TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

ĐẠI HỌC QUỐC GIA TP HCM

KHOA KHOA HỌC MÁY TÍNH BÁO CÁO

ĐỒ ÁN
PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN KNAPSACK

Lớp: CS112.L11

GVHD: Phạm Nguyễn Trường An

Sinh viên thực hiện:

16521692 - Nguyễn Vĩnh Huy

Mục lục

Chương 1

0-1 Knapsack

1.1 Giới thiệu bài toán

Bài toán Knapsack, hay còn gọi là xếp ba lô, hay bài toán cái túi, là một bài toán *Tối ưu hóa tổ hợp*. Bài toán được đặt tên từ vấn đề chọn những gì quan trọng để có thể chứa vừa vào một cái túi (với giới hạn khối lượng) để mang theo trong một chuyến đi. Ở đây, ta cho rằng các đồ vật không thể bị chia nhỏ/xé nhỏ/tách nhỏ ra.

Các bài toán tương tự thường xuất hiện trong kinh doanh, toán tổ hợp, lý thuyết độ phức tạp tính toán và mật mã học.

1.2 Liên hệ thực tế

Với việc một sinh viên mới lên thành phố, thường hay về nhà vào mỗi cuối tuần, thì có thể áp dụng Knapsack vào để giải quyết các đồ vật vào ba lô để tổng giá trị của đồ vật mang về nhà là nhiều nhất.

1.3 Phát biểu bài toán

Cho một cái túi có thể chứa một khối lượng W, hai mảng số nguyên là val[0..n-1] và wt[0..n-1] lần lượt là giá trị và trọng lượng của đồ vật n. Tìm một tập hợp các đồ vật sao cho nhét vừa vào ba lô và tổng giá trị của các đồ vật mang đi là lớn nhất.

Input:

- W: Sức chứa của túi
- mảng val: giá trị của n đồ vật
- mảng wt: trọng lượng của n đồ vật

Output:

- max: Tổng giá trị tối đa mà túi mang đi được

Bài toán Knapsack là một trong những bài toán có độ phức tạp là Pseudo polynomial (tạm dịch: Giả đa thức).

1.4 Độ phức tạp Pseudo Polynomial

Độ phức tạp Pseudo polynomial là độ phức tạp mà thời gian chạy trong trường hợp xấu nhất (worst case time complexity) bị phụ thuộc vào giá trị số học(numeric value) của Input thay vì số lượng input (number of inputs).

Ví dụ: Xét bài toán đếm số lần xuất hiện của tất cả các phần tử trong một mảng số nguyên dương. Ta có thể cài đặt một phương pháp giả đa thức cho bài toán này. Trước tiên tìm giá trị lớn nhất trong mảng, sau đó lặp từ giá trị 1 đến giá trị lớn nhất này và đối với mỗi giá trị, tìm số lần xuất hiện của nó trong mảng.

Chương 2

Phương pháp thiết kế bài toán

2.1 Đệ quy

2.1.1 Giới thiệu phương pháp

Đệ quy xảy ra khi bên trong một khái niệm X có sử dụng chính khái niệm X. Trong lập trình, *Đệ quy* xảy ra khi một phương thức được viết tự gọi lại chính nó. Ví dụ: Mã giả của hàm tính Fibbonacci

```
Fibo(n)
if n <= 1 then
   return 1
else
   return Fibo(n-1) + Fibo(n-2)
```

Hai yếu tố cần để tạo thành một phương thức đệ quy:

- Điều kiện dừng: Xác định cụ thể quy luật của phương thức và tìm giá trị cụ thể cho đến khi thỏa mãn một điều kiện nhất định.
- Phương thức đệ quy: Phương thức đệ quy sẽ tự gọi lại chính nó cho đến khi nó trả về điều kiện dừng.

2.1.2 Mã giả

```
    knapSack(W, wt, val, n):
if n = 0 or W = 0
    return 0
if wt[n-1] > W
    return knapSack(W, wt, val, n-1)
else:
    return max(
        val[n-1] + knapSack(
            W-wt[n-1], wt, val, n-1),
        knapSack(W, wt, val, n-1))
```

2.1.3 Phân tích độ phức tạp bằng các phương pháp toán học

Ta có phương trình đệ quy:

$$\begin{cases} T(0) = 1 \\ T(n) = O(1) + 2T(n-1), n > 0 \end{cases} \tag{2.1}$$

Giải hệ phương trình này, ta được:
$T(n) = 3O(1) + 4T(n-2)$
$T(n) = 7O(1) + 8T(n-3)$
...
$T(n) = (2^{n-1} - 1)O(1) + 2^{n-1}T(1)$
$T(n) = (2^n - 1)O(1)$
$T(n) = O(2^n)$

2.1.4 Mã nguồn

```
def knapSack(W, wt, val, n):
    if n == 0 or W == 0:
        return 0
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:
        return max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))
```

2.1.5 Phát sinh intput/output để kiểm tra tính đúng đắn

Input của bài toán được phát sinh ngẫu nhiên bằng cách sử dụng hàm randrange có sẵn trong python. Input được phát sinh ngẫu nhiên, sau đó tính Output bằng thư viện ortools.algorithms . Mã nguồn phát sinh Input/Output:

```
import random
from ortools.algorithms import pywrapknapsack_solver

n = random.randrange(50,60)
W = random.randrange(40,50)
wt = []
```

```
val = []

for i in range (n):
    wt.append(random.randrange(50))
    val.append(random.randrange(50))

solver = pywrapknapsack_solver.KnapsackSolver(
    pywrapknapsack_solver.KnapsackSolver.
    KNAPSACK_MULTIDIMENSION_BRANCH_AND_BOUND_SOLVER,
       'KnapsackExample')

solver.Init(val, [wt], [W])
bestvalue = solver.Solve()
file = open("Input.txt","w")
file.write(str(n) + " " + str(W) + " " + str(bestvalue) +
   "\n")
for i in range (n):
    file.write(str(wt[i]) + " " + str(val[i])+ "\n")

print (val)
print (wt)
```

File Input.txt sau khi chạy code có dạng:

- Hàng 1: chứa n, W và kết quả tối ưu nhất
- n hàng tiếp theo: lần lượt chứa wt[0], val[0], wt[1], val[1]...
- wt[n], val[n]: là cân nặng và giá trị của đồ vật

2.1.6 Phân tích độ phức tạp bằng thực nghiệm

2.2 Quy hoạch động

2.2.1 Giới thiệu phương pháp

Quy hoạch động (Dynamic Programming) được phát triển bởi nhà toán học Richard Bellman từ thập niên 1950s. Thời đó Programming không có ý nghĩa là lập trình máy tính như hiện tại. Programming có nghĩa là "tính toán bằng cách lập bảng", hay còn gọi là quy hoạch. Quy hoạch động được dùng để giải các bài toán có dạng đệ quy. Các bước sử dụng Dynamic Programming:

- Tìm cách mô phỏng dạng thức "một lời giải" của bài toán.
- Tìm phương trình đệ quy để tính lời giải tối ưu.
- Sử dụng phương pháp đệ quy có nhớ (Top down with memoization) hoặc phương pháp tìm lời giải từ dưới lên bottom-up để code tính kết quả tối ưu.

2.2.2 Mã giả

```
 knapSack(W, wt, val, n)
K = [][]
for i = 0 to W + 1
   for j in range n + 1
      K[i][j] = 0
 for i = 0 to n + 1
     for w = 0 to W + 1
         if i == 0 or w == 0
             K[i][w] = 0
         elif wt[i-1] <= w
             K[i][w] = max(val[i-1]
                       + K[i-1][w-wt[i-1]],
                           K[i-1][w])
         else
             K[i][w] = K[i-1][w]
     return K[n][W]
```

2.2.3 Phân tích độ phức tạp bằng các phương pháp toán học

Thuật toán gồm 2 vòng for chạy lồng vào nhau => T(n) = O(n x W)

2.2.4 Mã nguồn

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]
                          + K[i-1][w-wt[i-1]],
                          K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
```

2.2.5 Phát sinh intput/output để kiểm tra tính đúng đắn

Input của bài toán được phát sinh ngẫu nhiên bằng cách sử dụng hàm randrange có sẵn trong python. Input được phát sinh ngẫu nhiên, sau đó tính Output bằng thư viện ortools.algorithms . Mã nguồn phát sinh Input/Output:

```
import random
from ortools.algorithms import pywrapknapsack_solver

n = random.randrange(50,60)
W = random.randrange(40,50)
wt = []
val = []

for i in range (n):
    wt.append(random.randrange(50))
    val.append(random.randrange(50))

solver = pywrapknapsack_solver.KnapsackSolver(
    pywrapknapsack_solver.KnapsackSolver.
    KNAPSACK_MULTIDIMENSION_BRANCH_AND_BOUND_SOLVER,
       'KnapsackExample')

solver.Init(val, [wt], [W])
bestvalue = solver.Solve()
file = open("Input.txt","w")
file.write(str(n) + " " + str(W) + " " + str(bestvalue) +
   "\n")
for i in range (n):
    file.write(str(wt[i]) + " " + str(val[i])+ "\n")

print (val)
print (wt)
```

File Input.txt sau khi chạy code có dạng:

- Hàng 1: chứa n, W và kết quả tối ưu nhất
- n hàng tiếp theo: lần lượt chứa wt[0], val[0], wt[1], val[1]...
- wt[n], val[n]: là cân nặng và giá trị của đồ vật

2.2.6 Phân tích độ phức tạp bằng thực nghiệm