Analýza a zložitosť algoritmov

Marián Kurčina

ID: 127211

xkurcinam@stuba.sk

Cvičenie: Utorok 10.00 - 12.00

Obsah

Úloha 1

Zadanie

1. Navrhnite a implementujte algoritmus na riešenie problému plánovania so stanovenými termínmi. Určte plán s maximálnym celkovým ziskom, pričom každý úkon má zisk, ktorý bude získaný iba vtedy, ak bude úkon naplánovaný pred jeho termínom.

Vstupy: počet prác a pole celých čísel deadline, indexované od 1 do n, kde deadline[i] je termín pre i-tu prácu. Pole je zoradené vo vzostupnom poradí podľa ziskov.

Výstupy:
Optimálna sekvencia pre úkony.

1.1 Zvážte nasledujúce práce, termíny a zisky a použite algoritmus pre plánovanie so stanovenými termínmi na maximalizovanie celkového zisku. Zobrazte optimálnu sekvenciu úkonov s maximálnym ziskom na obrazovke.

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

Opis riešenia

Na riešenie som použil algoritmus 4.4 z Foundations of Algoritmhms od Richard E. Neapolitan, tak ako to bolo v zadaní. Program funguje nasledovne:
Polia s indexami, deadline a ziskami sú zadané v poliach a je ich možné zmeniť. Po načítaní údajov sa vykoná max sort na zoradenie údajov (pri úlohe 1 sa nič nestane, avšak kód je kompatibilný s úlohou 1.1). Po zoradení údajov podľa ziskov vzostupne sa spustí funkcia schedule, ktorá funguje tak ako v pseudokód zo zadania.
Funkcia schedule má parametre n-počet prác, deadline[]-pole deadlinov, jobs_inx[]- indexy prác, job_profit[]- pole profitov prác.
Na začiatku funkcie sa zistí maximálny deadline a teda aj počet prác, ktoré sa môžu vykonať, tento údaj nám bude slúžiť na vytvorenie pola na výsledné poradie prác. Toto pole bude inicializované na 0. Potom prebieha priradzovanie prác do pola výsledného poradia prác, ktoré prebiieha nasledovne:

Algoritmus vyskúša pridať prácu na pozíciu deadlinu, tak ako by ju robil na poslednú chvíľu. Ak sa to nedá skúsi to postupne so všetkými predchádzajúcimi pozíciami. Ak sa dostane až na začiatok zoznamu prácu odmietne a pokračuje s nasledujúcou. Po prejdení všetkých prác v zozname program vypíše najlepší spôsob zoradenia prác a zisk.

Výstup pre vstup zo zadania

Vstup:

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

Výstup:

```
Best job sequence:
7
1
3
2
Total Profit: 170
```

Analýza zložitosti algoritmu

d = maximálny deadline (maximálne n)

max sort – **O(n^2)**

find_max – **O(n)**

schedule:

Inicializácia na 0 – **O(d)**
Priradenie práce – **O(n^2)**
Celková zložitosť – O(d+ n^2) → **O(n^2)**

Celková zložitosť bez sortu- O(n+d+n^2) → **O(n^2)**

Úloha 2

Zadanie

Zvážte plánovací postup v algoritme Scheduling with Deadlines (Algoritmus 4.4). Nech d je maximum z termínov pre n úloh. Upravte postup tak, aby pridával úlohu čo najneskôr do plánu, ktorý sa vytvára, ale nie neskôr ako jej termín. Urobte to inicializovaním d+1 disjunktných množín, ktoré obsahujú celé čísla 0, 1, ..., d. Nech small(S) je najmenší člen množiny S. Keď je úloha naplánovaná, nájdite množinu S obsahujúcu minimum jej termínu a n. Ak je small(S) = 0, úloha sa zamietne. Inak ju naplánujte v čase small(S) a zlúčte S s množinou obsahujúcou small(S) − 1. Predpokladajme, že používame Disjoint Set Data Structure III z Prílohy C, ukážte, že táto verzia je θ(nlgm), kde m je minimum z d a n.

Opis riešenia

Pri implementácii som použil pseudokód na strane 732 z Foundations of Algoritmhms od Richard E. Neapolitan.
V svojom kóde používam poupravené funkcie z pseudokódu, keďže som nebol schopný úplnej implementácie. Tak ako v pseudokóde, mám taktiež zadefinovanú štruktúru pre prácu s parametrom smallest a nachádza sa v nej aj ukazovateľ na rodiča. V štruktúre nemám spomenutý depth, lebo tento parameter nepotrebujem, keďže pri spájaní množín pracujem s koreňovými prvkami. Rodič prvku vždy ukazuje na prvok vyššie v hierarchii a teda sa dokážem dostať ku koreňu skupiny, ktorý obsahuje správnu informáciu smallest.
Funkcia merge funguje tak, že spojí dva stromy, tak, že koreň jedného stromu bude ukazovať na koreň toho druhého. Celá skupina bude mať rovnaký koreň a teda aj rovnakú hodnotu smallest.
Funkcia small sa jednoducho dostane na koreň stromu a nájde hodnotu smallest.

Na začiatku sa práce zoradia taka ko v úlohe 1 podľa ziskov zostupne. Potom nájdem maximálnu hodnotu deadline a vytvorím pole setov a pole kde budem uchovávať najlepšie riešenie. Potom nastane process plánovania prác, ktorý prejde všetky prvky a pokúsi sa ich naplánovať pomocou nájdenia hodnoty smallest. Hodnota smallest značí pozíciu, kde sa daná práca môže najneskôr vykonať. Ak funkcia vráti hodnotu menšiu ako 1, značí to, že na danú prácu už nie je miesto. Ak ale vrráti hodnotu väčšiu ako 0, prácu môžem vykonať a spojím miesto kde sa vykoná s predchádzajúcim setom. Po prejdení všetkých prác vypíšem najlepší plan.

Výstup pre vstup zo zadania

Vstup:

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

Výstup:

```
Best job sequence:
7
1
3
2
Total Profit: 170
```

Analýza zložitosti algoritmu

d = maximálny deadline (maximálne n)

max sort – **O(n^2)**

find_max – **O(n)**

Inicializovanie setov – **O(d)**
Inicializovanie poľa na plán – **O(d)**

Plánovanie:

small – pri ideálnom strome je zložitosť nájdenia koreňu stromu log c, pričom c je počet prvkov v strome → **O(log d)**
merge – **O(1)**
plánovanie pre n prvkov = **O(n*(log d))**

Celková zložitosť bez sortu: O(n+2d+n*(log d)) → **O(n*log d)**, toto sa dá zapísať ako **O(n*log m),** pričom m je minimum z d a n, keďže d je vždy menšie alebo rovné ako n.

Úloha 3

Zadanie

Použite greedy prístup na napísanie algoritmu, čo minimalizuje číslo počet pohybov záznamov pri probléme spájania n súborov. Použite dvojité spájanie, kde sa počas každého spájania spájajú dva súbory.
a: uveďte dva rôzne vstupy pre vašu implementáciu a analyzujte každý krok vášho algoritmu
b: analyzujte váš algoritmus a ukážte výsledky pomocou notácie poradia

Opis riešenia

Súbory sú reprezentované pomocou listu ich dĺžok, keďže toto je jediný parameter, ktorý nás zaujíma a process spájania nie je nutné implementovať. Na riešenie používam queue, v ktorom držím najprv dĺžky pôvodných súborov a neskôr dĺžky spojených súborov. Algoritmus vždy vyberie z queue 2 najmenšie záznamy, spojí ich dĺžky a pridá nový záznam do queue. Na konci zostane v queue jeden záznam, ktorý reprezentuje finálny súbor utvorený zo všetkých predom spojených súborov. Môj kód na queue pochádza z môjho minuloročného riešenia implementovania queue z predmetu DSA. Mám dve funkcie týkajúce sa queue a to get_min a add. Get_min vráti hodnotu najmenšieho prvku, teda prvku na mieste 0 a vyrovná váhy pomocou rodičov a detí, tak aby novým prvým prvkom na mieste 0 bol naozaj najmenším z queue. Ďalšou funkciou je add, táto funkcia pridá prvok na správne miesto v queue aby poradie zostalo vyvážené.
Tento algoritmus je greedy, pretože pri každom kroku sa rozhoduje na základe aktuálneho lokálne optimálneho riešenia.

Analýza algoritmu

Majme vstupný list dĺžok súborov : [1,2,3,4,5,6,7] a [1,2,15,1,10,8,2]
Najprv sa tieto hodnoty vložia do queue queue bude vyzerať nasledovne:

[1,2,3,4,5,6,7] a [1,1,2,2,8,10,15]

Za každým vytiahne 2 najmenšie prvky z queue, teda najprv to bude 1,2 a 1,1. Toto sčíta a pridá do queue. Queue bude vyzerať takto: [3,3,4,5,6,7] a [2,2,2,8,10,15].

Počas procesu spájania by to vyzeralo takto:

[3,3,4,5,6,7] →3 + 3→6→[4,5,6,6,7]
[4,5,6,6,7] →4 + 5→9→[6,6,7,9]
[6,6,7,9] →6 + 6→12→[7,9,12]
[7,9,12] →7 + 9→16→[12,16]
[12,16] →12 + 16→28→[**28**]

[2,2,2,8,10,15] →2 + 2→4→[2,4,8,10,15]
[2,4,8,10,15] →2 + 4→6→[6,8,10,15]
[6,8,10,15] →6 + 8→14→[10,14,15]
[10,14,15] →10 + 14→24→[15,24]
[15,24] →15 + 24→39→[**39**]

Výsledkom programu je dĺžka finálneho súboru, teda 28 a 39.

Analýza zložitosti algoritmu

add – **O(log n)**
get_min **– O(log n)**

Pridanie n prvkov do queue **– O(n*log n)**

Zlúčenie súborov (počet spojení je n-1) – O((n-1)*(3*log n)) → **O(n*log n)**

Úloha 4

Zadanie

Bez konštrukcie Huffmanovho stromu generujte Huffmanov kód pre danú množinu znakov.

Character	Frequency
A	45
B	13
C	12
D	16
E	9
F	5

a: použite kanonické Huffmanovo kódovanie
b: použite približné kódovanie na základe frekvencie
c: uveďte príklady kódovania a dekódovania a analyzujte oba prístupy pomocou notácie poradia

Opis riešenia

Na začiatku sa zoradia údaje podľa frekvencií od najväčšej po najmenšiu. NAsledovne si užívateľ vyberie spôsob počítania dĺžky kódu pre každé písmeno.
Typ a:
Pre každé písmeno sa vytvoria structy, info a groups, info bude obsahovať informácie o frekvenciách, kóde, dĺžke kódu a character. Groups budú obsahovať pole charakterov a súčet ich frekvencií. Toto bude slúžiť ako informácia pri spájaní písmen. Pri tomto spôsobe sa postupne spájajú charaktery do skupín, pričom sa vždy spoja dve najmenšie skupiny. Počet spojení pre každé písmeno je dĺžka kódu pre toto písmeno.
Môj process spájania prebieha podobne ako v úlohe 3, používam queue a vyberám z nej 2 skupiny, spojím ich, aktualizujem informácie a vložím novú skupinu naspäť do queue. Pri aktualizovaní informácií pre každé písmenon v 1. skupine a 2. Skupine zväčším dĺžku kódu o 1.
Po spojení všetkých skupín opäť zoradím informácie o písmenách aby bol naozaj od najväčšej frekvencie po najmenšiu, pričom ak majú 2 písmená rovnakú frekvenciu ale rozdielnu dĺžku kódu sa zoradia od najmenšej po najväčšiu (Mal som problém pri zozname frekvencií, kde bolo 3 a viac s najvyššou hodnotou). Ak je dĺžka kódu prvého prvku 1, túto hodnotu inkrementujem aby tento prvok nebol prefixom iného kódu. Tento zoznam frekvencií info budeem používať pri hľadaní kódov.
Typ b:
Pre každý prvok použijem vzorec $\lceil -\log_2 p \rceil$, pričom $p = \frac{frekvencia\ pre\ prvok}{súčet\ všetkých\ frekvencií}$, tento vzorec mi určí pre každý prvok dĺžku kódu, túto informáciu pridám do poľa info.

Keď už viem dĺžky kódov, prichádza kódovanie. Pre kódovanie mám nasledujúci spôsob:
Mám premenné c a temp, c je aktuálna dĺžka kódu a temp je desiatková hodnota, ktorú budem kódovať do binárky. Keď sa mi dĺžka kódu zmení oproti predchádzajúcemu písmenu, temp vynásobím *2, takže ide o posun doľava a zmením hodnotu aktuálnej dĺžky kódu c o 1. Nasledovne

hodnotu temp zakódujem do binárnej sústavy, toto je kódom písmena. Týmto spôsobom nikdy nie je jeden kód prefixom iného.

Po zistení kódovanie má užívateľ možnosť odkódovať string a zakódovať string.
Pri odkódovaní ide po stringu a ukladá si prečítané symboli do premennej, pričom po každom zápise číslice skontroluje, či sa string nezhoduje s niektorým z kódov, ak sa zhoduje, vypíše ho a vyprázdni premennú. Ak sa nezhoduje prečíta ďalší znak.
Pri kódovaní prečíta znak, nájde ho v zozname info a vypíše jeho kód, pokračuje na ďalší znak.

Príklady kódovania a dekódovania

Majme zakódované písmená zo zadania pomocou Canonical Huffman coding:
A=00, B=010, C=011, D=100, E=1010, F=1011

Príklady odkódovania:
000101001011010 → adcfd
1001010000000010 → ceaaad

Príklady zakódovania:
adcfd → 000101001011010
ceaaad → 1001010000000010

Majme zakódované písmená zo zadania pomocou Frequency-Based Approximation:
A=00, B=010, C=011, D=1000, E=1001, F=10100

Príklady odkódovania:
000101001011010 → adebd
1001010000000010 → edaaad

Príklady zakódovania:
adcfd → 000101001011010
ceaaad → 1001010000000010

Analýza zložitosti kódovania a dekódovania

m = dĺžka stringu
Kódovanie – m krát kontrola medzi maximálne n písmenami → **O(m*n)**
Dekódovanie – m krát kontrola medzi maximálne n písmenami → **O(m*n)**

Analýza zložitosti algoritmu

add – **O(log n)**
get_min **– O(log n)**

max sort – **O(n^2)**

Typ A - kanonické Huffmanovo kódovanie:
- vytvorenie a naplnenie polí info a groups – **O(n)**
- priradenie do queue – **O(log*n)**, priradenie všetkých prvkov – **O(n*log n)**
- Spájanie prvkov:
  - Získanie prvku z queue – **O(log n)**
  - Skopírovanie polí charakterov do jedného – maximálny počet je n → **O(n)**
  - Inkrementovanie dĺžky kódu u spojených písmen – maximálny početje n → **O(n^2)**
  - Pridanie do queue – **O(log n)**
  - Celková zložitosť – **O(n^2)**
  - Celková zložitosť pre n opakovaní– **O(n^3)**
- Min sort – **O(n^2)**
- Celková zložitosť – **O(n^3)**

Typ B – kódovanie na základe frekvencie:
- Suma frequencií – **O(n)**
- Priradenie informácií do pola info – **O(n)**

Kódovanie pre písmená:
- m = Maximálna dĺžka kódu
- Prevod do binárnej sústavy – m môže byť maximálne n → **O(n)**
- Zakódovanie všetkých n písmen – **O(n^2)**

Celková zložitosť - kanonické Huffmanovo kódovanie – O(n^2+ n^3+ n^2) → **O(n^3)**
Celková zložitosť – kódovanie na základe frekvencie – O(n^2+n+ n^2) → **O(n^2)**