Analýza a zložitosť algoritmov

Dokumentácia k zadaniu

Radoslav Muntág

Cvičiaci: Ing. Michal Lüley

Contents

1. Scheduling with deadlines algoritmus

Pre input:

```
vector<Job> jobs = {
        { .index: 1,  .deadline: 2,  .profit: 40},
        { .index: 2,  .deadline: 4,  .profit: 15},
        { .index: 3,  .deadline: 3,  .profit: 60},
        { .index: 4,  .deadline: 2,  .profit: 20},
        { .index: 5,  .deadline: 3,  .profit: 10},
        { .index: 6,  .deadline: 1,  .profit: 45},
        { .index: 7,  .deadline: 1,  .profit: 55}
};
```

Output:

```
Selected Jobs: J7 J1 J3 J2
Total profit is 170
```

Analýza zložitosti

```
int schedule(vector<Job>& jobs, vector<int>& J, int max_deadline){
    int last_unfilled = 1;
    int total_profit = 0;

    for (const Job& job : jobs) {
        if (J[job.deadline] == -1){
            J[job.deadline] = job.index;
            total_profit += job.profit;
        }
        else if(J[job.deadline] > last_unfilled){
            for (int i = job.deadline; i >= last_unfilled; i--) {
                if (J[i] == -1){
                    J[i] = job.index;
                    total_profit += job.profit;
                    break;
                }
            }
        }
        for (int i = last_unfilled; i < max_deadline + 1; i++) {
            if (J[i] == -1) break;
            last_unfilled++;
        }
    }

    return total_profit;
}
```

Pre každý job sa prechádza vektor J aby sa mu našlo voľné miesto. Zložitosť je preto $O(n^2)$.

2. Scheduling with deadlines použitím Disjoint Set Data Structure III in Appendix C

Pre input:

```
vector<Job> jobs = {
        { .index: 1, .deadline: 2, .profit: 40},
        { .index: 2, .deadline: 4, .profit: 15},
        { .index: 3, .deadline: 3, .profit: 60},
        { .index: 4, .deadline: 2, .profit: 20},
        { .index: 5, .deadline: 3, .profit: 10},
        { .index: 6, .deadline: 1, .profit: 45},
        { .index: 7, .deadline: 1, .profit: 55}
};
```

Output:

```
Selected Jobs: J7 J1 J3 J2
Total profit is 170
```

Opis Kódu

V kóde je použitá implementácia algoritmu Disjoint Set Data Structure III (ďalej DS). Na začiatku je DS inicializovaný s universe, ktorý reprezentuje miesta v sekvencií jobov. Každé miesto je na začiatku prázdne, čo je reprezentované, že ich parent ukazuje na seba. Pri obsadení miesta parent bude ukazovať na root stromu, ktorí ukazuje na najbližšie voľné miesto (smallest). Voľné miesto a root môžu byť tie isté ak smallest ukazuje na seba. Pre získanie root skupiny je funkcia find. A pre získanie voľného miesta v skupine je funkcia small. Ak však smallest ukazuje na miesto s indexom 0, tak pre tento job nie je miesto a job nie je zaradený do sekvencie.

Analýza Zložitosti

```
int schedule(vector<Job>& jobs, vector<int>& J, int max_deadline){
    int total_profit = 0;

    int size =  min(max_deadline, (int)jobs.size());

    DS ds( size: size + 1);

    for (const Job& job : jobs) {
        int m = min(job.deadline, (int)jobs.size());

        int placement = ds.small( S: m);

        if (placement <= 0) continue;

        J[placement] = job.index;
        total_profit += job.profit;
        ds.merge( job1: placement,  job2: placement - 1);
    }

    return total_profit;
    }
```

Časová komplexita algoritmu schedule(vector<Job>& jobs, vector<int>& J, int max_dedline) je hlavne ovplyvnená veľkosťou vektoru jobs (neskôr len n kde n = jobs.size()) a max_deadline (neskôr len d kde d = max_deadline).

Ich hodnoty ovplyvňujú for loop, ktorý prechádza každý prvok vo vectore jobs (θ(n) komplexita) a funkcia find (θ(log(d)) komplexita), ktorá rekurzívne prechádza strom.

Pre každú iteráciu sa funkcia find spustí maximálne 3-krát (1-krát pri volaní ds.small a 2-krát pri volaní funkcie ds.merge)

```
int find(int i){
    if (i != universe[i].parent) {
        return find( i: universe[i].parent);
    }
    return i;
}

int small(int S) {
    int root = find( i: S);
    return universe[root].smallest;
}

void merge(int job1, int job2) {
    int root1 = find( i: job1);
    int root2 = find( i: job2);
    if (universe[root1].depth == universe[root2].depth){
        universe[root2].depth += 1;
        universe[root1].parent = root2;
        if (universe[root1].smallest < universe[root2].smallest){
            universe[root2].smallest = universe[root1].smallest;
        }
    }
    else if(universe[root1].depth < universe[root2].depth){
        universe[root1].parent = root2;
        if (universe[root1].smallest < universe[root2].smallest){
            universe[root2].smallest = universe[root1].smallest;
        }
    }
    else{
        universe[root2].parent = root1;
        if (universe[root2].smallest < universe[root1].smallest){
            universe[root1].smallest = universe[root2].smallest;
        }
```

Pre prípad že n >= d komplexita algoritmu je θ(n*logd), pre prípady kde je n >= d je tento prístup v poriadku, ale pre prípady kde n < d nechceme priraďovať deadline pre jobs, ktorý je väčší ako n. Tu nastupuje m = min(d, n) aby zabezpečil priradenie najmenší nutný deadline pre job. Z toho vyplýva že komplexita celého algoritmu bude θ(n*logm).

3. Two-way merge pattern greedy approach

Pre input:

```
vector<vector<int>> files = {
        {11, 12, 13, 14, 15},
        {1, 2, 3},
        {10},
        {4, 5},
        {6, 7, 8, 9}
};
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Pre input 2 :

```
vector<vector<int>> files = {
        {},
        {1, 2, 3},
        {4},
        {5, 6, 7, 8, 9, 10},
        {11, 12},
        {},
        {13, 14, 15, 16, 17, 18, 19},
        {20, 21, 22, 23},
        {24},
        {25, 26, 27, 28, 29, 30, 31, 32},
        {33},
        {34, 35, 36},
        {37, 38, 39, 40, 41, 42, 43, 44, 45}
};
```

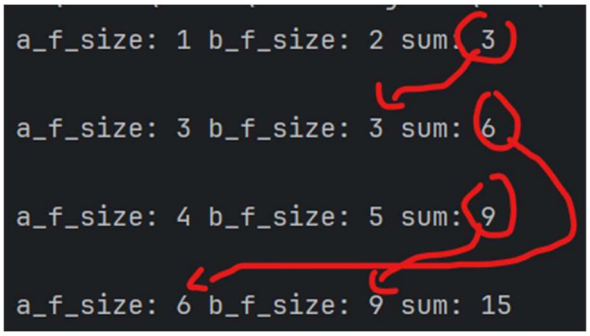
Output 2:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45
```

Opis kódu

Kód používa minheap pre výber dvoch najmenších súborov na spojenie, a tým minimalizuje počet potrebných iterácii cez súbory, aby ich spojil.

Postup pri spájaní



V tomto prípade je minimalizovaný počet nutných iterácií, ktoré sa nastávajú pri každom spájaní súborov. Celkový počet je 3 + 6 + 9 + 15 = 33.

Analýza zložitosti

```
while(!fileHeap.empty()){
    vector<int> a = fileHeap.top();
    fileHeap.pop();

    if(fileHeap.empty()){
        out = a;
        break;
    }

    vector<int> b = fileHeap.top();
    fileHeap.pop();

    vector<int> merge_out = {};

    two_way_merge( &: a,  &: b,  &: merge_out);

    fileHeap.push( Val: merge_out);
}
```

Pre n počet súborov, loop prejde (n – 1)-krát. Pre vyberanie z priority queue je čas konštantný. Na konci iteracie v loope sa vkladá merged_file späť do priority queue. Časová komplexita pre vkladanie do priority_queue je θ(logn).

Celková časová komplexita programu je preto θ(n*logn).

4. Generovanie huffman code bez huffman tree

Pre input:

Tabuľka pre zakódovanie:

```
vector<char> char_table = {'A', 'B', 'C', 'D', 'E', 'F'};
vector<int> freq_table = {45, 13, 12, 16, 9, 5};
```

Sekvencia pre dekódovanie:

```
"1010101000"
```

Output:

```
Standard Huffman Codes
A 0
D 111
B 101
C 100
E 1101
F 1100
Dekodovanie: BABAAA
Canonical Huffman Codes
A 0
D 100
B 101
C 110
E 1110
F 1111
Dekodovanie: BABAAA
Frequency Based Approximation
A 0
D 10
B 110
C 1110
E 11110
F 11111
Dekodovanie: DDDDAA
```

Standard Huffman Code Generating

V prípade štandardného generovania je použitý minheap, pre vyberanie najmenej frekventovaných charakterov vo funkci huffman_merging.

```
auto custom_cmp : bool (const Character *, const Character *) const = [](const Character* a, const Character* b) -> bool {
    return a->prob > b->prob;
};
priority_queue<Character*, vector<Character*>, decltype(custom_cmp)> minHeap( Pred: custom_cmp);

for (auto & current : Character & : table) {
    minHeap.push( Val: &current);
}
```

V loope sú vždy 2 najmenej frekventované charaktery spojené pomocou merge_characters. A out charakter udržuje list všetkých doposiaľ spojených charakterov a jeho frekvencia je suma týchto charakterov.

```
while (!minHeap.empty()){
    Character *a = minHeap.top();
    minHeap.pop();

    if (minHeap.empty()) break;
    Character *b = minHeap.top();
    minHeap.pop();

    auto *out = new Character;
    out->self = '\0';
    out->code = 0;
    out->prob = 0;
    out->rank = 0;
    out->merged = {};

    merge_characters(a, b, out);
    minHeap.push( Val: out);
}
```

Loop s $O(n)$ komplexitou a s použitím minheap push s $O(\log(n))$ komplexitou má celkovú komplexitu $O(n * \log(n)$. V loope sa ešte volá funkcia merge_characters.

```
void merge_characters(Character* a, Character* b, Character* out){
    out->merged.resize( Newsize: 2 + b->merged.size() + a->merged.size());
    int out_index = 0;

    a->code = a->code << 1;
    a->rank++;
    out->merged[out_index] = a;
    out_index++;

    for (Character* sub_a: a->merged){
        sub_a->code = sub_a->code << 1;
        sub_a->rank++;
        out->merged[out_index] = sub_a;
        out_index++;
    }

    b->code = (b->code << 1) + 1;
    b->rank++;
    out->merged[out_index] = b;
    out_index++;

    for (Character* sub_b: b->merged){
        sub_b->code = (sub_b->code << 1) + 1;
        sub_b->rank++;
        out->merged[out_index] = sub_b;
        out_index++;
    }
    out->prob = a->prob + b->prob;
}
```

Funkcia aktualizuje kód jednotlivých charakterov v subsetoch charakteru *a* a charakteru *b* a tak isto ich binárnu dĺžku (rank). Pre veľkosť subsetov k = a.merged.size() a m = b.merged.size(). Funkcia má časovú komplexitu $O(k + m)$.

Keďže merge_characters je volaná z loopu z funkcie huffman_merging a jej kombinovaná veľkosť všetkých spojených charakterov je proporčné k *n,* celková komplexita funkcie huffman_merging je:

$\mathrm{O}(n * \log(n))$

Canonical Huffman Code Generating

Pre konštruovanie kanonického huffman kódu je ptrebná bitová dĺžka pre každý charakter. Preto musí aj tu byť vykonaná funkcia huffman_merging, aby boli priradené tieto dĺžky.

```
void canonical(vector<Character>& table, unordered_map<string,char>& look_up_table){
    if (table[0].rank == 0){
        huffman_merging( &: table);
    }
    int binary = 0;
    int last_rank = 0;

    cout << "Canonical Huffman Codes" << endl;
    for (Character& current: table) {
        cout << current.self << " ";
        if (last_rank != 0){
            binary += 1;
            if(last_rank < current.rank){
                binary = binary << (current.rank - last_rank);
            }
        }
        current.code = binary;
        last_rank = current.rank;

        print_binary( &: current,  &: look_up_table);

        cout << endl;
    }
}
```

Ak nerátame volanie funkcie huffman_merging, kanonické generovanie má komplexitu $O(n)$. To vyplýva z faktu, že musí v loope prejsť len raz každý charakter z tabuľky, aby vygeneroval jeho kód.

So započítaním funkcie huffman_merging je táto komplexita $O(n * logn + n)$ čo je vo výsledku $O(n * logn)$.

Frequency Based Approximation

```
void frequency_based_approximation(vector<Character>& table, unordered_map<string,char>& look_up_table){

    cout << "Frequency Based Approximation" << endl;
    for (int i = 0; i < table.size(); ++i) {
        table[i].code = (1 << (i + 1)) - 1;
        table[i].rank = i;
        if(i != table.size() - 1){
            table[i].code = table[i].code << 1;
            table[i].rank += 1;
        }

        cout << table[i].self << " ";
        print_binary( &table[i],  &look_up_table);
        cout << endl;
    }
}
```

Pri generovaní kódu touto metódou, je každý charakter pre zakódovanie prechádzaný len raz. Z toho dôvodu je komplexita $O(n)$.

Zdroje:

Použitie chatGPT pre generovanie testovacích inputov a pomoc so syntaxou jazyka c++

https://www.geeksforgeeks.org/canonical-huffman-coding/

Neapolitan – Foundations Of Algorithms