Retornar None
    path ← []
    Enquanto v ≠ None e v ≠ self.s faça:
      path.insert(0, v)
      v ← self.edge_to[v]
    Se v == None então:
      Retornar None
    path.insert(0, self.s)
    Retornar path
```

Análise do algoritmo

Após a conclusão do algorítmo iniciamos enfim a fase de testes. Os testes foram todos realizados na mesma máquina afim de garantir a melhor veracidade dos dados coletados. Com o objetivo de descobrir a complexidade do algoritmo desenvolvido coletamos o tempo de execução e a contagem de operações. Foi desenvolvido representações do algoritmo através de tabelas e gráficos, como apresentado na Figura 1, com a intenção de apresentar melhor visualização dos dados da tabela.

Configuração da máquina usada para os testes:

- Ryzen 5 4600g
- Vega 8
- 16,00 GB RAM
- Windows 10 64 bits

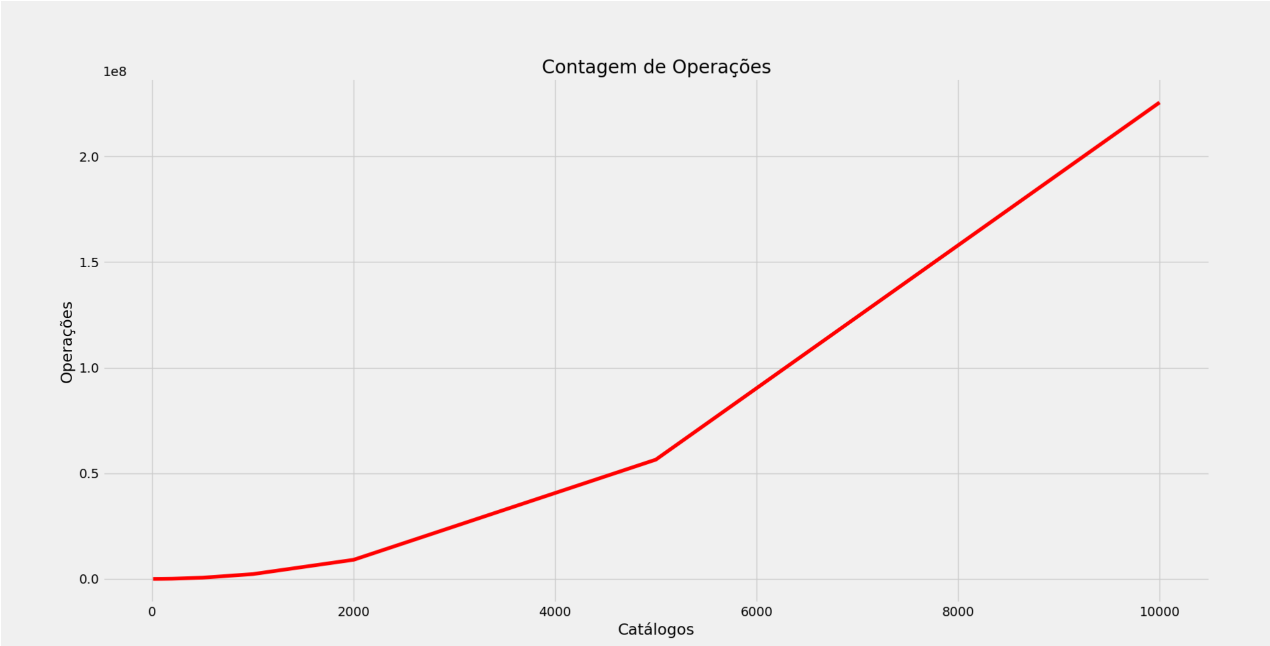

Figura 1: Contagem de Operações

De acordo com a análise do algoritmo, utilizando a fórmula da função polinomial descobrimos que o expoente aproximado é de 2.09, o que indica que sua complexidade é de $\mathcal{O}(n^2)$. Na Figura 2 a curva verde representa $n^{2.09}$ e a curva vermelha representa a contagem de operações do algoritmo, as duas em relação aos catálogos.

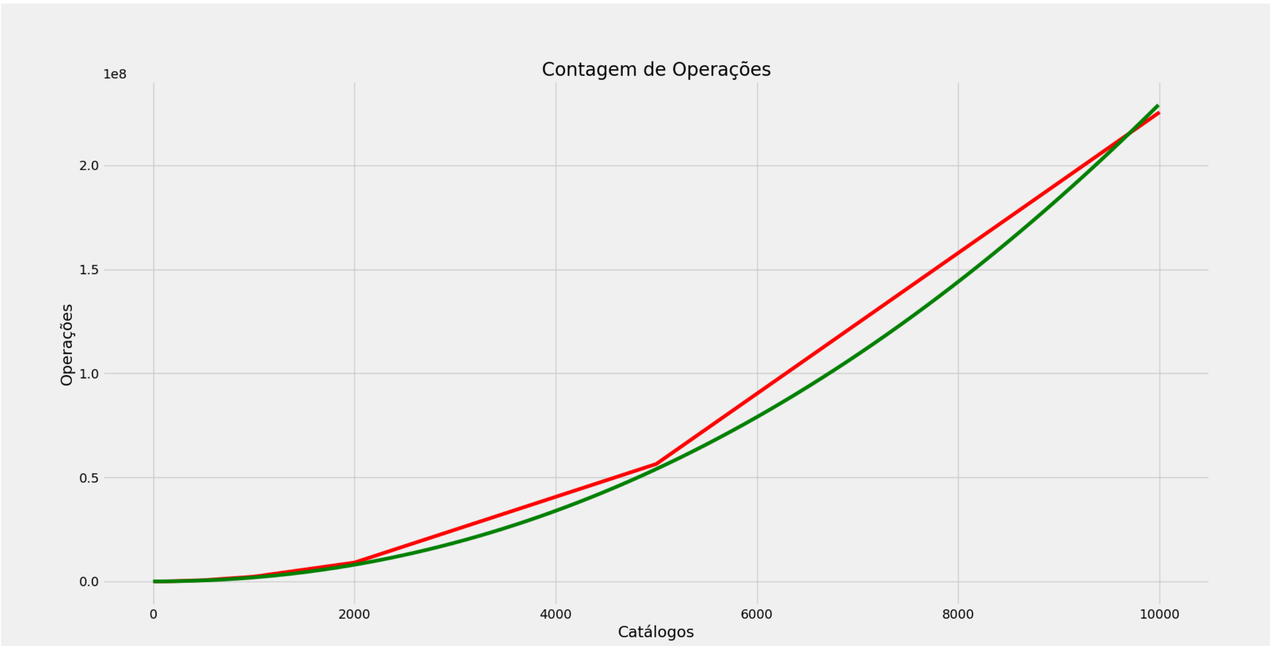

Figura 2: Algoritmo - Complexidade

Resultados

Após escolher o catálogo é possível chegar ao resultado desejado, assim descobrindo o maior caminho e as vértices que contemplam este caminho. Para uma melhor visualização montamos a Tabela 1 e a Tabela 2, que apresenta os dados coletados.

Catálogo	Caminho Máximo	Contagem de Operações	Tempo (s)
teste10	4	284	0.0
teste20	5	1.043	0.0
teste50	11	5.980	0.0
teste100	13	23.628	0.0
teste200	17	92.633	0.065
teste500	27	566.504	0.203
teste1000	32	2.265.431	1.156
test2000	43	9.042.733	5.109

Tabela 1: Resultado obtido do programa

Catálogo	Caminho Máximo	Contagem de Operações	Tempo (s)
teste5000	60	56.459.551	38.203
teste10000	75	225.547.121	207.078

Tabela 2: Resultado obtido do programa - Casos a mais (Não Obrigatórios)

Conclusão

Após compreendermos a estrutura do grafo gerado, bem como ele dever ser percorrido, o desenvolvimento do programa se tornou consideravelmente mais simples. Como resultado, conseguimos alcançar nosso objetivo: descobrir a maior quantidade de caixas que cabem uma dentro da outra, ou seja o maior caminho no grafo das caixas, apenas com as dimensões fornecidas pelos catálogos. Além disso, acreditamos ter desenvolvido uma boa solução para o problema, utilizando um algorimto eficiente. No entanto, como uma melhoria futura, gostaríamos de deixar o algoritmo ainda mais eficiente, pois acreditamos que haja um jeito para diminuir sua complexidade.