Pudełka

Karolina Drabent
Aleksander Truszczyński

Marzec 2020

Spis treści

1 Opis problemu

Dany jest zbiór pudełek B, którego elementy (pudełka) b_i mają dodatnie wymiary naturalne w_i i l_i, nazywane odpowiednio szerokością i długością. Pudełko można obrócić o $90°$, co skutkuje zamianą wartości w_i i l_i.

Mówimy, że pudełko b_j można ułożyć na pudełku b_i wtw. gdy

$$(w_i \geqslant w_j \wedge l_i \geqslant l_j) \vee (w_i \geqslant l_j \wedge l_i \geqslant w_j)$$

Lewa strona alternatywy zachodzi, gdy nieobrócone pudełko j można ułożyć na pudełku i, a prawa, kiedy obrócone pudełko j można ułożyć na pudełku i. Zakładamy, że na każdym pudełku może być ułożone co najwyżej jedno inne pudełko, przy czym jeśli na pudełku b_i stoi jedno pudełko b_j, to ułożenie następnego pudełka b_k na b_j jest dopuszczalne (ponieważ b_k nie jest ułożone bezpośrednio na b_i).

Zagadnieniem, które będzie rozważane w tej pracy, będzie problemem znalezienia najliczniejszego podzbioru $B_F \subseteq B$ takiego, że jego elementy można ułożyć w ciąg $b_1^f, b_2^f, ..., b_m^f$ $(m = |B_F|)$, w którym $\forall_{i,j \in \mathbf{N},\, i<j \leqslant m}$ pudełko b_j można ułożyć na pudełku b_i.

W intuicyjnym sformułowaniu: mając zbiór pudełek, algorytm będzie z nich układał najwyższy stos, w którym każde pudełko musi całą powierzchnią stać na innym pudełku (poza pudełkiem będącym podstawą stosu).

2 Opis rozwiązania

Rozwiązanie zostało zaimplementowane w języku C# i jest aplikacją konsolowa. Algorytm rozwiązujący zadanie jest oparty na programowaniu dynamicznym. Algorytm składa się z trzech części, które są wykonywane po kolei na liście pudełek:

1. obrócenie pudełek, które są w niepoprawnej pozycji,
2. posortowanie listy pudełek,
3. właściwa część algorytmu czyli znalezienie najdłuższej sekwencji pudełek.

Pudełko jest reprezentowane przez klasę Box, która zawiera informacje o jego szerokości, długości oraz czy pudełko jest obrócone. Sam algorytm jest zaimplementowany w klasie statycznej BoxStackingAlgorithm, w której znajdują się odpowiednie statyczne funkcje pomocnicze. Funkcją wykonującą cały algorytm jest funkcja Count, która zwraca gotowe rozwiązanie - listę pudełek.

2.1 Opis pierwszej części rozwiązania

Pozycja poprawna pudełka jest to taka pozycja, w której jego szerokość jest nie mniejsza niż jego długość. W tej fazie, następuje iterowanie po pudełkach i jeśli dane pudełko jest ustawione niepoprawnie to jest ono obrócone. Przykładowo po tej fazie algorytmu lista pudełek:

(3,3), (4,1),(5,19), (1, 2)

będzie wyglądała następująco:

(3,3), (4,1),(19,5), (2,1)

Część ta jest zaimplementowana w funkcji RotateBoxesToProper.

2.2 Opis drugiej części rozwiązania

W tej części następuje sortowanie pudełek. W pierwszej kolejności są one sortowane po szerokości, w drugiej natomiast po długości. Przykładowo po posortowaniu listy pudełek:

(14,3), (1,1), (14,5), (2,2), (4,1)

otrzymujemy następującą listę:

(14,5), (14,3),(4,1), (2,2), (1,1)

Część ta jest zaimplementowana w funkcji SortBoxes przy użyciu wbudowanej funkcji Sort, która wykonuje algorytm Quick Sort. W związku z tym złożoność średnia to $O(nlogn)$ a pesymistyczna to $O(n^2)$.

2.3 Opis trzeciej części rozwiązania

Ta część algorytmu jest zaimplementowana w funkcji GetSequence. Pseudokod tej części jest przedstawiony na diagramie 16.

Algorithm 1: Pseudokod funkcji GetSequence

```
Result: Maksymalna lista pudełek, z których można ułożyć stos
boxes ← posortowana lista pudełek
heights ← tablica list pudełek o wymiarze liczby pudełek w liście boxes,
  zainicjuj i-tą listę i-tym pudełkiem z listy boxes
for i od 1 do liczba pudełek -1 do
    for j od i-1 do 0 do
        if boxes[i] da się położyć na boxes[j] oraz długość listy
          heights[i] < długość listy heights[j] +1 then
            heights[i] ← heights[j] + [ boxes[j] ];
        end
    end
end
// znalezienie maksymalnej listy
max ← heights[0]
for i od 1 do ilość pudełek -1 do
    if długość listy heights[i] < długość listy max then
        max ← heights[i]
    end
end
return max
```

Najpierw inicjowana jest tablica Heights list pudełek, która jest długości listy wszystkich pudełek. W każdej komórce/liście znajduje się na początku jedynie odpowiadające danemu indeksowi pudełko. Można o tej tablicy myśleć jako o tablicy, która w danej komórce o indeksie i przechowuje najwyższy dotychczas znaleziony stos pudełek, na górze którego znajduje się i-te pudełko.

Następnie wykonują się dwie pętle. Pierwsza iteruje po każdym pudełku z listy (zmienna i z zakresu 1 do n-1). Druga po wszystkich nie mniejszych pudełkach w liście wszystkich pudełek (zmienna j z zakresu od i-1 do 0). Sprawdzane w nich jest czy pudełko i-te można położyć na pudełko j-te, a jeśli można to czy stos uzyskany po takim położeniu, byłby większy niż ten aktualnie już znaleziony z pudełkiem j-tym. Można zauważyć, że ponieważ pudełka są posortowane to po każdej iteracji pierwszej pętli w $Heights[i]$ będzie się znajdował maksymalny stos możliwy do uzyskania z pudełkiem i-tym. Jeżeli pudełka nie da się położyć na żadne z wcześniejszych pudełek to $Heights[i] = boxes[i]$. Na końcu tej części wybierana jest ta sekwencja pudełek, która jest najdłuższa.

3 Poprawność algorytmu

Definicje używane dalej w tej sekcji:

- „problemem” nazywamy problem znalezienia najwyższego stosu opisany w sekcji 1
- „podproblemem i-tym” nazwiemy problem znalezienia najwyższego stosu, który ma pudełko b_i na szczycie
- mówimy, że pudełka b_i i b_j są „równoważne” jeśli spełniają warunek $w_i = w_j \wedge l_i = l_j$

Rozwiązanie problemu znajduje się wśród rozwiązań podproblemów $0, ..., n-$ 1-pierwszego, ponieważ najwyższy stos musi mieć jedno z pudełek $b_0, b_1, ..., b_{n-1}$ na szczycie.

Przed podaniem poprawności algorytmu potrzebne będzie pokazanie dwóch lematów.

Lemat 1

Problem i wszystkie podproblemy i mają własność optymalnej podstruktury (ich optymalne rozwiązania składają się z pudełka b_t i największego stosu z pudełek $B - b_t$, na którym można ułożyć pudełko b_t)

Dowód Lematu 1

Jest to oczywisty wniosek wynikający z budowy najwyższego stosu. ■

Lemat 2

Jeśli dla wszystkich pudełek b_i zachodzi $w_i \geqslant l_i$ i pudełka są posortowane w ciąg $b_0, b_1, ..., b_{n-1}$, tak jak opisano odpowiednio w sekcjach 2.1 i 2.2 to dla pudełka b_i wszystkie pozostałe pudełka, które nie są mu równoważne, i na których można ułożyć b_i, znajdują się wśród pudełek o indeksach $0, ..., i-1$.

Dowód Lematu 2

Rozważmy w posortowanym ciągu parę pudełek b_h, b_i, w której pudełko b_i można ustawić na b_h. Możliwe są następujące przypadki:

- $w_h > w_i \wedge l_h \geqslant l_i$ - w posortowanym ciągu element z większą szerokością znajduje się wcześniej w ciągu, więc $h < i$
- $w_h = w_i \wedge l_h > l_i$ - w posortowanym ciągu element z równą szerokością i większą długością znajduje się wcześniej w ciągu, więc $h < i$
- $w_h = w_i \wedge l_h = l_i$ - sortowanie nie określa w tym przypadku relacji między h i i. Element z takimi samymi wymiarami może mieć indeks większe lub mniejsze od i

Podsumowanie tych przypadków kończy dowód lematu. ■

Dowód poprawności algorytmu opiera się na pokazaniu, że po wykonaniu algorytmu, wśród rozwiązań na listach $l_0, l_1, ..., l_{n-1}$ znajduje się najwyższy stos możliwy do ułożenia z pudełek $b_0, ..., b_{n-1}$

Oznaczamy posortowane pudełka w algorytmie $b_0, b_1, ..., b_{n-1}$. W listach $l_0, l_1, ..., l_{n-1}$ przechowywane są aktualnie znalezione rozwiązania podproblemów $0, ..., n-1$. Na początku w każdej liście l_i znajduje się tylko pudełko b_i.

Przed pierwszą iteracją algorytmu w liście l_0 znajduje się poprawne rozwiązanie problemu ułożenia jednego pudełka - b_0 - w najwyższy (jednoelementowy) stos.

W iteracji i-tej algorytmu, wybieramy z pudełek $b_0, ..., b_{i-1}$ takie, na których można położyć pudełko b_i. Z tych pudełek wybieramy jedno, które jest skojarzone z najdłuższą listą (najwyższym stosem), którą dołączamy do listy l_i. Na koniec iteracji i w liście l_i znajduje się najwyższy stos ułożony z pudełek $b_0, ..., b_i$, taki że pudełko b_i jest na szczycie tego stosu. Wtedy możliwe są dwa przypadki:

- **Przypadek 1** - wśród pudełek $b_{i+1}, ..., b_{n-1}$ nie ma pudełek równoważnych b_i. Wtedy z lematu 2 wynika, że wszystkie pudełka, na których można ułożyć b_i są już uwzględnione w rozwiązaniu, a wtedy z lematu 1 wynika, że l_i jest rozwiązaniem podproblemu i
- **Przypadek 2** - pudełka $b_h, b_{h+1}, ..., b_{i-1}$ i $b_{i+1}, ..., b_{j-1}, b_j$ $(h \leqslant i, i < j)$ są równoważne z b_i. Wtedy rozwiązanie w liście l_i nie uwzględnia wszystkich pudełek, na których można ułożyć b_i i nie jest rozwiązaniem podproblemu i. To nie wpływa jednak na poprawność działania algorytmu, ponieważ w trakcie analizy pudełka b_j zachodzi przypadek 1, w efekcie czego w liście l_j znajduje się rozwiązanie podproblemów $h, ..., i, ..., j$ (z dokładnością do permutacji równoważnych pudełek). Ponieważ algorytm w następnych krokach będzie zawsze wybierać najdłuższą z list $l_h, ..., l_j$ (czyli l_j), niekompletność rozwiązań w listach $l_h, ..., l_{j-1}$ nie wpływa negatywnie na wynik algorytmu

Po wykonaniu wszystkich pętli algorytmu, wśród list $l_0, ..., l_{n-1}$ znajdują się rozwiązania wszystkich podproblemów $0, ..., n-1$ (z dokładnością do permutacji pudełek równoważnych). W ostatnim kroku algorytm wybiera z tych rozwiązań najwyższy stos, który jest rozwiązaniem problemu.

4 Złożoność algorytmu

Zgodnie z opisem z sekcji 2, algorytm składa się z czterech etapów o następujących złożonościach:

1. Obracanie pudełek - wymaga przeglądnięcia wszystkich elementów listy i ewentualnej operacji ich obrócenia, która wymaga trzech przypisań. Optymistycznie wymaga to n operacji, pesymistycznie $4n$, więc złożoność pesymistyczna i optymistyczna tego procesu to $O(n)$
2. Sortowanie pudełek algorytmem Quick Sort (sekcja 2.2), które ma złożoność optymistyczną $O(nlog(n))$ a pesymistyczną $O(n^2)$
3. Budowanie najwyższych stosów pudełek - wymaga ono przeglądnięcia $n-1$ sekwencji pudełek, które mają długości $1, 2, ..., n-1$. Dla każdego przeglądane pudełka trzeba wykonać 1 do 3 porównań i ewentualnie operację połączenia list (w czasie stałym). Potrzebna do tego liczba operacji wynosi więc w optymistycznym przypadku $\frac{n^2-n}{2}$, a w pesymistycznym $2(n^2-n)$. To daje złożoność optymistyczna i pesymistyczną rzędu $O(n^2)$
4. Wybór najwyższego stosu - wymaga on przeglądnięcia całej listy pudełek, która ma n elementów. Złożoność tej operacji jest zawsze rzędu $O(n)$

Dominującymi procedurami w algorytmie są sortownie i budowanie stosów. W optymistycznym przypadku, kiedy te procedury zajmują możliwie najmniej czasu, algorytm ma złożoność optymistyczną rzędu $O(\frac{n^2-n}{2})$. Kiedy te procedury zajmują możliwie najwięcej czasu, algorytm ma złożoność pesymistyczną rzędu $O(n^2)$. To oznacza, że zarówno optymistyczna jak i pesymistyczna złożoność algorytmu są rzędu $O(n^2)$.

5 Wejście i wyjście algorytmu

Algorytm na wejściu przyjmuje listę pudełek. Pudełko definiuje się przez dwie dodatnie liczby całkowite w i l, które odpowiednio oznaczają szerokość i długość pudełka. W zadaniu powinno być przynajmniej jedno pudełko.

Aby uruchomić algorytm dla danego problemu można to zrobić na dwa sposoby:

- wpisać ręcznie w terminalu konfigurację,
- podać ścieżkę do pliku, w którym opisana jest konfiguracja.

Aby wpisać ręcznie konfigurację należy najpierw wpisać "!T", następnie należy wpisać liczbę pudełek - n, gdzie n jest dodatnią liczbą całkowitą. W następnej części należy wpisać w każdej linii wymiary każdego pudełka oddzielone spacją. Plik powinien zawierać tylko ostatnią opisaną część, tzn. bez liczby n.

Przykładowy plik zawierający 3 pudełka wygląda następująco:

2 1
15 1
4 4

Na wyjściu algorytm oddaje listę posortowanych pudełek, które są wypisywane na konsoli.