Relatório Técnico

Trabalho 2 - Alest II

Fernanda Franceschini, Luana Sostisso

Escola Politécnica— PUCRS

24 de junho de 2024

# Resumo

Este trabalho visa apresentar uma solução para o problema proposto na disciplina de Algoritmos e Estrutura de Dados II no terceiro semestre. O problema consiste em encontrar a maior sequência de caixas que podem ser colocadas uma dentro da outra, a partir de um catálogo que lista todas as dimensões das caixas fabricadas. As dimensões das caixas (largura, altura e comprimento) não estão identificadas e se misturam dentro do arquivo, o que adiciona um desafio adicional à tarefa. Para abordar essa questão, foi necessário desenvolver um algoritmo capaz de ordenar as caixas de acordo com suas dimensões, garantindo que uma caixa só possa ser colocada dentro de outra se todas as suas dimensões forem menores, e que fosse capaz de determinar a maior sequência de caixas que podem ser aninhadas uma dentro da outra. Através dessa narrativa, destacamos as lições aprendidas sobre otimização, escolha de estruturas de dados e implementação de algoritmo. Além disso, foram realizados testes de casos, cujos resultados são apresentados ao final deste artigo.

# Introdução

O ponto de partida para este artigo surgiu durante o curso de Algoritmos e Estrutura de Dados II, onde os alunos foram desafiados com um problema e solicitados a desenvolver um algoritmo para resolvê-lo de maneira otimizada, visando minimizar o consumo de recursos, tais como o tempo de execução.

Para resolver este problema, pensamos na melhor forma de implementar um algoritmo eficiente utilizando caminhamento em grafos. Este algoritmo deve ser capaz de ler e processar as dimensões das caixas, modelar o problema como um grafo, e aplicar técnicas de caminhamento para determinar a maior sequência possível de caixas aninhadas. O objetivo é desenvolver um algoritmo que não apenas resolva o problema, mas que o faça de forma otimizada, minimizando o tempo de execução e a complexidade computacional.

Texto, Tabela

Descrição gerada automaticamente

Figura 1: exemplo arquivo txt

Diagrama

Descrição gerada automaticamente

Figura 2: exemplo do dígrafo implementado

Na *figura 1* consta um arquivo texto que contêm em cada linha as dimensões largura, altura e comprimento de cada caixa. A *figura 2* é uma representação de como será implementado o caminhamento, na qual as menores caixas apontam para as “maiores” caixas, aquelas que elas caberiam dentro.

Algumas informações importantes para compreender melhor o código são:

* As dimensões não são identificadas e estão misturadas dentro do arquivo fornecido, o que torna o problema mais desafiador. Nesse caso, é necessário ordenar as dimensões de forma que as maiores medidas sejam consideradas como a largura da caixa, as menores medidas como a profundidade, e o valor restante como a altura.
* As caixas menores apontam para as caixas maiores. Se, por exemplo, uma das dimensões da caixa x for maior que a da caixa y e as outras dimensões forem menores, significa que uma caixa não cabe dentro da outra. Uma caixa só caberá dentro da outra se todas as suas dimensões forem menores que as da caixa maior.

A organização e o desenvolvimento deste artigo foram feitos a partir dos seguintes pontos:

* Identificação da abordagem mais adequada de caminhamento de grafos para implementar o código destinado à resolução do problema acima;
* Detalhamento do processo de modelagem do problema e do desenvolvimento da solução;
* Discussão dos casos de teste conduzidos para avaliar a eficácia e a precisão do programa implementado.
* Resultados obtidos apresentados de maneira a fornecer uma análise conclusiva sobre a performance e a

viabilidade do algoritmo proposto.

A seguir, apresentaremos como o problema foi modelado, descreveremos os algoritmos utilizados em cada etapa, demonstraremos os resultados obtidos nos casos de teste e, por fim, compartilharemos nossas conclusões sobre a abordagem final adotada.

Primeira versão do Algoritmo

# 1. Leitura das Informações

Depois de considerar o problema, percebe-se que o primeiro passo é armazenar as dimensões de uma caixa em um arquivo de texto e lê-lo. Com esta constatação, foi utilizado a classe FileReader e BufferedReader para transformar um arquivo de texto em um objeto, permitindo que seja possível abrir e ler o seu conteúdo linha por linha. Dessa forma, consegue-se que os dados sejam acessados de forma programática e utilizados dentro do código para realizar as operações necessárias

Usamos uma instancia da classe *FileReader* para abrir um arquivo de texto especificado pelo caminho *"src/CasosTeste/casoTest.txt*". Para poder ler linha por linha do arquivo fonte, foi necessário usar também a classe *BufferedReader* e junto dela um bloco *try-catch* para garantir que a classe será fechada automaticamente ao final do bloco, evitando possíveis vazamentos de recursos.

Inicialmente, declara-se uma variável " linha" para armazenar temporariamente o conteúdo de cada linha do arquivo. Em seguida, é iniciado um loop while que continua enquanto houver linhas a serem lidas no arquivo. Durante cada iteração do loop, a próxima linha do arquivo é lida e armazenada na variável "linha". Feito isso, a linha lida é então dividida em caracteres (símbolos) por meio do método "split" com parâmetro nulo – separando de um em um - resultando em um array de Strings chamado "caracteres".

Em seguida, é criado três variáveis *val1, val2, val3* para armazenar cada elemento do array "caracteres" e atribui-lo mais além nas dimensões de cada caixa *largura, altura e comprimento* . Entretanto, o array “caracteres” é do tipo String,. Dessa forma, para realizar a atribuição as variáveis, é necessário utilizar o método *"Integer.parseInt()”* servindo, nesse contexto, somente como um conversor de tipo, levando em consideração que existe somente um caractere em cada posição do array “caracteres”.

# 2. Criação da Classe Caixa

Para iniciar a implementação do código, é fundamental criar uma classe Caixas que seja capaz de armazenar e gerenciar todas as informações pertinentes de cada caixa. A classe Caixa deve encapsular as dimensões de largura, altura e profundidade de cada caixa, permitindo acesso seguro a esses dados através de métodos específicos. Além disso, fornece métodos que facilitam a manipulação e a comparação dessas dimensões dentro de algoritmos que lidam com essas caixas. A classe caixa foi desenvolvida utilizando o paradigma de orientação a objeto.

Cada instância da classe *Caixa* é inicializada com suas dimensões por meio de seu construtor. As dimensões são armazenadas como variáveis privadas na classe e podem ser acessadas por métodos públicos de acesso (getters).

O método *toString* é sobrescrito para fornecer uma representação textual da caixa, exibindo suas dimensões formatadas de maneira legível.

# 3. Processo de Ordenação das Dimensões

Para solucionar o problema, também é necessário primeiro determinar quais são as dimensões das caixas, visto que estas estão distribuídas de forma desordenada no arquivo de texto. O objetivo é processar cada conjunto de dimensões de modo que as três medidas de cada caixa sejam classificadas de forma consistente, facilitando as comparações subsequentes.

A forma que encontramos para fazer essa tarefa foi de ordenar as dimensões de cada caixa em maior valor, valor intermediário e menor valor. Assim foi aplicado para todas as caixas. Desta forma, conseguimos ter consistência na comparação entre as caixas e aumentamos a probabilidade de uma caber dentro da outra, visto que segue uma lógica de ordenação. É como se a fábrica produzisse dimensões de caixa que respeitassem um padrão (p.e. largura a maior dimensão, a altura a dimensão intermediária e a comprimento a menor dimensão), o que faria que se em dimensões menores, todas as caixas caberiam uma dentro das outras.

criar uma lista vazia de caixas

abrir o arquivo "src/CasosTeste/casoTeste.txt" para leitura

enquanto houver linhas para ler no arquivo

ler a linha

dividir a linha em valores separados por espaço

converter valores para inteiros val1, val2, val3

se val1 for maior que val2 e val3

largura = val1

se val2 for maior que val3

altura = val2

profundidade = val3

senão

altura = val3

profundidade = val2

senão se val2 for maior que val1 e val3

largura = val2

se val1 for maior que val3

altura = val1

profundidade = val3

senão

altura = val3

profundidade = val1

senão

largura = val3

se val1 for maior que val2

altura = val1

profundidade = val2

senão

altura = val2

profundidade = val1

adicionar uma nova caixa com largura, altura, profundidade à lista de caixas

O código realiza a leitura das dimensões de caixas a partir de um arquivo de texto, separando e convertendo os valores para inteiros.

Para garantir que as comparações entre caixas sejam consistentes, o código ordena as três dimensões de cada caixa. Ele identifica a maior dimensão e a classifica como a largura. Em seguida, compara as duas dimensões restantes para determinar qual será a altura (maior) e qual será o comprimento (menor). Este processo é repetido para cada linha do arquivo, garantindo que todas as caixas tenham suas dimensões ordenadas de forma consistente. Essas dimensões ordenadas são então usadas para criar instâncias da classe *Caixa*, que são armazenadas em uma lista para uso posterior na solução do problema de aninhamento de caixas.

# 4. Montando o Grafo

No contexto do problema de aninhamento de caixas, optamos por representar as caixas como vértices de um dígrafo (grafo direcionado). Essa escolha se baseia na relação hierárquica entre as caixas: uma caixa menor pode ser aninhada dentro de uma caixa maior, mas não o contrário. Portanto, se uma caixa A é menor que uma caixa B, há uma aresta direcionada de A para B no grafo, significando que não há arestas que apontem de um vértice maior para um vértice menor, apenas no sentido oposto, de menor para maior.

Além disso, não há ciclos no grafo, pois não é possível ter uma sequência de caixas onde a última caixa da sequência possa ser aninhada dentro da primeira. Isso garante que o grafo seja acíclico (DAG - Directed Acyclic Graph), o que simplifica a implementação de algoritmos como o de busca em profundidade (DFS) ou o de caminho mais longo.

public Digraph (ArrayList<Caixa> caixas) {

    graph = new HashMap<>();

    vertices = new HashSet<>();

    totalVertices = totalEdges = 0;

    for (int i = 0; i < caixas.size(); i++) {

      for (int j = 0; j < caixas.size(); j++) {

        if (i != j) {

          Caixa caixa1 = (Caixa) caixas.get(i);

          Caixa caixa2 = (Caixa) caixas.get(j);

          if (caixa1.getLargura() < caixa2.getLargura() && caixa1.getAltura() < caixa2.getAltura()

              && caixa1.getProfundidade() < caixa2.getProfundidade()) {

            addEdge(caixa1, caixa2);

          }

        }

      }

    }

  }

Ao observar o código fornecido para construir o grafo de aninhamento de caixas, podemos detalhar melhor o processo. Utiliza-se um laço duplo que itera sobre todas as caixas disponíveis, resultando em uma complexidade de tempo O(n^2), onde n representa o número de caixas. Durante essa iteração, cada caixa é comparada com todas as outras caixas para determinar se pode ser aninhada dentro delas. As caixas são então representadas como vértices no grafo, e arestas direcionadas são adicionadas conforme as relações de aninhamento entre elas.

Na primeira versão do algoritmo, havíamos adicionado um else if para o if criado, para adicionar a aresta no sentido oposto, testando as condições no novamente, mas com os operadores de maior que e menor que trocados. Isso fez com que o for interno começasse em i+1, na teoria poupando tempo de percorrer nodos e não entrando tantas vezes no if, poupando operações. Surpreendentemente, o algoritmo veio a ficar consideravelmente mais lento, o que achamos estranho de primeira vista, pois operações tendem a ser rápidas. Para entender o motivo, adicionamos um contador de operações e um contador de nodos percorridos e rodamos o código no último caso, casoTeste10000.txt.

Texto

Descrição gerada automaticamente

Imagem 1: Primeira versão do construtor Diagraph.

Texto

Descrição gerada automaticamente

Imagem 2: Resultados conforme código da imagem 1.

Texto

Descrição gerada automaticamente

Imagem 3: Resultados conforme código entregue.

Podemos enxergar dessa forma que realizamos 249.854.950 operações conforme código da imagem 1, enquanto fizemos 349.854.950 operações no código que foi entregue. Apesar do número maior de operações realizados pelo código entregue, ao que tudo indica, percorrer um arrayList consome muito menos tempo do que operações aritméticas. Dessa forma, deixamos o seguindo a lógica de percorrer mais caixas.

Inicializa um mapa (HashMap) que será usado para representar o grafo. Neste contexto, *graph* mapeará cada vértice (caixa) para uma lista de vértices adjacentes (ou conexões). “vertices = new HashSet<>()” é um conjunto (HashSet) que armazenará todos os vértices do grafo, ajudando a garantir que cada vértice seja adicionado apenas uma vez, mesmo que apareça em múltiplas arestas. As variáveis *totalVertices* e *totalEdges* são contadores para o número total de vértices e arestas no grafo que são atualizados ao adicionar vértices e arestas.

O método addEdge(Caixa v, Caixa w) é utilizado para adicionar uma aresta direcionada do vértice v para o vértice w no grafo. Primeiramente, ele chama o método addToList(v, w), que adiciona w à lista de adjacência de v. Em seguida, o método verifica se os vértices v e w já estão presentes no conjunto de vértices vertices. Se não estiverem, ambos são adicionados ao conjunto, e o contador totalVertices é incrementado. O método addToList(Caixa v, Caixa w) gerencia a lista de adjacência de v no grafo. Ele começa obtendo a lista de adjacência atual para o vértice v a partir do mapa graph. Se a lista não existir (ou seja, se v não tiver adjacências), uma nova lista é criada. Depois, w é adicionado a essa lista, e a lista atualizada é colocada de volta no mapa graph associada ao vértice v. O contador totalEdges é incrementado para refletir a adição da nova aresta no grafo.

Essa abordagem de modelagem através de um dígrafo oferece uma representação clara e eficiente das hierarquias entre as caixas. Isso facilita a aplicação de algoritmos que buscam encontrar o caminho mais longo de aninhamento entre as caixas ou realizar outras operações essenciais para a solução do problema proposto. Os métodos implementados na classe Diagraph foram baseados em material prévio da disciplina, acrescentados de ajustes que fizemos ao contexto.

# 5. Caminhamento pelo Grafo

O objetivo do algoritmo, como foi visto, no contexto do problema de aninhamento de caixas é encontrar a sequência mais longa de caixas que podem ser aninhadas uma dentro da outra. Essa abordagem busca determinar o caminho que contém o maior número de caixas, representadas como vértices no grafo, e arestas direcionadas que indicam a possibilidade de uma caixa ser aninhada dentro de outra. Logo, para encontrar o caminho máximo no grafo, pode-se aplicar técnicas como a busca em profundidade (DFS - Depth-First Search) ou algoritmos de caminho mais longo em DAGs (grafos direcionados acíclicos). Esses métodos exploram recursivamente as conexões entre as caixas, buscando estender o caminho o máximo possível. Isso é garantido pela estrutura de um dígrafo, onde cada caixa menor aponta para caixas maiores, refletindo a relação de aninhamento físico das caixas.

O processo de busca envolve explorar todos os nós descendentes de um nó atual antes de retroceder para explorar outros caminhos. Isso garante que todos os nós sejam visitados de maneira sistemática e que nenhum seja abandonado na exploração. No contexto do problema de aninhamento de caixas, essa abordagem garante que todos os potenciais aninhamentos entre as caixas sejam considerados. Após explorar todas as opções para uma caixa específica, retornamos para verificar se há outras caixas menores que ainda não foram consideradas como potencialmente aninháveis dentro de caixas maiores.

private static Map<Caixa, Integer> longestPathCache = new HashMap<>();

private static int longestPathLength = 0;

Digraph diagraph = new Digraph(caixas);

for (Caixa caixa : diagraph.getGraph().keySet()) {

            int currentPathLength = findLongestPath(diagraph, caixa);

            longestPathLength = Math.max(longestPathLength, currentPathLength);

        }

        System.out.println("O maior caminho possível é: " + longestPathLength);

    private static int findLongestPath(Digraph diagraph, Caixa caixa) {

        if (longestPathCache.containsKey(caixa)) {

            return longestPathCache.get(caixa);

        }

        int maxLength = 0;

        for (Caixa adjCaixa : diagraph.getAdj(caixa)) {

            int pathLength = findLongestPath(diagraph, adjCaixa);

            maxLength = Math.max(maxLength, pathLength);

        }

        longestPathCache.put(caixa, 1 + maxLength);

        return 1 + maxLength;

    }

O código apresentado implementa um algoritmo para encontrar o maior caminho possível de aninhamento de caixas em um grafo direcionado (*Digraph*). Inicialmente, são inicializadas duas estruturas principais: *longestPathCache*, um mapa que armazena o comprimento do caminho mais longo conhecido a partir de cada caixa para evitar cálculos repetidos, e *longestPathLength*, uma variável que mantém o comprimento do maior caminho encontrado até o momento.

O processo começa criando um objeto *Digraph* com base na lista de caixas fornecida. Em seguida, o programa itera sobre todas as caixas no grafo, calculando o comprimento do caminho mais longo iniciando de cada uma delas. Para cada caixa, o método *findLongestPath* é invocado para determinar o comprimento máximo do caminho a partir dessa caixa. Esse método utiliza recursão para explorar todos os vértices adjacentes à caixa atual no grafo. Para evitar o recalculo repetido, o resultado de caminhos mais longos já calculados é armazenado em *longestPathCache*, um mapa que associa cada caixa ao seu caminho mais longo conhecido até o momento, número de nodos que conseguiu passar.

Ao calcular *pathLength* para cada vértice adjacente, o método *findLongestPath* determina o comprimento do caminho mais longo a partir daquele vértice, e *maxLength* armazena o maior valor encontrado entre todos os vértices adjacentes à caixa. Isso é repetido até que todos os caminhos possíveis tenham sido explorados a partir da caixa atual. Ao final do processo, o valor *longestPathLength* armazenará o comprimento do maior caminho encontrado em todo o grafo de aninhamento de caixas, pois é atualizado a cada nova caixa.

Essa abordagem utiliza uma combinação de recursão e armazenamento em cache para otimizar o cálculo do caminho mais longo, explorando as possibilidades de aninhamento entre as caixas. A implementação resultou em uma forma intuitiva de ler o algoritmo, mais facilitada na hora de entender a lógica. Ao final do algoritmo é exibido o maior caminho encontrado para o usuário.

Nota-se que essa abordagem não armazena o caminho mais longo em si, mas a distância de nodos percorridas. Dessa forma, se quiséssemos saber qual foi o caminho e por quais nodos passou, deveríamos alterar a estrutura apresentada acima.

# 6. Casos Teste

Os casos de testes que foram utilizados nesse trabalho foram fornecidos pelo professor da disciplina. São 11 arquivos, que variam de tamanho, tendo mais ou menos quantidade de caixas dentro deles. O gabarito também foi disponibilizado pelo professor.

A nossa solução de algoritmo rodou em todos os casos de teste disponibilizados, batendo com os resultados fornecidos.

# 7. Resultados

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Arquivo | Caso00010 | Caso00020 | Caso00050 | Caso00100 | caso00200 | caso00300 | caso00500 | caso01000 | caso02000 | caso05000 | caso10000 |
| Caminho | 3 | 5 | 8 | 12 | 17 | 20 | 24 | 32 | 42 | 59 | 76 |
| Tempo(s) | 0.0025696 | 0.0043093 | 0.0040366 | 0.0076439 | 0.0174113 | 0.0246399 | 0.0321056 | 0.0831711 | 0.1619059 | 0.5537173 | 1.8202554 |

Consideramos que o algoritmo respondeu bem ao problema. Podemos concluir que o algoritmo apresentou um desempenho eficiente, considerando os tempos de execução em relação ao número de caixas processadas. O tempo de execução é proporcional e tende a linearidade ao número de caixas, pois o passamos por todos os nodos na execução do algoritmo. Os caminhos não necessariamente apresentam a mesma proporcionalidade, pois podemos ter tamanhos variados que não combinam entre si. No gráfico abaixo pode ser analisada a curva entre tamanho e caminhos encontrados. A curva, neste caso, segue uma proporção, mas não uma linearidade.

Gráfico 1: linha de tam por caminho dos resultados encontrados.

**8. Dificuldades**

Não tivemos grandes dificuldades na execução deste trabalho. Antes de começarmos tivemos uma grande troca de ideias entre a lógica da solução entre os colegas da turma. A ideia inicial era começar mapear os filhos de cada nodo, selecionar o nodo com mais filhos e começar com ele a traçar os caminhos. Porém, enquanto desenvolvíamos percebemos que usar uma variável para marcar os nodos já visitados também era uma boa estratégia, o que traria a otimização desejada na primeira ideia.

Utilizou-se o material disponibilizado no moodle da disciplina sobre diagraphs e conseguimos implementar os algoritmos sem grandes problemas. A classe Caixa foi implementada com a estrutura base de orientação a objeto, também sem grandes problemas.

Na implementação da classe Diagraph, tivemos que alterar para que recebesse uma lista. Primeiramente, havíamos utilizado linkedList, sem pensar sobre o que estávamos fazendo, o que tornou o algoritmo extremamente lento, tendo dois get() dentro de 2 fors aninhados, complexidade de O elevado na quarta. Quando percebemos o motivo da lentidão, alteramos para ArrayList e o problema foi resolvido consideravelmente.

# 9. Conclusão

Os resultados deste trabalho demonstraram que, através da implementação de um algoritmo, foi possível analisar a quantidade de caixas potencialmente aninhadas em diversas situações. No entanto, a aplicação prática dessa solução ainda é inviável para aplicações reais, pois a espessura das caixas não foi considerada. Contudo, a solução possui grande potencial, especialmente quando essa informação for adicionada.

As técnicas de memoização que utilizamos foram cruciais para o bom desempenho do algoritmo. O map criado para armazenar os caminhos já passados foi importante para evitar processamento desnecessário. Com isso, chegamos na parte de busca com um algoritmo O(n), visto que é uma busca recursiva na qual os nodos visitados não precisam ser passados novamente. A complexidade do algoritmo na criação do grafo é O(n²).

A implementação do algoritmo para resolver o problema de encontrar a maior sequência de caixas aninháveis mostrou-se eficaz e educativa. O desenvolvimento deste trabalho abrangeu diversos aspectos fundamentais da disciplina de Algoritmos e Estrutura de Dados II, destacando a importância da escolha adequada das estruturas de dados e da otimização de algoritmos.

Este trabalho não só ofereceu uma solução prática para o problema proposto, mas também aprofundou o entendimento sobre algoritmos de grafos e técnicas de otimização, consolidando os conhecimentos adquiridos ao longo do semestre.