Análise e Implementação de um Algoritmo para Determinação da Maior Sequência de Caixas Encaixáveis

Thomaz Abrantes de Oliveira Martinelli Silva*
Escola Politécnica — PUCRS

30 de junho de 2024

Resumo

Neste trabalho, foi abordado o problema de encontrar a maior sequência de caixas encaixáveis, onde cada caixa pode ser inserida em outra se todas as suas dimensões forem estritamente menores. Foi utilizado um grafo dirigido para modelar as relações de encaixe entre as caixas, onde cada vértice representa uma caixa e uma aresta indica que uma caixa pode ser inserida em outra. A ordenação inicial das caixas e a construção do grafo permitem identificar as relações de encaixe eficientemente. Em seguida, foi aplicado o algoritmo de busca em profundidade (DFS) para determinar o maior caminho no grafo, que corresponde à maior sequência de caixas encaixáveis. Os resultados mostram que a abordagem é eficaz para conjuntos grandes de caixas, fornecendo uma solução clara e detalhada para o problema.

Palavras-chave

Algoritmo, Grafos dirigidos, Caminhamento DFS, Caixas, Maior Sequência de Caixas Encaixáveis.

Introdução

Dentro do escopo da disciplina de Algoritmos e Estruturas de Dados 2 da Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), foi concebida a seguinte problemática: dada uma determinada quantidade de caixas, tendo suas três dimensões dispostas em arquivos de texto, foi solicitado aos estudantes que desenvolvessem uma solução capaz de determinar a maior sequência de caixas encaixáveis, ou seja, a maior quantidade de caixas que podem ser postas uma dentro da outra. Assim, os alunos deveriam utilizar seus conhecimentos acerca da estrutura de dados "Grafo", fortemente estudada em aula durante o período de um semestre, para desenvolver um algoritmo capaz de encontrar uma solução eficaz para solucionar a problemática.

Com o intuito de auxiliar os alunos em sua realização do trabalho, cada professor disponibilizou uma sequência de arquivos de texto, simulando um catálogo contendo todos os tamanhos de caixas fabricadas, sendo fornecidos os casos de teste com: 10, 20, 50, 100, 200, 300, 500, 1000, 2000, 5000 e 10000 caixas. Os arquivos dispunham de três números inteiros separados por vírgulas, onde cada linha representava uma caixa e cada um dos números representava uma dimensão. Entretanto, altura, largura e comprimento não estavam identificados, se misturando dentro do arquivo. Abaixo está um exemplo de um modelo de arquivo de texto contendo um dos casos de teste cedidos aos estudantes:

*thomaz.abrantes@edu.pucrs.br

```
991 443 126
733 281 710
910 720 218
743 512 162
988 955 720
680 603 649
336 326 615
566 764 487
680 579 148
629 222 697
```

Figura 1: Caso de teste contendo as dimensões de dez caixas.

Para facilitar a compreensão e visualização do problema, foi apresentada a seguir uma representação gráfica de um simulador de caixas encaixadas. O esquema mostra uma caixa externa de cor verde, com dimensões de **281 x 710 x 733**, contendo uma caixa interna de cor rosa, com dimensões de **222 x 629 x 697**.

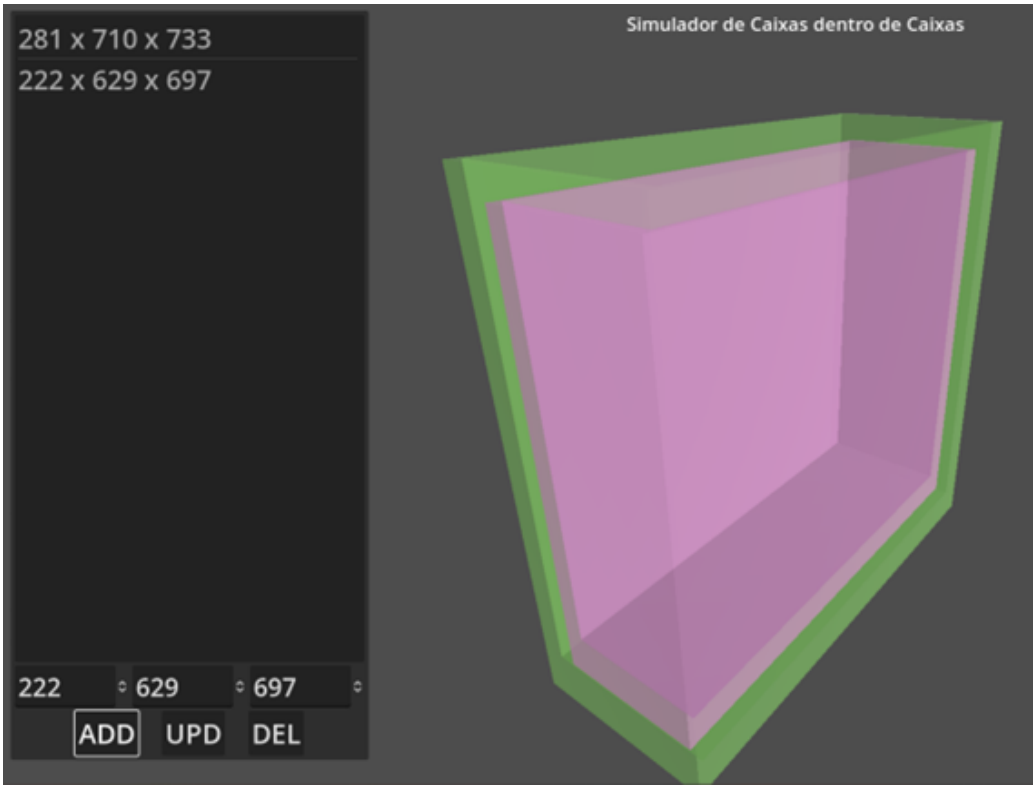

Figura 2: Simulador de caixas dentro de caixas desenvolvido pelo Professor Marcelo Cohen.

O fato de se ter que lidar com caixas de diversos tamanhos, tornou o problema de encontrar a maior sequência de caixas encaixáveis um tanto quanto desafiador. Para resolver esse problema de forma eficiente e ao mesmo tempo utilizar estruturas de dados estudadas em aula, foi decidido utilizar grafos dirigidos e caminhamento DFS (DepthFirstSearch).

Por que utilizar grafos dirigidos?

Um grafo dirigido é uma estrutura de dados composta por vértices e arestas, onde cada aresta conta com uma direção associada, unindo um vértice de **origem** e um outro vértice de **destino**. No contexto do problema, os vértices do grafo representam as caixas, enquanto as arestas dirigidas representam a possibilidade de uma caixa caber dentro de outra. Além disso, os grafos dirigidos permitem a análise de caminhos entre os vértices,

Por que utilizar caminhamento DFS?

O caminhamento DFS é um algoritmo que visita todos os vértices de um grafo, se movimentando pelos arcos de um vértice a outro, numerando-os de acordo com a ordem em que são visitados. Isso garante a exploração de todos os possíveis caminhos a partir de uma caixa, garantindo que todas as sequências possíveis sejam consideradas.

Arquitetura da apresentação da solução

De modo a resolver o problema proposto, será apresentada uma possível alternativa de solução, bem como suas características, contando com códigos, juntamente com suas devidas explicações. Em seguida os resultados obtidos em cada um dos casos testes serão apresentados além de uma análise de complexidade, bem como as conclusões obtidas no decorrer do trabalho.

Metodologia e Solução

Após analisar o problema, o primeiro passo para conseguir tentar chegar a uma solução eficiente foi pensar em como o algoritmo seria elaborado. Assim, optou-se por desenvolver codificações em Java, aos moldes da programação orientada a objetos (POO), contando com determinadas classes, sendo elas: "App", "Caixa", "DepthFirstSearch"e "Digraph"(essa última contando com as classes auxiliares: "In", "Out", "StdIn"e "StdOut"). Todas elas serão devidamente mais detalhadas ao decorrer desta seção.

Classe Caixa

Tendo a idealização de como seria implementada a solução, seguiu-se para o passo seguinte, que nesse caso foi tentar encontrar uma lógica para realizar a verificação se uma caixa poderia ser encaixada em outra. Partindo desse cenário, criou-se a classe "Caixa", que é essencial para representar cada caixa - cada vértice do grafo - individualmente. A seguir será explicada cada parte da classe.

Atributos

int[] dimensoes: Um arranjo de inteiros que armazena as dimensões da caixa.

Construtor

```
1        public Caixa(int[] dimensoes) {
2            this.dimensoes = dimensoes;
3            Arrays.sort(this.dimensoes);
4        }
```

Inicializa uma instância da classe Caixa com as dimensões fornecidas e as ordena em ordem crescente. Nesse caso, ordenar as dimensões facilita a comparação das caixas, pois garante que a menor dimensão esteja no início do arranjo e a maior no final, uma vez que as dimesões não estão dispostas de maneira ordenada nos arquivos de texto dos casos de teste.

Métodos

cabeDentro:

```
    public boolean cabeDentro(Caixa outraCaixa) {
        Arrays.sort(this.dimensoes);
        Arrays.sort(outraCaixa.dimensoes);
        for (int i = 0; i < dimensoes.length; i++) {
            if (dimensoes[i] >= outraCaixa.dimensoes[i]) {
                return false;
            }
        }
        return true;
    }
```

Função: Verifica se a caixa atual cabe dentro de uma outra caixa.
Detalhamento:

1. Ordena as dimensões das duas caixas para garantir a ordem crescente.
2. Compara cada dimensão da caixa atual com a dimensão correspondente da outra caixa.
3. Se em qualquer comparação, a dimensão da caixa atual for maior ou igual à dimensão da outra caixa, a caixa atual não pode caber dentro da outra e o método retorna false.
4. Se todas as comparações passarem, a caixa atual cabe dentro da outra e o método retorna true.

compareTo:

```
    public int compareTo(Caixa outraCaixa) {
    for (int i = 0; i < this.dimensoes.length; i++) {
        if (this.dimensoes[i] != outraCaixa.dimensoes[i]) {
            return this.dimensoes[i] - outraCaixa.dimensoes[i];
        }
    }
    return 0;
}
```

Função: Comparação entre duas caixas com base em suas dimensões.
Detalhamento:

1. Compara as dimensões da caixa atual com as dimensões da outraCaixa, uma a uma.
2. Retorna a diferença entre as dimensões na primeira posição onde elas diferem.
3. Caso todas as dimensões sejam iguais, retorna 0.

toString:

```
    public String toString() {
        return Arrays.toString(dimensoes);
    }
```

Função: Fornece uma representação em string das dimensões da caixa.
Detalhamento:

1. Converte o arranjo de dimensões em uma string e a retorna.
2. Facilita a visualização das dimensões da caixa ao imprimir objetos da classe Caixa.

Classe Digraph

A classe "Digraph", como o próprio nome indica, trata-se da representação de um grafo dirigido. Ela tem o intuito de criar, manipular e visualizar grafos dirigidos. Ela inclui funcionalidades para adicionar vértices e arestas, verificar conexões, e representar o grafo tanto em formato de string quanto em formato DOT para visualização gráfica.

É importante destacar que essa classe "Digraph"esteve disponível para os estudantes da disciplina de Algoritmos e Estruturas de Dados 2 desde o início do estudo acerca de grafos e por isso, não foi preciso implementá-la do zero. Dessa forma, de modo a facilitar a realização deste trabalho, a classe "Digraph"foi usada, juntamente de suas classes auxiliares: "In", "Out", "StdIn"e "StdOut", da forma em que foi cedida aos estudantes, com pequenas mudanças, que consistem no apagamento de comentários no código e traduções do inglês para o português. A classe será explicada abaixo.

Atributos

1. int V: Número de vértices no grafo.
2. int E: Número de arestas no grafo.
3. List<Integer>[] adj: Lista de adjacências para cada vértice.
4. int[] indegree: Arranjo que armazena o grau de entrada de cada vértice, ou seja, o número de arestas que chegam a cada vértice.

Construtor

```
1    public Digraph(int V) {
2        if (V < 0)
3            throw new IllegalArgumentException("Numero de vertices em um grafo
                 nao pode ser negativo");
4        this.V = V;
5        this.E = 0;
6        indegree = new int[V];
7        adj = (List<Integer>[]) new List[V];
8        for (int v = 0; v < V; v++) {
9            adj[v] = new LinkedList<>();
10       }
11   }
```

O construtor da classe recebe o número de vértices como parâmetro e lança uma exceção caso o mesmo seja negativo negativo. Além disso, ele também faz todas as inicializações:

1. this.V: Define o número de vértices.
2. this.E: Inicializa o número de arestas como 0.
3. indegree: Inicializa um array para armazenar o grau de entrada de cada vértice.
4. adj: Inicializa uma lista de adjacências para cada vértice, onde cada uma irá armazenae os vértices adjacentes.

Métodos

addEdge:

```
    public void addEdge(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        adj[v].add(w);
        indegree[w]++;
        E++;
    }
```

O método recebe como parâmetros v e w (vértices entre os quais a aresta será adicionada) e verifica se os mesmos são válidos. Além disso, ele faz as seguintes operações:

1. Adiciona w à lista de adjacências do vértice v.
2. Incrementa o grau de entrada do vértice w.
3. Incrementa o número de arestas 'E' do grafo.

adj:

```
    public Iterable<Integer> adj(int v) {
        validateVertex(v);
        return adj[v];
    }
```

O método recebe um vértice cujo adjacente se deseja obter, faz a verificação se o mesmo é válido e retorna a lista de vértices adjacentes ao vértice v.

toString:

```
    public String toString() {
        System.out.println("Estrutura do Grafo Dirigido: \n");
        StringBuilder s = new StringBuilder();
        s.append(V + " vertices, " + E + " arestas " + NEWLINE);
        for (int v = 0; v < V; v++) {
            s.append(String.format("%d: ", v));
            for (int w : adj[v]) {
                s.append(String.format("%d ", w));
            }
            s.append(NEWLINE);
        }
        return s.toString();
    }
```

O método retorna uma representação em string do grafo, útil para fins de depuração e visualização. Ele mostra:

1. o número de vértices e arestas
2. cada vértice seguido de seus vértices adjacentes.

toDot:

```java
    public String toDot() {
        System.out.println("Construcao do Grafo Dirigido: \n");
        StringBuilder s = new StringBuilder();
        s.append("digraph {" + NEWLINE);
        s.append("rankdir = LR;" + NEWLINE);
        s.append("node [shape = circle];" + NEWLINE);
        for (int v = 0; v < V; v++) {
            for (int w : adj[v]) {
                s.append(v + " -> " + w + ";" + NEWLINE);
            }
        }
        s.append("}");
        return s.toString();
    }
```

O método retorna uma representação em string do grafo no formato DOT, que pode ser usado para visualizar o grafo usando ferramentas como Graphviz.

Exemplo: construção de um grafo usando o conteúdo do método **toDot** com as caixas contidas no primeiro arquivo de texto que contém o primeiro caso de teste, dez caixas, no Graphviz:

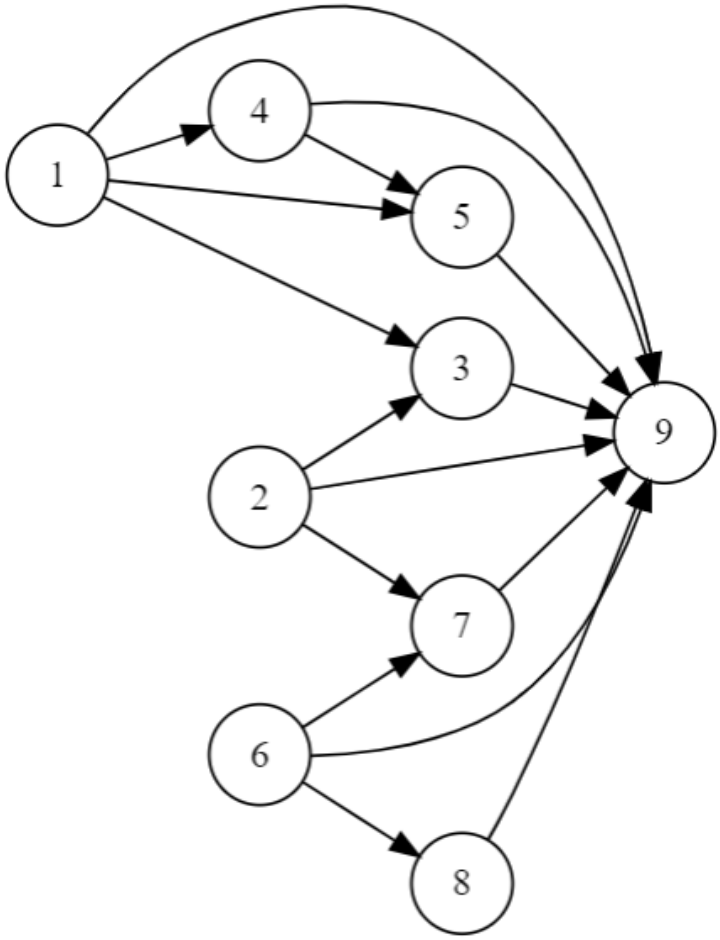

Figura 3: Demonstração do uso da saída do método toDot para construir um grafo com dez vértices e dezesseis arestas no Graphviz Online.

Classe DepthFirstSearch

A classe "DepthFirstSearch"representa a execução da busca em profundidade no grafo dirigido. Seu objetivo principal é encontrar o maior caminho dentro do grafo, aquele que conta com o maior número de arestas. Na busca em profundidade se visita um vértice e, a partir dele, recursivamente visita-se todos os seus vértices adjacentes não visitados.

Novamente, assim como a classe "Digraph", a classe "DepthFirstSearch"também esteve disponível para os estudantes da disciplina de Algoritmos e Estruturas de Dados 2 desde o início do estudo de grafos. Dessa forma, o que foi feito com a classe foi realizar algumas mudanças, de modo similar ao que foi feito com a classe "Digraph". Uma das principais mudanças foi a criação do método "encontrarMaiorCaminho". Abaixo está estruturada a classe "DepthFirstSearch".

Atributos

1. boolean[] marked: Um arranjo que indica se um vértice foi visitado durante a execução da busca em profundidade.
2. int[] edgeTo: Um arranjo que mantém o rastreamento dos caminhos no grafo.
3. int[] distTo: Um arranjo que armazena o comprimento do maior caminho a partir do vértice inicial até cada vértice.
4. ArrayList<Integer>[] longestPaths: Um arranjo de listas que armazena os caminhos mais longos encontrados até agora.

Construtor

```
1     public DepthFirstSearch(Digraph g) {
2         marked = new boolean[g.getV()];
3         edgeTo = new int[g.getV()];
4         distTo = new int[g.getV()];
5         longestPaths = new ArrayList[g.getV()];
6         for (int v = 0; v < g.getV(); v++) {
7             distTo[v] = -1;
8             longestPaths[v] = new ArrayList<>();
9         }
10    }
```

O construtor da classe "DepthFirstSearch"tem a função de inicializar todos os atributos. O laço **for** inicializa distTo para -1, indicando que as distâncias ainda não foram calculadas, e cria novas listas para longestPaths.

Métodos

dfs:

```
1     private int dfs(Digraph g, int v) {
2         if (distTo[v] != -1) return distTo[v];
3         marked[v] = true;
4         int maxLength = 0;
5         for (int w : g.adj(v)) {
6             int length = dfs(g, w) + 1;
7             if (length > maxLength) {maxLength = length; edgeTo[v] = w;}
8         }
9         distTo[v] = maxLength;
10        return maxLength;
11    }
```

Função: Realizar a busca em profundidade para calcular a distância máxima de um vértice v até qualquer outro vértice acessível a partir dele.

Detalhamento:

1. Faz a verificação para ver se distTo[v] não for -1, pois isso significaria que a distância máxima já foi calculada, então nesse caso retornaria esse valor.
2. Define marked[v] como true, indicando que o vértice v foi visitado.
3. Declara a variável maxLength para guardar o comprimento do caminho mais longo a partir do vértice v.
4. Para cada vértice adjacente w, chama de forma recursiva dfs(g, w) e calcula o comprimento do caminho.
5. Se o caminho encontrado é mais longo, atualiza maxLength e registra w como o vértice que maximiza a distância.
6. Define distTo[v] como maxLength e retorna maxLength.

encontrarMaiorCaminho

```
1     public ArrayList<Integer> encontrarMaiorCaminho(Digraph g) {
2         int maxLength = 0;
3         int endVertex = 0;
4
5         for (int v = 0; v < g.getV(); v++) {
6             int length = dfs(g, v);
7             if (length > maxLength) {
8                 maxLength = length;
9                 endVertex = v;
10            }
11        }
12
13        ArrayList<Integer> path = new ArrayList<>();
14        for (int x = endVertex; distTo[x] != 0; x = edgeTo[x]) {
15            path.add(x);
16        }
17        path.add(0);
18        Collections.reverse(path);
19
20        return path;
21    }
```

Função: Identificar e retornar a sequência de vértices que compõem o maior caminho em todo o grafo. No contexto do trabalho, retornar a maior quantidade de caixas encaixáveis.

Detalhamento:

1. Inicializa maxLength e endVertex: Para rastrear o comprimento do caminho mais longo encontrado até agora e o vértice onde esse caminho termina.
2. Itera Sobre Todos os Vértices: Para cada vértice v no grafo, chama dfs(g, v) e atualiza maxLength e endVertex se encontrar um caminho mais longo.

3. Reconstrói o Caminho: A partir de endVertex usa edgeTo e distTo para reconstruir o caminho mais longo.

4. Inverte o Caminho: Usa Collections.reverse(path) para obter a ordem correta e retorna o caminho como uma lista de vértices.

A aplicação

A classe "App"é a responsável pela interação com o usuário. Ela permite que o usuário escolha o arquivo de texto com o teste correspondente ao número de caixas que deseja analisar, através de um menu. Além disso, ela também é a responsável pela leitura do arquivo texto, onde cada uma de suas linhas representa uma caixa com suas dimensões, usando a biblioteca "BufferedReader"inerente à linguagem Java. Ela também cria uma instância da classe "Digraph"com o número de vértices igual ao número de caixas, ou seja, cada caixa é representada por um vértice no grafo, como pode ser visto a seguir.

```
    Digraph digraph = new Digraph(caixas.size());
```

Depois disso, percorre todas as caixas para adicionar arestas ao grafo. A ideia é verificar a partir de um par de caixas 'i' e 'j' se 'i' pode ser encaixada a 'j'. Isso é feito através do método "cabeDentro"da classe "Caixa". Se essa condição for verdadeira, uma aresta é adicionada do vértice 'i' ao vértice 'j' no grafo, usando o método "addEdge"da classe "Digraph". Isso pode ser observado abaixo.

```
    for (int i = 0; i < caixas.size(); i++) {
        for (int j = 0; j < caixas.size(); j++) {
            if (i != j && caixas.get(i).cabeDentro(caixas.get(j))) {
                digraph.addEdge(i, j);
            }
        }
    }
```

Ademais, o algoritmo DFS é executado para encontrar a maior sequência de caixas encaixáveis, através do chamamento do método "encontrarMaiorCaminho"da classe "DepthSearchFirst", como é possível ser visto a seguir.

```
    DepthFirstSearch dfs = new DepthFirstSearch(digraph);
    ArrayList<Integer> maiorCaminho = dfs.encontrarMaiorCaminho(digraph);
```

Por fim, a aplicação calcula e mostra o tempo de execução do programa em milissegundos, exibe a estrutura do grafo e imprime a maior sequência de caixas encaixáveis.

```
    long tempoInicial = System.nanoTime();
    ...
    long tempoFinal = System.nanoTime();
    long duracao = (tempoFinal - tempoInicial) / 1_000_000;
    System.out.println("Tempo de execucao: " + duracao + " milissegundos");

    System.out.println("Estrutura simples do grafo dirigido: \n");
    int numeroVertices = digraph.getV();
    System.out.printf("Numero de vertices eh : %d \n", numeroVertices);
    int numeroArestas = digraph.getE();
    System.out.printf("Numero de arestas eh  : %d \n", numeroArestas);

    System.out.println("A maior sequencia de caixas que cabem uma na outra: " +
        maiorCaminho.size() + " \n");
        System.out.println("Dimensoes caixas: \n");
        for (int v : maiorCaminho) {
            System.out.println(caixas.get(v));
        }
```

Abaixo está um roteiro resumido da execução sequencial do programa:

1. Exibe o menu e lê a escolha do usuário.
2. Carrega os dados das caixas a partir do arquivo selecionado.
3. Cria e ordena as instâncias de caixas.
4. Constrói o grafo dirigido representando as relações de encaixe.
5. Executa o DFS para encontrar a maior sequência de caixas encaixáveis.
6. Exibe os resultados e o tempo de execução.

Resultados

Após a implementação do algoritmo explicado anteriormente em linguagem Java e execução em um tempo aproximado de um segundo, foi possível determinar a maior sequência de caixas encaixáveis, realizar dez simulações para encontrar o tempo médio de execução de cada caso teste. Depois disso, realizou-se o estudo acerca da complexidade do algoritmo em questão. Sendo esse último tópico algo extremamente importante no contexto tanto da disciplina de Algoritmos e Estruturas de Dados 2, quanto em sua antecessora. Sendo assim, a seguir estão representados cada um desses tópicos.

Maiores sequências de caixas encaixáveis

teste10.txt

Com dez caixas, obteve-se:

- Número de vértices é : 10.
- Número de arestas é : 16.
- Maior sequência de caixas encaixáveis é : 4.
- Tempo médio de execução em milissegundos após dez simulações : 788,0000 ms.

teste20.txt

Com vinte caixas, obteve-se:

- Número de vértices é : 20.
- Número de arestas é : 90.
- Maior sequência de caixas encaixáveis é : 5.
- Tempo médio de execução em milissegundos após dez simulações : 897,7778 ms.

teste50.txt

Com cinquenta caixas, obteve-se:

- Número de vértices é : 50.
- Número de arestas é : 627.
- Maior sequência de caixas encaixáveis é : 11.
- Tempo médio de execução em milissegundos após dez simulações : 974,5556 ms.

teste100.txt

Com cem caixas, obteve-se:

- Número de vértices é : 100.
- Número de arestas é : 2666.
- Maior sequência de caixas encaixáveis é : 13.
- Tempo médio de execução em milissegundos após dez simulações : 1017,1111 ms.

teste200.txt

Com duzentas caixas, obteve-se:

- Número de vértices é : 200.
- Número de arestas é : 9998.
- Maior sequência de caixas encaixáveis é : 17.
- Tempo médio de execução em milissegundos após dez simulações : 1049,8889 ms.

teste300.txt

Com trezentas caixas, obteve-se um resultado interessante:

- Número de vértices é : 100.
- Número de arestas é : 2666.
- Maior sequência de caixas encaixáveis é : 13.
- Tempo médio de execução em milissegundos após dez simulações : 1071,3333 ms.

Após analisar o conteúdo presente no arquivo de texto correspondente ao teste com trezentas caixas, foi possível concluir que o motivo do número de vértices, número de arestas e a maior sequência de caixas encaixáveis ser exatamente igual ao caso teste com cem caixas deve-se ao fato de que durante a geração dos casos teste, pode ter ocorrido um descuido, onde o conteúdo presente no **teste100.txt** foi copiado no **teste300.txt**, provavelmente por engano, ou para verificar se os alunos iriam conseguir reparar.

teste500.txt

Com quinhentas caixas, obteve-se:

- Número de vértices é : 500.
- Número de arestas é : 62912.
- Maior sequência de caixas encaixáveis é : 27.
- Tempo médio de execução em milissegundos após dez simulações : 1150,0000 ms.

teste1000.txt

Com mil caixas, obteve-se:

- Número de vértices é : 1000.
- Número de arestas é : 249135.
- Maior sequência de caixas encaixáveis é : 32.
- Tempo médio de execução em milissegundos após dez simulações : 1197,4444 ms.

teste2000.txt

Com duas mil caixas, obteve-se:

- Número de vértices é : 2000.
- Número de arestas é : 1003846.
- Maior sequência de caixas encaixáveis é : 43.
- Tempo médio de execução em milissegundos após dez simulações : 1326,1111 ms.

teste5000.txt

Com cinco mil caixas, obteve-se:

- Número de vértices é : 5000.
- Número de arestas é : 6345313.
- Maior sequência de caixas encaixáveis é : 60.
- Tempo médio de execução em milissegundos após dez simulações : 1916,1111 ms.

teste10000.txt

Com dez mil caixas, obteve-se:

- Número de vértices é : 10000.
- Número de arestas é : 25257593.
- Maior sequência de caixas encaixáveis é : 75.
- Tempo médio de execução em milissegundos após dez simulações : 3297,1111 ms.

Simulações

De modo a estimar um tempo médio de execução do algoritmo com cada caso teste em milissegundos, decidiu-se realizar dez simulações (S) para cada um deles. Em um tempo estimado de um segundo para selecionar cada caso teste no menu, foi possível obter os seguintes resultados:

Casos teste	S.1	S.2	S.3	S.4	S.5	S.6	S.7	S.8	S.9	S.10	Média
teste10.txt	893	765	841	785	772	819	700	784	733	706	788,0000
teste20.txt	851	886	1063	877	787	969	880	834	933	982	897,7778
teste50.txt	986	869	993	802	892	1191	989	893	1156	972	974,5556
teste100.txt	998	956	1019	992	934	1065	1111	938	1141	1122	1017,1111
teste200.txt	882	975	1090	1174	1301	1100	985	935	1007	1387	1049,8889
teste300.txt	1142	1101	1073	1210	944	1102	956	1017	1097	1245	1071,3333
teste500.txt	1154	1026	1051	1216	1193	1090	1219	1277	1124	1359	1150,0000
teste1000.txt	1027	1094	1168	1470	1254	1056	1418	1239	1051	1102	1197,4444
teste2000.txt	1418	1307	1235	1369	1300	1213	1268	1684	1141	1180	1326,1111
teste5000.txt	1993	2041	1881	1860	1926	1686	1824	1907	2127	1862	1916,1111
teste10000.txt	3502	3280	3531	3351	3330	3458	3550	2738	2934	3165	3297,1111

Abaixo é possível verificar um gráfico com o tempo médio de execução contendo cada um dos casos de teste.

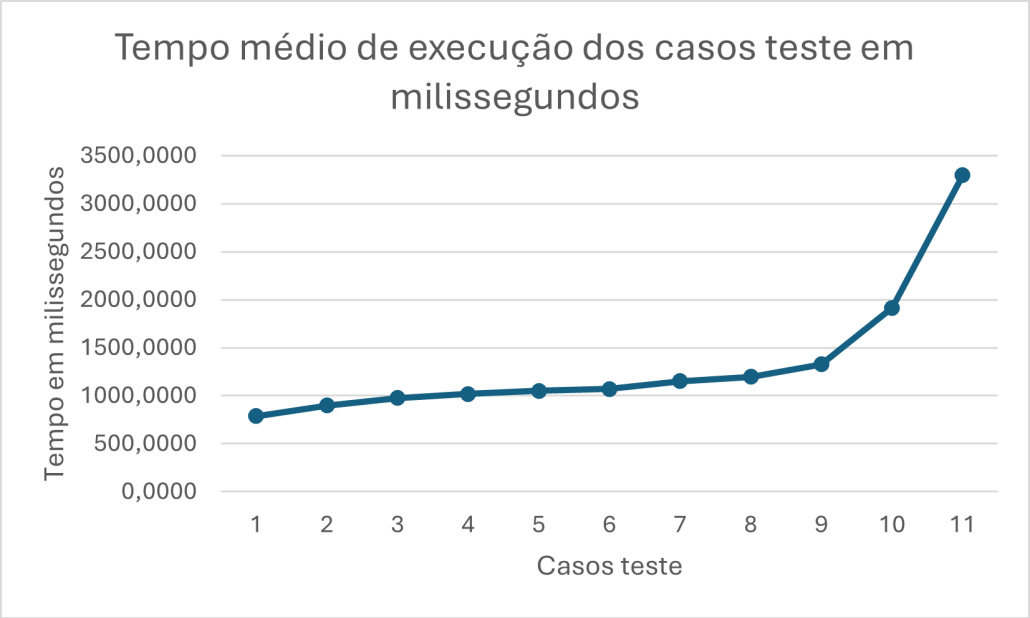

Figura 4: Gráfico do tempo médio de cada caso teste com base nas simulações, feito no Excel.

Análise de complexidade

A complexidade do algoritmo pode ser analisada em duas partes principais: a construção do grafo e a busca do maior caminho usando DFS.

1. **Construção do grafo**

   - Ordenação de caixas: Para ordenar as caixas, foi utilizado o comando "Collections.sort", que possui complexidade $O(n\ log\ n)$, onde 'n' é o número de caixas.

- Adição de arestas: Para adicionar arestas de um vértice a outro do grafo, foi verificado cada par de caixas se uma poderia ser encaixada dentro da outra, utilizando dois laços. Dessa forma, apresenta complexidade $O(n^2 * d)$ no pior caso, onde 'd' representa o número de dimensões, nesse caso sendo uma constante, 3. Ou seja, é possível considerar a complexidade como sendo $O(n^2)$.

2. **Caminhamento DFS**

   - A DFS precisa percorrer cada vértice e cada aresta do grafo. Dessa forma, apresenta uma complexidade $O(V + E)$, onde 'V' é o número de vértices (caixas) e 'E' é o número de arestas (encaixes). Logo a complexidade é linear $O(n)$.

Sendo assim, como o a parte do algoritmo que mais gerou operações resultou em uma complexidade $O(n^2)$, é possível tomá-la como a complexidade total do algoritmo, pois ela dominará seu tempo de execução. Assim, diz-se que o algoritmo possui uma complexidade polinomial quadrática.

Conclusões

Neste trabalho, foi desenvolvido e explicado um algoritmo para encontrar a maior sequência de caixas encaixáveis, utilizando conceitos de grafos dirigidos e caminhamento em profundidade (DFS). Partindo desse contexto, a abordagem proposta demonstrou-se ser eficaz na resolução do problema, fornecendo uma solução que, embora tenha uma complexidade polinomial $O(n^2)$, se mostra adequada para uma ampla gama de tamanhos de entrada, conforme demonstrado nos testes realizados. Assim, é possível afirmar que o algoritmo é capaz de lidar de forma eficiente com conjuntos de dados de diferentes tamanhos, de dez a dez mil caixas. Ademais, é importante visualizar que o uso da DFS para encontrar a maior sequência de caixas encaixáveis mostrou-se ser confiável para encontrar uma solução satisfatória. Levando em consideração a possibilidade de haver pesquisas futuras acerca do mesmo problema, otimizações adicionais poderiam ser exploradas, como a redução do número de comparações ao verificar o encaixe das caixas, acelerando a busca do maior caminho. Em suma, tendo em vista o breve tempo destinado à realização deste trabalho, foi possível desenvolver um algoritmo que resolve o problema proposto de forma eficiente e que oferece uma base sólida para futuras melhorias e adaptações.

Referências

[1] COHEN, M. **Boxes**. Disponível em: Boxes. Acesso em: 30 jun. 2024.

[2] DE OLIVEIRA, V. A.; RANGEL, S.; DE ARAUJO, S. A. **Teoria dos Grafos**. Disponível em: Teoria dos grafos. Acesso em: 30 jun. 2024.

[3] FEOFILOFF, P. **Algoritmo de busca em profundidade num digrafo**. Disponível em: Algoritmo de busca em profundidade num digrafo. Acesso em: 30 jun. 2024.

[4] **Graphviz Online**. Disponível em: Graphviz Online. Acesso em: 30 jun. 2024.

[5] COHEN, M. **Introdução à Complexidade de Algoritmos**. Disponível em: Introdução à Complexidade de Algoritmos. Acesso em: 30 jun. 2024.