Trabalho 2

Augusto Calado (9779134) e Fernando Chiu (9436743)

Novembro 2020

1 Ordenação : Presentes Grandes (2720)

1.1 Contexto

Neste problema, é solicitado que ajudemos o Bruninho a selecionar presentes de natal. Os presentes são descritos como caixas retangulares e quanto maior for o volume da caixa, maior será o valor do presente. A entrada do problema contêm as dimensões (comprimento, largura e altura) e os id's de uma lista de n presentes, e é pedido para se imprimir os k presentes de maior valor ordenados lexicograficamente pelo id.

1.2 Solução

A solução envolveu, primeiramente, computar os volumes de cada presente ($Volume = altura \times comprimento \times largura$). A seguir os presentes foram ordenados a partir do volume. Por fim, os presentes de mesmo volume foram ordenados lexicograficamente pelo id.

A complexidade da solução desenvolvida é $O(nlog(n))$, pois foi utilizado o *MergeSort*. Nota-se que, apesar de duas ordenações terem sido realizadas sequencialmente, a complexidade da solução é a mesma.

2 Tentativa e erro: O enigma do príncipe (1843)

2.1 Contexto

No problema enigma do príncipe, o jogo *Flood It* é descrito. Neste jogo, dado uma matriz retangular de cores, o jogador vence quando esta se torna monocromática. A cada momento, o jogador pode selecionar uma cor que será aplicada à grade. Esta cor começa colorindo a célula superior esquerda e se espalha por todas as células adjacentes que tenham a cor da célula inicial ou a cor selecionada. O objetivo do problema é, dado uma configuração inicial da matriz, determinar a quantidade mínima de passos para torná-la monocromática.

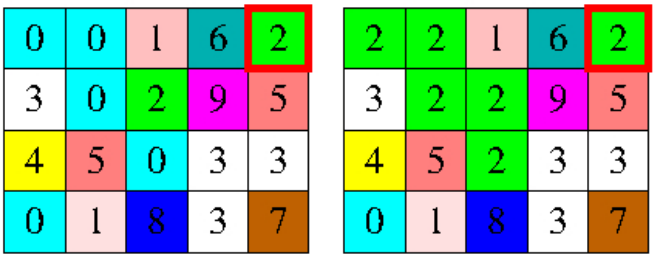

Figura 1: Matriz do jogo *Flood It*

2.2 Solução

A solução do problema envolveu testar recursivamente todas as possibilidades de atribuição de cores para a grade. Um contador de passos global foi utilizado para limitar a profundidade das chamadas recursivas. Caso uma solução s fosse encontrada em uma profundidade d, nenhuma chamada recursiva seria feita para profundidades $d+1$. O pseudocódigo da chamada recursiva da tentativa e erro é mostrado abaixo:

Algoritmo 1: Tentativa

```
Data: Matrix m, int profundidade
if profundidade ≥ limitemin then
    Retorne
end if
for Cor c ∈ CoresPossiveis do
    Aplique c na matriz m
    if m é monocromática then
        limitemin ← profundidade
    end if
    Tentativa(m, profundiade + 1)
    Reverta a cor a aplicação da cor c na matriz m
end for
```

Considerando que o fator de ramificação médio das cores possíveis seja C e que a profundidade mínima seja D, a complexidade da solução é $O(D^C)$.

O código desenvolvido passou em todos os casos te teste do *Udebug*, contudo não passou no URI por conta do erro *timelimit*. Assim, ele foi substituído pelo problema Botas perdidas da categoria 2 ad-hoc.

3 Ad-hoc: Botas Perdidas (1245)

3.1 Contexto

Neste problema, dado um conjunto de botas de diferentes tamanhos e de diferentes pés, é solicitado calcular o número de pares de botas que podem ser

formados.

3.2 Solução

A solução envolveu armazenar as contagens do número de botas para cada tamanho em um *Hashmap*. A quantidade de botas que pode ser formada para cada tamanho é o mínimo entre o número de botas de pé esquerdo e o número de botas de pé direito. Assim, para se computar a solução, bastou-se passar por todas as entradas do *Hashmap* de botas e acumular os mínimos entre contagens esquerda e direita. A solução desenvolvida tem complexidade $O(n)$, onde n é o tamanho de botas, pois o algoritmo passa uma única vez por cada bota durante o processamento.

4 Caminhos mínimos: Containers (2237)

4.1 Contexto

Este problema é situado em um porto de cargas com containers e guindastes que os movem. Dado o peso de dois containers adjacentes, o custo para trocá-los de posição é igual a soma do pesos dos dois. Assim, é pedido no problema, dado uma configuração inicial e uma configuração final de containers, para se descobrir o custo mínimo para sair da configuração inicial e chegar na configuração final.

4.2 Solução

As configurações dos containers podem ser modeladas como nós em um grafo e as trocas de containers como arestas ponderadas. O custo de cada aresta é igual ao custo da troca entre os dois containers envolvidos na transição.

Uma primeira solução que nós desenvolvemos envolveu precomputar o grafo inteiro e então aplicar o algoritmo de Dijkstra para caminhos mínimos. Dado o número de vértices V, e o número de arestas A, a complexidade dessa solução é $O((E+V)log(V))$. Contudo, tendo em vista limitações no tempo e espaço da execução do código, uma outra abordagem foi utilizada.

A solução final implementada envolveu aplicar uma modificação de um algoritmo de busca no grafo. Essa abordagem possui dois benefícios em relação a primeira solução. Em primeiro lugar, não foi necessario precomputar o grafo por completo, e em segundo lugar, a complexidade do algoritmo de busca é $O(V+E)$.

Em específico, a busca implementada começava no nó referente a configuração inicial e mantinha uma fila de prioridades para os nós adjacentes que ainda não foram expandidos. A cada iteração do algoritmo, o nó que possuísse o menor custo era selecionado de uma maneira gulosa para ser expandido. Caso o nó expandido fosse o nó referente a configuração final, a busca era dada como encerrada e o custo era impresso.

5 Programação Dinâmica: Árvore de Natal (2026)

5.1 Contexto

O problema Árvore de Natal consiste em comprar uma quantidade X de pacotes de enfeites natalinos de tal forma que maximize o número de enfeites utilizados em cada galho da árvore. Os pacotes possuem pesos e eles são variados dependendo do enfeite que está contido nele, e os galhos da árvore possuem um peso máximo de enfeites que pode-se suportar.

A entrada consiste:

- quantidade de galhos presentes na árvore,
- número de pacotes de enfeites ,
- capacidade de peso que o galho da árvore suporta,
- número de enfeites em cada pacote E, e o peso de cada pacote PC.

5.2 Solução

A solução para o problema da Árvore de Natal utiliza a técnica da programação dinâmica para encontrar a melhor quantidade de enfeites por galho.

A abordagem aplicada é muito próxima da abordagem utilizada para resolver o problema da mochila booleana. O algoritmo se baseia na utilização de uma matriz para guardar o resultado da soma dos pesos dos pacotes já utilizado nas soluções intermediárias. A matriz possui uma linha para cada pacote e as colunas vão de zero até o peso máximo suportado pelo galho.

A quantidade de enfeites ótimas para cada galho é calculado usando uma abordagem tabular de baixo para cima. Portanto, utilizamos uma matriz (númeroPacoteXpeso) onde uma posição x,y indica que o pacote x foi incluído como parte da decoração para um peso y suportado pelo galho.

Considerando que o vetor de pesos por pacote é W e a função recursiva é R, temos a seguinte recorrencia:

$$R(x,y) = \begin{cases} 0 & \text{se } x = 0 \text{ ou } y = 0 \\ R(x,y) = R(x-1,y) & \text{se } W[x] > y \\ max(R(x-1,y), R(x-1,y-W[x]) & \text{se } W[x] <= y \end{cases}$$

A complexidade da solução desenvolvida é $O(N \times Y)$, onde N é o número de pacotes e Y é o peso máximo suportado pelo galho.

6 Busca em Grafos : Detectando Pontes (1790)

6.1 Contexto

O problema Detectando Pontes, é um desafio proposto por um professor para detectar a quantidade de pontes que conectam as cidades não estão contidas em

qualquer ciclo. As pontes são arestas e as cidades são os vértices de um grafo não direcionado

A entrada consiste no número de cidades (vértices), o número pontes (arestas) e as respectivas conexões entre as cidades utilizando as pontes.

A solução do desafio proposto é obter a quantidade de pontes que não estão contidas em qualquer ciclo.

6.2 Solução

Para encontrar a quantidade de pontes que não estão contidas em qualquer ciclo utilizamos a técnica de busca em profundidade (do inglês Depth-first search, abreviadamente DFS). Executa-se um DFS para cada nó não visitado.

Como parte da solução para realizar a contagens de nós que não fazem parte de um ciclo, precisamos criar duas estruturas. A primeira é denominada PTA (primeiro tempo de acesso), que conterá a informação do tempo (em qual nível de recursão da recursão) em que esse nó foi acessado pela primeira vez. A segunda estrutura é denominada NRC (nível da recursão que começou o ciclo), que armazena o tempo (nível da recursão) que o nó que iniciou o ciclo foi acessado. Ambas as estruturas são arrays de tamanho N, onde N é o número de nós do grafo.

Para cada nó (V), faremos a aplicação da DFS para seus adjacentes (adjOfV) não visitados. Ao aplicar uma chamada DFS para um nó fazemos as seguintes operações:

```
RECURSIONLEVEL = RECURSIONLEVEL + 1;
NRC[V] = RECURSIONLEVEL;
PTA[V] = RECURSIONLEVEL;
```

Após o retorno da chamada recursiva de DFS para um nó adjacente, verificamos se ele faz parte de um ciclo fazendo as seguintes comparações:

(I) $NRC[adjOfV] > PTA[vertice]$. Caso esta comparação seja verdadeira significa que não houve ciclo ao se fazer a recursão da DFS para o adjOfV. Portanto adicionamos 1 no contador de pontes que não fazem parte de ciclos;

(II) Caso o adjOfV esteja marcado como visitado, verificamos se o pai de adjOfV é igual ao nó V. Caso essa verificação seja falsa, significa que o adjOfV já foi visitado por um alguém que não é o nó V e portanto fechamos um ciclo. Nesse caso, fazemos a seguinte atribuição ao NRC[V]:

```
NRC[V] = min(NRC[vertice], PTA[adjOfV])
```

Na atribuição acima, escolhemos o tempo "mais cedo" (nível da recursão) que o ciclo começou. Quando a chamada recursiva retorna, a comparação (I) é realizada, e assim detectamos que houve um ciclo e portanto não adicionamos no contador.

A complexidade da solução desenvolvida é $O(V + E)$, pois o programa percorre o grafo usando DFS que é representado usando a lista de adjacência, onde V é o número de nós e E é o número de arestas.

7 Guloso: Produção em Ecaterimburgo (2115)

7.1 Contexto

Este problema está situado em uma fábrica de Ecaterimburgo. Dado um conjunto de tarefas que começam em momentos discretos de tempo e tem duração finita, é pedido para se escalonar as tarefas de modo que a execução de todas elas termine o mais cedo possível.

7.2 Solução

Para selecionar as tarefas que devem ser processadas em primeiro, a abordagem gulosa prioriza a ordem de chegada na fábrica. Portanto, tarefas que chegarem primeiro tem prioridade no processamento.

A solução consiste em ordenar as tarefas pela sua ordem de chegada na fábrica. Após a ordenação, selecionamos de maneira gulosa as tarefas. O algoritmo desenvolvido conta com um contador que guarda o momento final em que uma tarefa foi finalizada. A medida que vamos selecionando as tarefas, verificamos se elas já podem ser executadas, caso isso seja possível adicionamos ao contador o tempo em que a tarefa termina. Caso não seja possível executar a tarefa, pois ela ainda não chegou na fábrica, adicionaremos ao contador o tempo em que a máquina ficou inativa mais o tempo em que a tarefa termina.

Ao realizarmos essa abordagem obtemos um escalonamento tal que as tarefas terminam o mais cedo possível. A complexidade da solução desenvolvida é $O(nlog(n))$, pois ordenamos as tarefas e, em seguida, iteramos sobre a lista ordenada.