Problema de empilhar caixas

Marcela Ribeiro de Oliveira
GRR20157372
Ciência da Computação
Universidade Federal do Paraná
Email: mro15@inf.ufpr.br

I. Compilação e Execução

Para compilar basta executar o comando *make*.

Para executar o programa:

```
./caixas <arquivo>
```

II. Descrição do Problema

Dado um conjunto de **n** caixas onde cada caixa tem 3 dimensões (altura, largura e profundidade). O problema consiste em empilhar as caixas de forma que a altura seja a maior possível. As restrições para que uma caixa possa ser colocada sobre a outra são que a largura e a profundidade da caixa de cima devem ser estritamente menores que a largura e a profundidade da caixa de baixo.

III. Recorrência

A recorrência encontrada para resolver este problema foi:

$$pd\_vector(i) = \begin{cases} height[i] & \text{se } i = 1. \\ MAX\{pd(j)\} + height(i) & \text{caso contrário.} \\ {\scriptstyle widthi<widthj} & \\ {\scriptstyle depthi<depthj} & \\ {\scriptstyle j<i} & \end{cases}$$

Para esta recorrência o vetor pd_vector está ordenado de forma não crescente pela concatenação de largura com profundidade. Por exemplo, para a entrada:

1 1 1

5 2 2

2 2 3

2 5 4

As concatenações são respectivamente:

11

22

23

54

E então as caixas ficam na seguinte ordem:

2 5 4

2 2 3

5 2 2

1 1 1

Dessa forma, é possível percorrer a PD da caixa menor que possivelmente caberia sobre em uma menor. Possivelmente porque por exemplo a caixa 5 2 2 não cabe sobre a caixa 2 2 3. Então é somente uma ordenação para facilitar a execução da PD e a construção da solução.

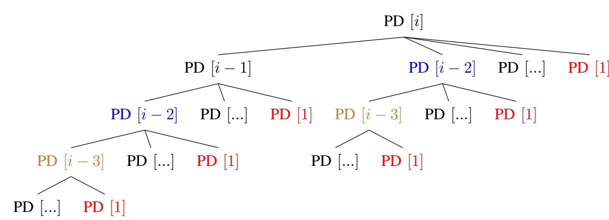
Nesta recorrência então buscamos a maior altura dentre as alturas máximas já calculadas nas posições anteriores do vetor.

Como esta recorrência calcula o valor da altura máxima para cada posição do vetor de caixas, para dar a resposta final é necessário buscar neste vetor o maior valor da altura, ou seja, a maior altura é o máximo do vetor.

A. *Análises*

Utilizando programação dinâmica e começando a preencher o vetor da primeira posição até n o custo é $\mathcal{O}(n^2)$, não o custo da ordenação, somente o custo da PD. Neste problema, programação dinâmica é muito mais eficaz que backtracking, pois, como para calcular cada posição do vetor são necessárias todas as posições anteriores com programação dinâmica evitamos todas as repetições.

1) Algoritmo de Backtracking: Abaixo temos a árvore para a recorrência no caso de um backtracking.



Como pode-se observar, cada nó tem n-1 filhos, portanto o custo do algoritmo de backtracking é $\mathcal{O}(n!)$. Além disso, é possível notar-se também que há muitas repetições, ou seja, cálculos iguais feitos diversas vezes.

Por ser bem mais custosa que a PD e gerar muitas repetições recursão direta torna-se inviável.

2) Algoritmo Guloso: Para que fosse possível construir um algoritmo guloso para este problema utilizando a recorrência apresentada um oráculo seria necessário. Este oráculo é que iria dizer para qual nodo da árvore deve-se descer, formando um caminho pelo qual as caixas empilhadas são feitas de tal forma que a altura é máxima.

Para que o algoritmo guloso seja melhor que o algoritmo de programação dinâmica, a função de oráculo deve ser $o(n)$.

Pois, como percorrer um ramo da árvore da raiz até a folha custará $\mathcal{O}(n)$ e a recursão da PD custa $\mathcal{O}(n^2)$, então a função de oráculo deve custar $o(n)$.

Como este oráculo é desconhecido, não existe um algoritmo guloso para este problema.

IV. Algoritmo Implementado

O pseudocódigo do algoritmo implementado pode ser visto abaixo.

Algoritmo 1: caixas

Entrada: $int\ n$ caixas com $altura$, $largura$ e $profundidade$
Saída: Sequência de caixas (seus índices)

início
  $ordena(\{boxes\})$
  para $i = boxes.begin$ *ate* $boxes.end$ **faça**
    pd_vector[i]$.height$:= $boxes$[i]$.height$
    pd_vector[i]$.ids$:= $boxes$[i]$.id$
  fim
  para i=1 *ate* n **faça**
    para j=0 *ate* i **faça**
      se $boxes[i].width$ <$boxes[j].width$ *e* $boxes[i].depth$ <$boxes[j].depth$ *e* $pd\_vector[i].height$ <$pd\_vector[j].height$ + $boxes[i].height$ **então**
        pd_vector[i]$.height$:= pd_vector[j]$.height$ + $boxes$[i]$.height$
        pd_vector[i]$.ids$:= $\emptyset$
        pd_vector[i]$.ids$:= $\{boxes$[i]$.id\}$
        para it=$pd\_vector[j].ids.begin$ *ate* $pd\_vector[j].ids.end$ **faça**
          pd_vector[i]$.ids$:= pd_vector[i]$.ids$∪it
        fim
      fim
    fim
  fim
  $max := -1$
  $position := -1$
  para i=0 *ate* n **faça**
    se max <$pd\_vector[i].height$ **então**
      $max := pd\_vector$[i]$.height$
      $position := i$
    fim
  fim
  para it=$pd\_vector[position].ids.begin$ *ate* $pd\_vector[position].ids.end$ **faça**
    imprime it
  fim
  retorna 0
fim

Para armazenar as caixas foi utilizado um vetor de structs que contém os seguintes dados:

```
//Boxes
typedef struct box{
    int id;
    int height;
    int width;
    int depth;
    int lex_sort;
}box;
```

Nesta struct lex_sort é onde fica armazenado a concatenação da largura com a profundidade, e como dito previamente, é o valor de lex_sort que é usado para ordenação não crescente.

Já para calcular a PD outro vetor de structs foi utilizado, porém como era necessário os ids das caixas para imprimir no fim da execução, cada elemento deste vetor além de guardar a altura da caixa guarda também um vetor com os ids das caixas empilhadas. Esta estrutura é detalhada abaixo:

```
typedef struct pd_vector{
    std::vector<int> ids;
    int height;
}pd_vector;
```

O algoritmo inicia ordenando não crescente pelo valor de lex_sort o vetor $boxes$. Em seguida, a altura e o id de cada posição de cada posição de $boxes$ é copiada para o vetor pd_vector.

Então é iniciada a PD. Começamos com $i = 1$, pois, $i = 0$ é a própria altura da caixa na posição i do vetor, que já foi preenchido. Dentro do for de i temos um for de j que inicia em 0 e vai até i. Neste for são percorridos os valores de pd_vector que já foram calculados. E se a caixa j cabe sobre a caixa i, e o valor de pd_vector em j somada com a altura da caixa i é maior que o valor já acumulado em pd_vector na posição i, pd_vector na posição i é atualizado com pd_vector na posição j mais a altura da caixa i. Também é atualizado o vetor de ids das caixas que compõem a pilha de caixas.

Após o cálculo da PD, encontra-se o máximo do vetor, e então imprime-se os ids das caixas da pilha de maior altura.

O laço da recorrência da PD executa até o número de caixas da entrada n, então é evidente que o algoritmo termina.

A recorrência da PD calcula as alturas máximas possíveis baseadas nas posições anteriores, onde a primeira posição é a altura da primeira caixa. E um valor só adicionado se aumenta o tamanho da pilha de caixas e a caixa pode ser empilhada. Então as restrições são satisfeitas. Como após a execução da PD há um for que busca pelo máximo do vetor, a resposta devolvida é realmente o maior empilhamento. Dessa forma, podemos concluir que o algoritmo funciona.