Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 19 Bratislava

AZA - Practical implementation and analysis of an algorithm

Dominik Zaťovič
2023/24
Analýza a zložitosti algoritmov

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 19 Bratislava

Úloha č. 1

Zadanie:

Máme zadanú tabuľku jednotlivých prác, pričom každá ma pridelený zisk, ktorý získame jej vykonaním, a deadline, teda čas, v ktorom môžeme prácu vykonať. Našou úlohou je spraviť program, ktorý nam vyberie najlepšiu kombináciu prác, ktoré môžeme spraviť, aby sme mali čo najväčší zisk.

Pracovať budeme s touto tabuľkou:

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

Riešenie:

Na začiatku programu si spravím pole s jednotlivými prácami, ich deadlinami a ziskom. Následne toto pole usporiadam podľa zisku, teda od najväčšieho zisku po najmenší.

Následne zistím, aký je maximálny deadline (v tomto prípade 4), teda vieme že budeme pracovať s deadlinami od 1 po 4. (deadline x znamená, že ho môžem dať na deadline 1 až x).

Spravím si pole „result,“ ktoré ma veľkosť maximálneho deadline a naplním ho hodnotami -1, ktoré mi indikujú, že na týchto pozíciach ešte nebola priradená žiadna práca.

Dominik Zaťovič

Potom prechádzam do funkcie scheduleJobs(), ktorá robí následovné:

Prechádzam polom zoradených prác (od najväčšieho zisku po najmenší) a pozerám na pole „result“ a hľadám miesto, kam by som ho mohol priradiť, s tým že začínam na mieste deadlinu tejto práce a postupne idem k menšiemu deadlinu. Ak sa nájde prázdne miesto, tento job tam priradím.

Pre prípad našej tabuľky to bude vyzerať následovne:

1) Práca č. 3 s deadlinom 3 a profit 60 – priradí na deadline 3
2) Práca č. 7 s deadlinom 1 a profit 55 – priradí na deadline 1
3) Práca č. 6 s deadlinom 1 a profit 45 – nepriradí
4) Práca č. 1 s deadlinom 2 a profit 40 – priradí na deadline 2
5) Práca č. 4 s deadlinom 2 a profit 20 – nepriradí
6) Práca č. 2 s deadlinom 4 a profit 15 – priradí na deadline 4
7) Práca č. 5 s deadlinom 3 a profit 10 – nepriradí

Výsledok programu bude: 7 1 3 2

Total profit: 170

Časová a priestorová zložitosť:

std::sort -> O(n logn) – známa náročnosť pre sort v C++

maxDeadline -> O(n) – prechádzam každý prvok raz

scheduleJobs -> O(n*max_deadline) -> prvý for-loop prechádza každý job (n) a druhý for-loop sa vykonáva toľko krát, koľko je deadline jobu, teda najviac krát keď robíme pre job s maximálnym deadlinom. AK zobereme že deadline nie je väčši ako počet jobov, bude to n*n

Celková časová zložitosť je teda $O(n^2)$

Priestorová zložitosť je O(n), keďže máme polia schedule, kde sa nachádzajú počiatočné práce a pole result, ktoré obsahujé iba max_deadline prvkov

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 19 Bratislava

Úloha č. 2

Zadanie:

Máme maticu z úlohy č. 1. Upravte program tak, aby sa využili disjoint sety. Nech d je maximálny deadline a n je počet prác. Práca by sa mala priradiť najneskôr ako sa dá, ale nie neskôr ako je jej deadline. Inicializujeme d+1 disjoint setov, od 0 po d. Nech smallS je najmenší prvok zo setu S. Keď sa priraďuje práca, nájdi set S, ktorý obsahuje minimum jeho deadlinu a n. Ak smallS je rovný 0, nepríjmi prácu. Ak je rovný niečomu inému, naplánuj ho na smallS a spoj set S so setom small(S)-1.

Riešenie:

Spravil som si classu Disjoinset, ktorá obsahuje pointer na parenta (na začiatku sám na seba).

Mám maticu s jobmi, ich deadlinami a profitami. Zistím maximálny deadline zo všetkých jobov.

Vytvorím d+1 disjoint setov (d=max deadline).

Zoradím všetky joby od najviac profitového po najmenej.

Vo for-loope prechádzam každý job, následne pozerám na najskorší voľný deadline, na základe jeho deadlinu a momentálneho stavu disjoint setov.

Vyberem minimum z deadlinu tohto jobu a počtu jobov – ak je deadline napr. 4, tak minimum zo 4 a 7 je 4. Toto minimum pošlem do disjointSet.find, ktorý mám definovaný v classe DisjointSet, ktorý slúži na nájdenie parent setu.

Ak tento set nie je 0, tak ho tám napíšem a následne spojím tento set so setom o 1 menším.

Príklad:

1) Job 3 má deadline 3, pozrie sa na disjoint set 3, ten je voľný, dam ho tam a spojím so setom 2, teda oba sety budú mať parent 2
2) Job 7 má deadline 1, pozrie sa na disjoint set 1, je voľný, dam ho tam a spojím so setom 0, oba sety majú parent 0
3) Job 6 má deadline 1, pozrie sa na disjoint set 0 (ten má parent 0), takže ho tam nepridelí.
4) Job 1 má deadline 2, pozrie sa na disjoint set 2 (parent je 2), takže ho tam dá a spojí so setom o 1 menším, teda teraz už s 0 (sety 0,1,2 majú parent 0)
5) Job 4 má deadline 2, pozrie sa na disjoint set 2 (ten má parent 0), takže ho tam nepridelí.
6) Job 2 má deadline 4, pozrie sa na disjoint set 4 (ten má panet 4), takže ho tam pridelí a spojí sa so setom 3 (parent bude 0)
7) Disjoint sety majú všetky parent 0, takže už žiadny job nemôže byť pridelený

Časová a priestorová zložitosť:

Std::sort -> O(n logn) -> sorting algoritmus v C++ má takúto časovú náročnosť

For-loop -> O(n) -> prechádzam každý job a prideľujem mu miesto

Maximálna časová zložitosť -> O(n logn)

Maximálna priestorová zložitosť je O(n) -> záleží od maximálneho deadlinu, čo znamená O(d), avšak bereme v úvahu že max deadline nie je väčší, ako počet jobov.

Porovnanie úlohy 1 a 2:

Zistili sme, že použitie disjoint-setov zmenší časovú zložitosť a taktiež je menšia šanca pre nastanie chyby v programe.

Disjoint-sety sú všeobecne najlepšie pre použitie pri tomto typu zadania, a to z nejakého dôvodu. Sú síce komplexnejšie na spravenie a porozumenie, ale vo výsledku sú lepšie.

Úloha č. 3a

Zadanie:

Predpokladajme, že priradíme n ľudí k n prácam. Nech Cij je cena priradenia i-teho človeka k j-tej práci. Máme teda napr. 3 ľudí a 3 práce, pričom priradenie každého človeka k práci stojí nejakú sumu. Chceme nájsť také priradenie, aby nás to vyšlo čo najmenej.

V úlohe a) máme spraviť program s využitím greedy approach.

Uvažujme tabuľku:

	J1	J2	J3
P1	10	5	5
P2	2	4	10
P3	5	1	7

Greedy approach pôjde postupne od prvého človeka po posledného, pričom každému priradí prácu s najmenšiou cenou, ktorá ešte nebola nikomu pridelená.

Riešenie:

Na začiatku programu si spravím maticu cien costMatrix, v ktorej sa nachádzajú hodnoty z tabuľky. Potom prechádzam do funkcie greedyApproach, kam pošlem túto maticu.

Vo funkcii si spravím pole usedPositions, ktoré mi bude slúžiť na zapisovanie prác, ktoré už boli niekomu pridelené, a teda nebudem ich brať do úvahy pri ďalšiom prideľovaní.

Následne prechádzam riadky v matici a hľadám najmenšiu hodnotu v tomto riadku a pozerám, či sa táto práca už nenachádza v usedPositions (používam funkciu notPresent, ktorá mi prejde cez pole usedPositions a pozerá, či sa tám nachádza hodnota s pozíciu terajšieho jobu).

Ak nájdem voľnú prácu s najmenšiou cenou, pridám ju do usedPositions a idem do ďalšieho riadku (ďalší človek).

Funkcia mi vráti pole usedPositions, ktoré obsahuje vybrané pozície v každom riadku. Toto pole následne pošlem to funkcie printTable(), ktoré mi do konzole vykreslí tabuľku n krát n s hodnotami 0 a 1, pričom 1 predstavuje prácu pridelenú človeku, ktorému patrí „riadok.“

Časová a priestorová zložitosť:

greedyApproach() -> O(n^2) -> prechádza každým riadkom a v ňom každou hodnotou, teda n*n

printTable() -> O(n) – obsahuje 2 for loopy, jeden ide n-krát (prechádza prvky v usedPositions) a druhý printuje riadky, teda tiež n-krát

Celková časová zložitosť -> O(n^2)

Celková priestorová zložitosť -> O(n^2) -> narábam s maticami n*n

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 19 Bratislava

Úloha č. 3b

Zadanie:

Zadanie je rovnaké ako v úlohe 3a, avšak v úlohe b) máme spraviť program s využitím dynamic programming approach.

Ja konkrétne som si vybral maďarský algoritmus (Hungarian algorithm)

Tento algoritmus obsahuje niekoľko krokov potrebných k dostaniu sa k optimálnemu výsledku, každý krok opíšem nižšie kde budem opisovať fungovanie môjho kódu.

Hlavný rozdiel oproti greedy approachu je ten, že pri greedy som šiel postupne, hľadal najmenšiu cenu nepridelených jobov, čo však nie vždy dalo ten najlepší výsledok, napr. tretí človek mohol mať pre job 2 cenu 5 a prvý 10, avšak keďže tento job bol pridelený prvému, nemohol byť tretiemu, čo by cenu znížilo.

Výsledok hungarian algoritmu nám dá ten najlepší vyber za najnižšiu cenu, avšak výmenou za náročnosť programu

Riešenie:

Môj program pozostáva z veľa funkcií, pričom 5 z nich sú kroky hungarian algoritmu, ktoré volám v jednej veľkej funkcii.

Hlavná funkcia je hungarianAlgorithm(), do ktorej vstupuje naša začiatočná matica.

Dôležité dodať, že každá funkcia, ktorá sa tu volá, nám vráti novú, upravenú maticu.

Ako prvá funkcia sa zavolá step1(), ktorá nájde najmenšiu hodnotu v každom riadku a odpočíta ju od každej hodnoty v tomto riadku, čo nám zaistí aspoň jednu nulu v každom riadku.

Následuje funkcia step2(), ktorá nájde najmenšiu hodnotu v každom stĺpci a odpočíta ju od každej hodnoty v tomto stĺpci, to nám zaistí aspoň jednu nulu v každom stĺpci.

Funkcia step3() je celkom komplikovane napísaná, keďže som nevedel ako inak to spraviť. Podstata tejto funkcie je nakresliť čiary cez riadky/stĺpce tak, aby ich bolo čo najmenej a pokrývali všetky 0 v matici.

Túto funkciu podrobnejšie opíšem neskôr, keďže môže byť komplikovaná na porozumenie.

Po step3() sa pozerám, či počet čiar je rovný n. Ak áno, matica už je riešiteľná a prechádzam do funkcie stepfinal(), ak je počet čiar menší, prechádzam na step4().

Step4() nám hľadá najmenší prvok, cez ktorý neprechádza čiara. Následne toto číslo odpočíta od všetkých riadkov, cez ktoré nie je čiara a následne pripočíta ku všetkým stĺpcom, cez ktoré ide čiara. Následne sa vraciame ku kroku 3.

Stepfinal() hľadá z matice taký výber núl, aby bola práve jedna pre riadok a stĺpec, čo nám ukáže priradenie prác k ľuďom. Túto funkciu taktiež podrobnejšie popíšem nižšie.

Funkcie:

Step3()-> Prvý for-loop slúži na zistenie, koľko núl sa nachádza v každom stĺpci a riadku. Chceme totižto, aby naša čiara prechádzala čo najviac nulami naraz.

Tieto údaje sú uložené v poliach coveredRows a coveredCols.

Následuje while cyklus, kde zistím najväčšie čislo z týchto 2 polí.

Ak sa nerovnajú, tak zistím, či bolo z coveredRows alebo Cols.

Povedzme že najviac ich je v riadku č. 1. Spravím cez neho čiaru, počet núl v coveredRows na pozícií 0 (prvý riadok) zmením na 0, keďže som ich pokryl, a v coveredCols od každého indexu odpočítam 1, keďže máme o 1 nulu menej v každom stĺpci.

Ak sa maximá rovnajú, tak si vyberiem či spravím čiaru cez riadok alebo stĺpec a postupujem podľa kroku vyššie.

Ak sa obe maximá rovnajú 0, znamená to, že všetky nuly boli pokryté čiarami.

Túto funkciu volám 2x z hlavnej funkcie, raz kde sa vyberá riadok a raz kde sa vyberá stĺpec v prípade, že maximá sú rovnaké, a zistim pri ktorom bolo treba menej čiar (Slúži na edge casy). To kde bolo treba menej čiar zavolám znova a vráti mi to počet čiar a cez ktoré riadky/stĺpce prechádzajú.

Stepfinal() -> mam 3 polia (firstRow, secondRow, thirdRow), jedno pre každý riadok, zistím na akých pozíciach sa nachádzajú nuly – prvý for loop

V ďalšom for-loope prechádzam cez všetky pozície v prvom riadku, nájdem pozíciu, kde sa nachádza 0, potom prechádzam pozície v druhom riadku, hľadám pozíciu, kde sa nachádza 0, pričom musí byť na inom indexe ako v prvom riadku (iný stĺpec), a následne prechádzam tretí riadok, nájdem 0 ale musí byť iný index ako v prvom a druhom riadku.

Áno, tento kód funguje len na matice 3x3, určite by sa to dalo spraviť všeobecne, ale najprv som nevedel spraviť ani toto, takže som rád že mi to aspoň nejak funguje a hlavne správne.

Časová a priestorová zložitosť:

Step1() -> O(n^2) -> prechádza každý prvok v matici, hľadá minimum v riadku

Step2() -> O(n^2) -> prechádza každý prvok v matici, hľadá minimum v stĺpci

Step3() -> O(n^2) -> prvý for-loop prechádza každý prvok v matici, while cyklus ide maximálne n-krát

Step4() -> O(n^2) -> prechádza každý prvok v matici, hľadá minimum z nepokrytých prvkov

Stepfinal() -> O(n^3) -> obsahuje trojitý for-loop, ktorý prechádza n-prvkov.

hungarianAlgorithm() -> O(n^4) -> obsahuje while-cyklus, ktorý sa vykoná maximálne n-krát, a v ňom sa nachádza funkcia step3, ktorá ma O(n^3)

Maximálna časová zložitosť -> O(n^4)

Maximálna priestorová zložitosť -> O(n^2) -> pracujeme s maticami n*n

Porovnanie 3a a 3b

Zistili sme, že greedy approach má časovú zložitosť O(n^2), avšak jeho výsledok nie je optimálny, keďže v málo prípadoch nájde ten najlepší, ktorý hľadáme.

Na druhej strane hungarian algorithm slúži na nájdene tohto optimálneho výsledku, avšak za cenu vyššiej časovej zložitosti. Hungarian algorithm by mal mať časovú zložitosť O(n^3), avšak v mojom programe ju má až O(n^4).

Záver:

Zistili sme, že čím je kód komplexnejší a zložitejší na naprogramovanie, tým je lepší, presnejší a rýchlejší v praxi. Čo sa týka hungarian algoritmu v úlohe 3b, moja zložitosť je väčšia, než by mala byť teoreticky. Môže za to pravdepodobne to, že kód nie je optimalizovaný, je komplikovaný a je napísaný tak, aby hlavne fungoval, nie aby fungoval čo najlepšie. Nemám na to dostatok znalostí ani času, aby som ho optimalizoval.