Trabalho Prático 3
Algoritmos I

Luís Felipe Ramos Ferreira
2019022553

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

lframos_ferreira@outlook.com

1. Introdução

A proposta do trabalho prático 3, da disciplina de Algoritmos, foi implementar um algoritmo que, ao receber uma planta de de uma casa, em um formato de uma matriz binária, assim como uma lista de possíveis mesas para a casa, fosse encontrada a mesa de maior área dentre aquelas da lista que coubessem na planta segundo as especificações passadas.

Os detalhes da implementação e modelagem deste problema, assim como a análise de complexidade das funções utilizadas e as instruções de compilação, estão descritas ao longo desta documentação.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection, no sistema operacional Windows.

2.1. Modelagem computacional

Para realizar uma análise da modelagem computacional utilizada, primeiro é necessário reiterar o que exatamente o trabalho pede.

A planta da casa é representada por uma matriz que contém apenas dois valores, que podem ser entendidos como espaço vazio e espaço não vazio. Dessa forma, a melhor modelagem para esse caso é o de uma matriz binária, ou seja, uma matriz que admite apenas dois valores. Tendo esta planta armazenada, é necessário, a partir de uma lista de dimensões de mesas, checar se essas mesas cabem na casa e, posteriormente, obter a mesa de maior área dentre as que cabem.

A princípio, uma solução não muito eficaz é a primeira que vem à mente. Para cada mesa, realiza-se uma pesquisa de força bruta para checar se ela cabe na planta da casa. No entanto, um algoritmo baseado nisso seria extremamente lento. Dessa maneira, torna-se necessário encontrar uma forma mais eficiente de checar se uma mesa cabe nos espaços vazios da casa.

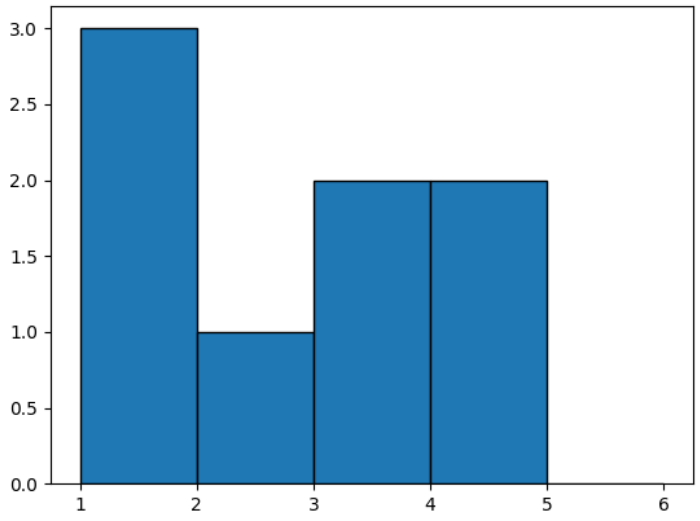
Como foi dito, a interpretação da planta da casa é a de uma matriz binária, o que permitiu a pesquisa e análise de diversos algoritmos para esse tipo de estrutura. O principal deles foi o algoritmo de retângulo de maior área para uma matriz binária. Esse algoritmo não resolve o problema enunciado, mas é uma ótima base para chegar à solução.

Primeiramente, vamos entender como esse algoritmo funciona. Dada a matriz binária, para cada linha da matriz, percorremos um laço e criamos um histograma referente à iteração atual pela matriz. Isso é feito pois cada histograma representa de forma mais simples os espaços que estão disponíveis para a alocação de um retângulo no espaço em questão e, dessa maneira, poderemos utilizar um outro algoritmo, que será detalhado posteriormente, que permite calcular a área máxima de um retângulo dentro de um histograma.

Para cada linha, iremos percorrer um laço e atualizar o histograma, com as seguintes regras: se o valor for 1, somamos esse valor ao retângulo em questão do histograma, caso contrário, zeramos o valor. Para entender melhor o que foi descrito, segue o seguinte exemplo para a seguinte instância de exemplo do problema, onde 1 representa um espaço vazio e 0 o contrário:

Instância de exemplo:	Histogramas para cada linha:
00000	00000
11001	11001
10111	20112
11110	31220

Dessa forma, o histograma da quarta linha, por exemplo, seria o seguinte:



Tendo a criação desses histogramas esclarecida, é necessário explicitar o porquê de utilizá-los. Como explicado anteriormente, existe um algoritmo que permite encontrar, em tempo linear, a maior área existente dentro de um histograma. Entretanto, isto não é uma solução para o problema, visto que um retângulo pode possuir essa mesma área, mas não cabe nos espaços vazios do histograma. Poderíamos, por exemplo, ter um retângulo de maior área no histograma como um retângulo 4x4 (área igual 16) como a maior área, mas uma mesa 2x8, com a mesma área, mas que não cabe nos espaços permitidos.

Dessa forma, precisamos modificar o algoritmo para que, ao invés de checar a maior área dentro do histograma, verificar se um dado retângulo cabe ou não nele. Como essa modificação é feita é detalhada nos próximos tópicos, mas essas modificações podem ser feitas sem alterar a eficiência do algoritmo.

Agora, possuímos uma solução que consegue checar se um determinado retângulo cabe na matriz binária (neste caso, se uma determinada mesa cabe na planta da casa). Com isso, podemos checar apenas aquelas que cabem e, conforme analisamos uma a uma, podemos encontrar aquela de maior área e assim chegar à resposta do problema.

Mas é claro, nem tudo é tão perfeito assim. A solução encontrada dá a resposta certa para instâncias pequenas do problema, mas conforme elas crescem, o tempo necessário para que cheguemos na resposta é muito alto. Mais detalhes são dados na sessão de análise de complexidade.

Nesse cenário, entra a ideia geral do uso de programação dinâmica para melhorar a solução empregada. Para utilizá-la, vamos pensar na seguinte situação: a primeira a mesa da lista de entrada é uma mesa com dimensões N x M, onde N é seu comprimento e M sua largura. Então, rodamos o algoritmo e descobrimos que essa mesa cabe na casa. O que se pode inferir sobre isso?

Ora, é evidente que, se uma mesa N x M cabe na planta da casa, uma mesa menor irá caber também. Ou seja, uma mesa com dimensões N x (M - 1), por exemplo, com certeza irá caber na casa e, portanto, se armazenarmos essa informação em uma tabela, não precisaremos computá-la novamente e poderemos apenas realizar os cálculos em tempo constante para checarmos a área que a mesa cobre.

Uma situação análoga ocorre quando utilizamos o algoritmo e descobrimos que uma mesa com dimensões N x M não cabe na planta da casa. Ora, se ela não cabe, uma mesa com dimensões maiores com certeza também não irá caber, isto é, uma mesa com dimensões N x (M + 1) por exemplo. Dessa forma, assim como antes, se armazenarmos essa informação, poderemos evitar computações repetidas e apenas ignorar essas mesas.

Essa é a ideia geral de como foi utilizado o paradigma da programação dinâmica para melhorar a solução do trabalho. As estruturas de dados e os algoritmos serão mais precisamente explicados no tópico a seguir. Os sites e autores que apresentam os algoritmos nos quais me baseei estão dispostos nas referências bibliográficas ao fim desta documentação.

2.2. Estruturas de Dados e algoritmos

O trabalho foi realizado, como dito anteriormente, na linguagem de programação C++. Essa linguagem possui diversas funcionalidades e bibliotecas que podem ser utilizadas para resolver problemas de forma simples. Especificamente para este trabalho, o uso da estrutura de dados *vector* (vetor), da biblioteca padrão, foi de excelente utilidade na representação dos histogramas utilizados nos algoritmos e da planta da casa como uma matriz binária.

Em primeiro lugar, vamos abordar a forma como o algoritmo principal do trabalho foi implementado. Isto é, o algoritmo para dizer se uma mesa cabe ou não na planta da casa. Como especificado na modelagem computacional, isso é feito transformando cada linha da matriz que representa a planta da casa, a cada iteração, em um histograma diferente. Esses histogramas são representados por um *vector* de inteiros, onde o valor indica a altura da barra no histograma. Assumiu-se que cada barra possui um comprimento unitário, por motivos de simplicidade.

Cada histograma criado foi armazenado em um outro *vector*, criando assim o que pode ser chamado de matriz de histogramas. Essa estrutura permitiu que os histogramas criados fossem acessados rapidamente, sem custo de computação repetitivo, acelerando a performance do algoritmo.

O algoritmo para checar se uma mesa cabe ou não na planta da casa funciona de forma intuitiva. Para cada histograma, temos N possibilidades de altura 'mínima' de cada retângulo, isto é, podemos considerar, um por vez, cada barra como a menor altura possível para um retângulo na estrutura. No entanto, para cada um desses casos, como podemos saber qual o comprimento máximo permitido de um retângulo?

Por meio da estrutura de uma *stack* (pilha), podemos manter um controle de qual índice do histograma estamos checando no momento. Enquanto encontrarmos barras de tamanho maior ou igual à presente no topo da pilha, adicionamos esses novos índices à pilha. Caso contrário, retiramos o índice do topo da pilha.

Ao retirarmos um índice da pilha, consideramos a barra deste índice como a de maior altura permitida no momento. O detalhe do maior comprimento se destaca agora. Se a pilha estiver vazia após retirarmos esse elemento, o próprio índice é considerado como o comprimento máximo. Isso acontece pois, se a pilha estiver vazia, quer dizer que aquela é a menor altura conhecida para uma barra do histograma. Em contrapartida, se a pilha não estiver vazia, o maior comprimento permitido é igual ao valor do índice da barra atual menos o valor do índice da barra que está no topo da pilha, uma vez que esta segunda teria uma altura menor que a atual, gerando uma contradição com o que definimos como a menor altura permitida no momento.

Tendo isso esclarecido, é simples notar que, para cada um desses retângulos, checamos se as dimensões da mesa atual cabem no retângulo gerado. Se sim, já podemos retornar um valor *true* e garantimos que a mesa cabe. Caso contrário, checamos todos os retângulos possíveis e, se nenhum permitir as dimensões da mesa, retornamos *false*.

Nesse cenário, para garantir que checamos cada retângulo de cada histograma, criamos uma função auxiliar que utiliza o algoritmo anterior para cada um dos histogramas da planta da casa.

Em relação à eficiência do código, como dito na modelagem computacional, foi utilizado o paradigma da programação dinâmica para evitar computações sobre mesas das quais já temos informações. Em relação às estruturas de dados, bastou utilizar uma matriz que armazena, na coordenada (i , j) se a mesa com essas dimensões, isto é, comprimento i e largura j (e vice versa, devido a possível rotação da mesa), cabem na casa, não cabem na casa, ou se não sabemos se cabe ou não. Isso, como explicitado, permite evitar computações repetidas, bastando checar o valor que a dimensão da mesa atual possui nessa matriz.

3. Análise de Complexidade de Tempo

Função 'tableFitsHistogram': essa função realiza as computações e cálculos necessários para checar se a mesa cabe em um histograma. Ela percorre cada índice do vetor do histograma, checando cada retângulo máximo permitido por aquela altura de índice no momento e, devido a isso, possui uma complexidade linear para o tamanho de cada histograma. Como especificado, cada histograma possui um tamanho igual ao comprimento da planta da casa, que podemos chamar de C. Logo, a complexidade de tempo da função é O(C).

Função ‘tableFitsHouse’: essa função utiliza a função anterior para cada um dos histogramas da planta da casa. Como especificado, a quantidade de histogramas é igual à largura da planta, que podemos chamar de L. Assim, em um pior caso onde todos os histogramas devem ser checados, a função possui uma complexidade quadrática O(LC).

A execução do da função principal, em um cenário sem otimização, seria O(NLC), pois chamaria a função ‘tableFitsHouse’ para cada uma das N mesas de entrada. No entanto, com o uso de programação dinâmica, pudemos melhorar essa implementação. Imaginando um pior caso, onde a análise de uma mesa não permite inferir nada novo sobre as mesas que serão analisadas, o algoritmo continuaria com a mesma complexidade.

No entanto, vamos imaginar um caso médio, onde cada mesa consegue cortar aproximadamente 1/4 das mesas que serão analisadas subsequentemente, seja por caber ou não na casa. Em uma matriz 5x5, por exemplo, se descobrirmos que a mesa de dimensões 3x3 não cabe na planta da casa, poderíamos ‘cortar’ o quarto formado pelas mesas de dimensões 3x4, 3x5, 4x4, 4x5, 5x4, 5x3, 4x3, 5x5. Julgando um caso médio acontecendo constantemente, o número de vezes que a função 'tablets House' será chamada será logarítmica para o número de mesas de entrada e, portanto, a o programa como um todo terá um gargalo de complexidade igual a O(LClogN).

4. Referências bibliográficas

https://www.geeksforgeeks.org/maximum-size-rectangle-binary-sub-matrix-1s/

https://www.geeksforgeeks.org/largest-rectangle-under-histogram/

https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2G1pAq0OO0#:~:text=Dynamic%20Programming%20(DP)%20is%20an,optimal%20solution%20to%20its%20subproblems.

https://www.youtube.com/watch?v=oBt53YbR9Kk

5. Compilação e execução

- Acesse o diretório TP
- No terminal, utilize o seguinte comando:
  g++ main.cpp -o main
- Para utilizar o executável gerado, utilize no terminal o comando:
  ./main
  seguido da entrada desejada para o programa.
- Exemplo de execução e saída esperada:
  g++ main.cpp -o main
  ./main

  6 16
  ################
  #..#..........#
  #.....#.......#
  #.....#.......#
  #.....#...####
  #########..#####
  4
  3 3
  4 6
  15 2
  2 7

  2 7