Peter Brenkus, ID: 120762

AZA
Praktické zadanie a analýza algoritmov
2023 - 24

Cvičiaci: Ing. Ján Bartoš
Čas cvičení: Utorky 16:00 – 17-50

Table of Contents

Úloha 1

Vytvoriť a implementovať algoritmus pre plánovanie prác s deadlinami podľa pseudokódu algoritmu 4.4 z knihy „Neapolitan – Foundations of Algorithms“.

Scheduling with Deadlines

Problem: Determine the schedule with maximum total profit given that each job has a profit that will be obtained only if the job is scheduled by its deadline.

Inputs: n, the number of jobs, and array of integers *deadline*, indexed from 1 to n, where *deadline*[i] is the deadline for the ith job. The array has been sorted in nonincreasing order according to the profits associated with the jobs.

Outputs: an optimal sequence J for the jobs.

```
void schedule (int n,
               const int deadline[],
               sequence_of_integer& J)
{
  index i;
  sequence_of_integer K;

  J = [1];
  for (i = 2; i <= n; i++){
     K = J with i added according to nondecreasing values of deadline[i];
     if (K is feasible)
        J = K;
  }
}
```

Máme zadanú tabuľku jednotlivých prác, pričom každá ma pridelený zisk, ktorý získame jej vykonaním, a deadline, teda čas, v ktorom môžeme prácu vykonať:

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

Implementácia

Zdrojový kód som implementoval v samostatnom súbore Z1_1.cpp.

Na začiatku programu si spravím pole s jednotlivými prácami, ich deadlinami a ziskom. Následne toto pole usporiadam podľa zisku od najväčšieho zisku po najmenší. Funkcia pre zoraďovanie zároveň vytvorí a vráti pole deadlinov takto zoradených prác. Toto pole je potrebné pre návrh plánovacieho algoritmu podľa pseudokódu zo zadania.

```
// vytvorim si pole jobov
job arr[] = {
        [0]: { .id: 1, .deadline: 2,  .profit: 40},
        [1]: { .id: 2, .deadline: 4,  .profit: 15},
        [2]: { .id: 3, .deadline: 3,  .profit: 60},
        [3]: { .id: 4, .deadline: 2,  .profit: 20},
        [4]: { .id: 5, .deadline: 3,  .profit: 10},
        [5]: { .id: 6, .deadline: 1,  .profit: 45},
        [6]: { .id: 7, .deadline: 1, .profit: 55}
};

// zoradim joby podla profitu, vytvorim pole deadlinov pre planovaci algoritmus
int* deadlines = sort(arr);

// vypisem joby zoradene podla profitu
cout << "Array of jobs sorted by profit: " << endl;
for(auto & i :job&  : arr){
    cout << "Job id: " << i.id << ", Deadline: " << i.deadline << ", Profit: " <<
}
```

Potom prechádzam do funkcie pre plánovanie. Funkcia pracuje presne podľa pseudokódu zo zadania. Vytvorí si dve premenné pre sekvenciu plánovania, do finálnej pridá prvú prácu a do druhej postupne v cykle pridáva ďalšie a volá funkciu na vyhodnotenie prijateľnosti sekvencie. Ak je nová sekvencia prijateľná, finálna sekvencia sa rozšíri, ak nie, zostáva rovnaká. Na koniec sa vracia výsledná sekvencia naplánovaných prác.

```
// vypise optimalnu sekvenciu naplanovania jobov
string schedule(int* deadlines){

    // premenim pole deadlinov na string
    string Deadlines;
    for (int i =0; i < n; i++){
        Deadlines += to_string( val: deadlines[i]);
    }

    // pridam prvy job do sekvencie
    string sequence_final = "1", sequence_temp;

    // postupne pridavam dalsie joby do sekvencie, ak je to mozne
    for (int i = 2; i <= n; i++){
        sequence_temp = sequence_final + to_string( val: i);
        sequence_final = isFeasible( sequence: sequence_temp, Deadlines);
    }
    return sequence_final;
}
```

Funkcia pre vyhodnotenie prijateľnosti sekvencie funguje tak, že prejde všetky možné permutácie danej sekvencie a ak nájde také usporiadanie, ktoré vyhovuje podmienkam deadlinov, vracia rozšírenú sekvenciu. Ak vhodné usporiadanie neexistuje, vracia pôvodnú sekvenciu bez novej práce.

```cpp
// najde splnitelnu sekvenciu, ak nenajde, vrati sekvenciu bez noveho jobu
string isFeasible(string sequence, string deadlines){

    // vytvorim si kopiu sekvencie, ktoru budem modifikovat
    string sequence2 = sequence;
    sequence2.erase( pos: sequence2.length() -1);

    // prejdem vsetky permutacie sekvencie s novym jobom
    do{
        string output;

        // prejdem vsetky joby v momentalnej permutacii
        for(int i = 0; i < sequence.length(); i++){
            char s = sequence[i];
            int d = deadlines[((s - '0') - 1)] - '0';

            // ak je deadline vacsi ako deadline jobu, pridam ho do vystupu
            if(d >= (i + 1)){
                output += sequence[i];
            }
        }

        // ak je dlzka vystupu rovna dlzke sekvencie, nasiel som splnitelnu sekvenciu
        if(output.length() == sequence.length()){
            return output;
        }
    } while(next_permutation( first: sequence.begin(),  last: sequence.end()));

    // ak som nenasiel splnitelnu sekvenciu, vratim povodnu sekvenciu bez noveho jobu
    return sequence2;
}
```

Pre prípad tabuľky zo zadania vyzerá proces nasledovne:

(sekvencia uchováva poradie prác v ich poli, nie ich id)

1) Práca č. 3 s deadlinom 3 je priradená, sekvencia = „1“
2) Práca č. 7 s deadlinom 1 je priradená, sekvencia = „21“
3) Práca č. 6 s deadlinom 1 nie je priradená, sekvencia = „21“
4) Práca č. 1 s deadlinom 2 je priradená, sekvencia = „241“
5) Práca č. 4 s deadlinom 2 nie je priradená, sekvencia = „241“
6) Práca č. 2 s deadlinom 4 je priradená, sekvencia = „2416“
7) Práca č. 5 s deadlinom 3 nie je priradená, sekvencia = „2416“

Nakoniec sa daná sekvencia prepíše na ID priradených prác: „7132“ a vypíše sa do konzoly:

```
The optimal sequence of jobs is:
Job id: 7
Job id: 1
Job id: 3
Job id: 2
```

Peter Brenkus, ID: 120762

Analýza zložitosti

1) Časová zložitosť insert sortu je známa: $O(n^2)$.
2) Časová zložitosť funkcie pre vyhodnotenie prijateľnosti je O(n!), lebo funkcia v najhoršom prípade musí prejsť všetky permutácie sekvencie.
3) Časová zložitosť plánovacieho algoritmu je O(n), lebo funkcia vo for cykle prejde každou jednou prácou.

Nás zaujíma zložitosť samotného plánovacieho algoritmu, usporadúvanie je vedľajšie.

Celková časová zložitosť samotného plánovacieho algoritmu je teda n * n! = O(n!).

1) Priestorová zložitosť poľa pre práce je O(n)
2) Priestorová zložitosť reťazcov deadlinov a sekvencií je tiež O(n)

Celková priestorová zložitosť je teda n + n + n + n = O(n).

Úloha 2

Máme tabuľku prác z úlohy č. 1. Upravte program z prvej úlohy tak, že prácu bude pridávať na jej posledný možný termín pomocou dátovej štruktúry disjunktných množín. 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 small(S) je najmenší prvok zo setu S. Keď sa priraďuje práca, nájdi set S, ktorý obsahuje minimum jeho deadlinu a n. Ak small(S) je 0, neprijmi prácu. Ak je rovný niečomu inému, naplánuj ho na čas small(S) a spoj set S so setom small(S)-1.

Implementácia

Dátovú štruktúru disjunktných množín som si implementoval v samostatnom súbore disjoint_sets_v2.cpp. Jednotlivé množiny sú reprezentované uzlami binárneho stromu, univerzum množín je ukazovateľ na uzly s veľkosťou n.

Úlohu 2 som implementoval v samostatnom súbore Z1_2_v2.cpp, do ktorého importujem súbor s funkciami pre disjuktné množiny. Keďže v úlohe ide o úpravu algoritmu z prvej úlohy, väčšina programu je rovnaká a bola popísaná vyššie. Rozdiel je iba vo funkcii pre plánovanie prác.

V nej sa na začiatku nájde maximálny deadline „d“ zo všetkých prác a vytvorí sa univerzum d + 1 disjunktných množín s hodnotami od 0 po d. Následne vo for cykle prechádzam každú prácu a vyberiem minimum z deadlinu tejto práce a celkového počtu prác – ak je deadline napr. 4, tak minimum zo 4 a 7 je 4. Toto minimum potom vyhľadám v univerze pomocou funkcie find(), ktorá vráti ukazovateľ na množinu, v ktorej sa daná hodnota nachádza. V danej množine potom nájdem najmenší prvok pomocou funkcie small(). Ak ten nie je 0, tak nájdem množinu s prvkom o 1 menším ako je momentálny deadline opäť pomocou funkcie find(). Tieto dve množiny spojím pomocou funkcie merge() a aktuálnu prácu si uložím do premennej „scheduled_jobs“ ktorá zaznamenáva naplánované práce.

Na konci tejto funkcie sa potom vypíšu naplánované práce do konzoly.

Peter Brenkus, ID: 120762

```
// vypise optimalne naplanovanie jobov
void schedule(job* jobs){
    // najdem maximalny deadline - treba na vytvorenie mnoziny U
    int max_deadline = jobs[0].deadline;
    for (int i = 1; i < n; i++){...}
    // pomocna funkcia na vytvorenie universe
    string U = make_universe( n: max_deadline);
    // sets predstavuje moj universe, v ktorom su jednoprvkove mnoziny
    string* sets = new string[U.length()];
    for (int i = 0; i < U.length(); i++) {
        sets[i] = make_set( x: U[i]);
    }
    // mnozina jobov, ktore budu vysledne naplanovane
    job* scheduled_jobs = new job[max_deadline];

    // planovaci algoritmus podla zadania 2
    for (int i = 0; i < n; i++){
        int min_deadline = min(jobs[i].deadline, max_deadline);
        string* p1 = find( x: to_string( val: min_deadline), sets);
        int smallest = small( set: p1);
        if (smallest != 0){
            string* p2 = find( x: to_string( val: min_deadline - 1), sets);
            sets = merge( p1: p2,  p2: p1, sets);
            scheduled_jobs[min_deadline] = jobs[i];
        }
    }
    // vypis naplanovanych jobov
    cout << "Scheduled jobs: " << endl;
    for (int i = 1; i <= max_deadline; i++){
        cout << "Job id: " << scheduled_jobs[i].id << ", Deadline: " << sc
    }
}
```

Pre prípad tabuľky zo zadania vyzerá proces nasledovne:

Na začitaku je U = [0], [1], [2], [3], [4] *(reálne ukazovatele vyzerajú inak, stromy tu kresliť neviem)*

1) Práca č.3 má deadline 3, nájde sa množina, kde je č.3, keďže v nej nie je 0, práca sa naplánuje a spoja sa množiny 2 a 3. U = [0], [1], [2 -> 3], [4]
2) Práca č.7 má deadline 1, nájde sa množina, kde je č.1, keďže v nej nie je 0, práca sa naplánuje a spoja sa množiny 0 a 1. U = [0 -> 1], [2 -> 3], [4]
3) Práca č.6 má deadline 1, nájde sa množina, kde je č.1, keďže v nej je 0, práca sa nenaplánuje. U = [0 -> 1], [2 -> 3], [4]
4) Práca č.1 má deadline 2, nájde sa množina, kde je č.2, keďže v nej nie je 0, práca sa naplánuje a spoja sa množiny 01 a 23. U = [0 -> 1 -> 2 -> 3], [4]
5) Práca č.4 má deadline 2, nájde sa množina, kde je č.2, keďže v nej je 0, práca sa nenaplánuje.

   U = [0 -> 1 -> 2 -> 3], [4]

Cvičiaci: Ing. Ján Bartoš
Čas cvičení: Utorky 16:00 – 17-50

6) Práca č.2 má deadline 4, nájde sa množina, kde je č.4, keďže v nej nie je 0, práca sa naplánuje a spoja sa množiny 0123 a 4. U = [0 -> 1 -> 2 -> 3 -> 4]
7) Práca č.5 má deadline 3, nájde sa množina, kde je č.3, keďže v nej je 0, práca sa nenaplánuje. U = [0 -> 1 -> 2 -> 3 -> 4]

Nakoniec sa sekvencia naplánovaných prác vypíše do konzole:

```
Scheduled jobs:
Job id: 7, Deadline: 1, Profit: 55
Job id: 1, Deadline: 2, Profit: 40
Job id: 3, Deadline: 3, Profit: 60
Job id: 2, Deadline: 4, Profit: 15
```

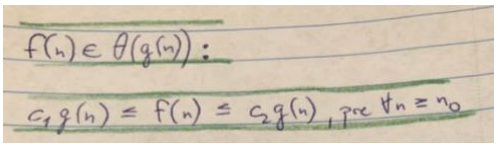
Analýza zložitosti

1) Časová zložitosť insert sortu je známa: $O(n^2)$.
2) Časová zložitosť for cyklu v plánovacom algoritme je $O(n)$, lebo prechádza každú prácu raz.
3) Časová zložitosť vyhľadávania a spájania v binárnych stromoch je známa: $O(\log(m))$, kde m reprezentuje hĺbku stromu, v našom prípade maximálny deadline.

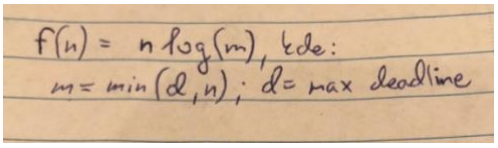
Nás zaujíma zložitosť samotného plánovacieho algoritmu, usporadúvanie je vedľajšie.

Celková časová zložitosť samotného plánovacieho algoritmu je preto $O(n*\log(m))$. Úlohou v zadaní je ale dokázať že tento algoritmus patrí do $\Theta(n*\log(m))$:

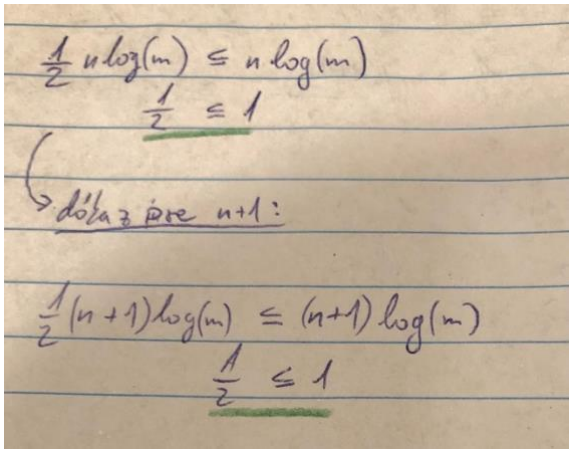
Definícia: Funkcia patrí do zložitostnej triedy $\Theta(g(n))$ keď je aj zhora aj zdola ohraničená nejakým konštantným násobkom $g(n)$ pre všetky n >= n_0.

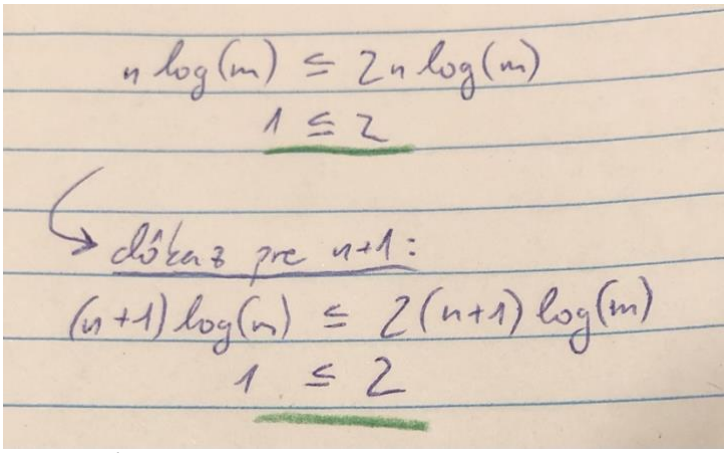


Dôkaz: Naša funkcia je $n*\log(m)$, keďže algoritmus sa opakuje n-krát a v každej iterácií vykoná log(m) opakovaní na vyhľadávanie a spájanie v stromoch.



Nech naše $c_1 = ½$, $c_2 = 2$ a $n_0 = 1$. Potom dôkaz vyzerá nasledovne:





Záver: Keďže sme našli také c_1 , c_2 a n_0 , ktoré spĺňajú nerovnicu z definície, dokázali sme že algoritmus naozaj patrí do $\Theta(n*\log(m))$. To znamená že je o dosť efektívnejší než algoritmus z úlohy 1. Hlavný dôvod pre túto skutočnosť je implementácia binárnych stromov, ktorých vyhľadávacia zložitosť je $O(\log(n))$, kde n je hĺbka stromu. To je oveľa lepšie ako $O(n!)$ pri riešení permutácií v úlohe 1.

Cvičiaci: Ing. Ján Bartoš
Čas cvičení: Utorky 16:00 – 17-50

1) Priestorová zložitosť premenných pre práce je O(n), lebo je priamo úmerná počtu prác n.
2) Priestorová zložitosť pemenných pre disjunktné množiny je O(d), lebo ich je toľko koľko je maximálny deadline d.

Celková priestorová zložitosť algoritmu je preto n + d = O(n).

Úloha 3

Predpokladajme, že priradíme n ľudí k n prácam. Nech C_{ij} 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.

a) Implementovať algoritmus s lačným prístupom k problému.
b) Implementovať algoritmus, ktorý bude používať dynamické programovanie.

Príklad matice cien:

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

Implementácia lačného prístupu

Tento algoritmus pôjde postupne od prvého človeka po posledného, pričom každému priradí prácu s najmenšou cenou, ktorá ešte nebola nikomu priradená.

Na začiatku programu si spravím maticu cien cost_matrix, v ktorej sa nachádzajú hodnoty z tabuľky. Tú potom posielam do funkcie greedy_algorithm().

```
int cost_matrix[n][n] = {
        [0]: { [0]: 10,  [1]: 5,  [2]: 5},
        [1]: { [0]: 2,  [1]: 4,  [2]: 10},
        [2]: { [0]: 5,  [1]: 1,  [2]: 7},
};

int** cost_matrix_copy = copy_matrix( cost_matrix: &cost_matrix[0][0]);

cout << "Greedy algorithm assignments:" << endl;
greedy_algorithm( cost_matrix: &cost_matrix[0][0]);
```

V nej si vytvorím pole pre výsledné priradenie a potom prechádzam po jednom ľudí. Každému priradím prácu s najmenšou cenou a následne vynulujem danú prácu v matici, aby som už rovnakú prácu neprideľoval nikomu inému. Nakoniec priradenie vypíšem.

```cpp
void greedy_algorithm(int* cost_matrix){

    //vytvorim si pole assignments, ktore bude obsahovat indexy priradenych jobov
    int* assignments = new int[n];
    for (int i = 0; i < n; i++) {

        //najdem najmensiu cenu v riadku a priradim ju do pola assignments - greedy vlastnost
        int min_cost = INF;
        for (int j = 0; j < n; j++) {
            if (*(cost_matrix + i * n + j) != 0 && *(cost_matrix + i * n + j) <= min_cost) {
                assignments[i] = j;
                min_cost = *(cost_matrix + i * n + j);
            }
        }

        //vynulujem stlpec, v ktorom sa nachadza priradeny job
        for (int k = 0; k < n; k++) {
            *(cost_matrix + k * n + assignments[i]) = 0;
        }

        //vypisem vysledok
        cout << "The " << i + 1 << ". person is assigned the " << assignments[i] + 1 << ". job." << endl;
    }
}
```

Lačný algoritmus je pomerne jednoduchý, no negarantuje vždy optimálny výsledok. V tomto prípade optimálne priradenie našiel, ale v iných to tak byť nemusí.

```
Greedy algorithm assignments:
The 1. person is assigned the 3. job.
The 2. person is assigned the 1. job.
The 3. person is assigned the 2. job.
```

Implementácia dynamického programovania

V tomto algoritme sa najprv vytvorí rovnako veľká matica „dp“ pre udržiavanie medzivýpočtov, ktoré sú základom dynamického programovania.

Do prvého riadku sa uložia ceny prvého človeka tak ako sú. Každý ďalší riadok sa postupne naplní cenami daného človeka pre danú prácu s ohľadom na minimálnu cenu z predošlého riadku.

Po naplnení novej matice sa nájde optimálne pridelenie smerom *zdola nahor.* Nájde sa minimálna cena z posledného riadku a daná práca sa zapíše poslednej osobe. Cena danej práce pre ostatné osoby sa potom zmení na maximálnu hodnotu akú môže typ „int“ nadobúdať. Tým sa zabezpečí, že daná práca sa už nepridelí nikomu inému.

```cpp
void dynamic_programming_algorithm(int* cost_matrix) {

    // vytvorim maticu dp a naplnim ju nekonecnymi hodnotami
    int dp[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dp[i][j] = INF;
        }
    }

    // naplnim prvy riadok hodnotami z cost_matrix
    for (int j = 0; j < n; j++) {
        dp[0][j] = *(cost_matrix + 0 * n + j);
    }

    // naplnim zvysne riadky suctom cien z cost_matrix a min hodnotou z predchadzajuceho riadku
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                if (k != j) {
                    dp[i][j] = min(dp[i][j], *(cost_matrix + i * n + j) + dp[i - 1][k]);
                }
            }
        }
    }
```

Cvičiaci: Ing. Ján Bartoš
Čas cvičení: Utorky 16:00 – 17-50

```cpp
// v poslednom riadku najdem najmensiu hodnotu a jej index zapisem do pola assignments
int* assignments = new int[n];
int min_index = static_cast<int>(min_element( first: dp[n - 1], last: dp[n - 1] + n) - dp[n - 1]);
assignments[n - 1] = min_index;
cover_columns( dp: &dp[0][0], index: min_index);

// dany job uz je priradeny, takze v matici dp prekryjem stlpec, v ktorom sa nachadza nekonecnymi hodnotami
for (int i = n - 2; i >= 0; i--) {
    min_index = static_cast<int>(min_element( first: dp[i], last: dp[i] + n) - dp[i]);
    assignments[i] = min_index;
    cover_columns( dp: &dp[0][0], index: min_index);
}

// Vypisem vysledok
for (int i = 0; i < n; i++) {
    cout << "Person " << i + 1 << " is assigned to Job " << assignments[i] + 1 << endl;
}

// vycistim pamat
delete[] assignments;
```

Prístup dynamického programovania je síce o čosi zložitejší na implementáciu, no jeho veľkou výhodou je garancia optimálneho riešenia.

```
Dynamic Programming Optimal Assignments:
Person 1 is assigned to Job 3
Person 2 is assigned to Job 1
Person 3 is assigned to Job 2
```

Analýza zložitostí

Celková časová zložitosť lačného algoritmu je $O(n^2)$, dominantný faktor je for-cyklus prechádzajúci všetkými riadkami (bežiaci n-krát) a v ňom vnorený ďalší for-cyklus prechádzajúci všetkými stĺpcami, tiež bežiaci n-krát.

Celková časová zložitosť algoritmu dynamického programovania je $O(n^3)$, dominantným faktorom sú tri vnorené for-cykly, všetky bežiace n-krát.

Celková priestorová zložitosť lačného algoritmu je $O(n^2)$, dominantným faktorom je samotná matica cien, ktorá má veľkosť n na druhú.

Celková priestorová zložitosť algoritmu dynamického programovania je tiež $O(n^2)$, dominantným faktorom je znovu matica cien. V tomto algoritme je aj nová matica pre medzivýsledky, ktorá má tiež veľkosť n na druhú. To ale nič nemení na tom že to patrí pod $O(n^2)$.

Porovnanie prístupov

Obidva prístupy majú svoje výhody a nevýhody, obidva majú rovnakú priestorovú zložitosť. Lačný algoritmus je síce časovo výhodnejší, no nevedie vždy k optimálnemu priradeniu. Naopak prístup dynamického programovania je síce časovo náročnejší, ale za to spoľahlivejší.