Practical implementation and analysis of an algorithm *

Hlib Kokin
ID: 117991

December 3, 2023

*Documentation of Analysis and Complexity of Algorithms assignment, teacher: Ing. Ján Bartoš

Contents

1 Task 1

Schedule function has biggest complexity of O(n^2) because of nested loops in it, first loop goes through every possible job, second one takes to attention days available to assignment before actual deadline (for instance if deadline is 5, we check if 5 is already taken, if yes - check 4 etc.).

In main function there are sort function that is nlogn of complexity, and then function that prints the matrix which is O(n^2) as schedule function. so the biggest complexity is O(n^2) then complexity of this solution is O(n^2).

2 Task 2

According to the book, complexity of disjoint set 3 is O(nlogm), this is because of attributes depth (makes sets to merge to a more compact structure, where root is always the biggest depth i.e assigned earlier than other members of the set) and smallest which contains in it the

```
vector<int> schedule(int n, vector<vector<int>> input) { // input dvojrozmerne pole, kde 0 element je index deadlinu, 1 je deadline a 2 - profit
    // let result be collection of jobs indexes
    // as deadlines are sorted in decreasing order from the biggest job profit
    vector<int> result(n+1);
    vector<int> empty_indexes(n+1);                        // let empty_indexes be a vector with states of indexes
    result.resize(input.size());
    result[input[1][1]] = input[1][0];

    // we will initialize free indexes with number -1 on the deadline of the first element as it is already picked
    empty_indexes[input[1][1]] = -1;

    // we start from the i = 2, because 1st element is already the best option
    // outer loop is for n iterations, complexity is O(n)
    // inner is O(n) aswell, together they give us O(n^2)
    // other operations are O(1)
    // so, the complexity is O(n)
    for (int i = 2; i <= n; i++) {                          // O(n)
        for(int j = input[i][1]; j > 0; j--) {              // O(n)
            if (empty_indexes[j] != -1) {
                result[j] = input[i][0];
                empty_indexes[j] = -1;
                break;
            }
        }
    }
    result.erase(result.begin());
    result.resize(result.size() - 1);
    return result;
}
```

Figure 1: task1_schedule

smallest member of the set which, if the deadlines before actual selected deadline are taken, will be 0, which means that job for that deadline cannot be assignmed in any way. Those modifications (depth and smallest) will save us some iterations, for example iteration to the top of the tree to find the smallest deadline that has been taken and in result we will achieve better complexity of O(nlogm)

The biggest complexity oin schedule function is merge that is O(nlogm), merge have this modification because root member always got to have smallest member so we got to update it and sinse it is a tree - O(nlogm)

```
void merge(set_pointer p, set_pointer q) {
    // if both sets are at the same depth and q set doesn't already have parent
    if (U[p].depth == U[q].depth && equal( p: U[q].parent, q)) { // O(1)
        U[p].depth = U[p].depth + 1;
        U[q].parent = p;
        if (U[q].smallest < U[p].smallest) {
            U[p].smallest = U[q].smallest;
        }
    }
    // either depths of sets are not the same or q set has parent that is not q itself
    else if (U[p].depth <= U[q].depth) { // O(nlgn)
        U[p].parent = q;

        // update smallest member of the set in the root
        while (U[p].smallest < U[q].smallest) { // O(nlgn)  thanks to depth attribute
            U[q].smallest = U[p].smallest;
            index parent = U[q].parent;
            p = q;
            q = parent;
        }
    }
    // p set is deeper
    else { //O (1)
        U[q].parent = p;
        if (U[q].smallest < U[p].smallest) {
            U[p].smallest = U[q].smallest;
        }
    }
}
```

Figure 2: task2_merge

3 Task 3

3.1 A

Greedy approach that is O(n^2) because we go through matrix once, check each row for max value, take it, ban the column that we found it in, continue to the next row. With such approach we will find only local optimum, maybe sometimes it will be the same with global optimum

3.2 B

Steps are described in code with complexity for each function

```cpp
void schedule(vector<vector<int>> input) {

    int d = get_max_deadline(input);
    // creates d + 1 sets
    initial( i: d);

    // go through all jobs
    for (int i = 1; i <= n; i++)
    {
        // maximum available free slot for this job
        // input[i][1] - deadline of the first job
        set_pointer availableSlot = find( s: input[i][1]); // O(n)
        // smallest member of the found set
        int availableSlotSmallest = small( p: availableSlot); // O(1)

        // If smallest available slot is not 0 (what would mean that no free slots left for this deadline)
        // then proceed to scheduling
        if (availableSlotSmallest > 0)
        {
            // merge sets of currently selected one and smaller (earlier set beside it)
            merge( p: find( s: availableSlotSmallest - 1),  q: availableSlotSmallest); // O(nlgn)
            cout << "job number " << input[i][0] << " at day " << availableSlotSmallest << "; " << endl;
        }
    }
}
```

Figure 3: task2_schedule

```cpp
vector<int> schedule(int n, vector<vector<int>> matrix) { // O (n^2)
    vector<int> res(n);
    vector<int> taken(n,  value: -1); // contains indexes of jobs that are taken


    vector<int> free_jobs(n);
    // this cyclus will be O(n)
    for (int i = 0; i < n; i++) {
        free_jobs[i] = i;
    }

    for (int p = 0; p < n; p++) {       // check person
        int min_val = 30;               // value that will contain min and initialized with the biggest value, which cannot be in matrix
        int min_index = -1;             // index of the job that will be the best
        int erase_index = -1;
        for (int j = 0; j < free_jobs.size(); j++) {   // check job one by one
            if (matrix[p][free_jobs[j]] < min_val) {   // if job is smaller than minimum - we take a closer look
                min_val = matrix[p][free_jobs[j]]; // set new minimum value
                min_index = free_jobs[j];          // and index
                erase_index = j;
            }
        }
        res[p] = min_index + 1; // add to result
        taken[p] = min_index; // add to taken
        free_jobs.erase( position: free_jobs.begin() + erase_index); // shrink our free jobs vector

    }

    // Outer cycle is O(n), Inner as well O(n), which makes it O(n^2),
    // first cycle with assigning free_jobs indexes doesn't influence the common complexity since it is faster than
    // main cycle
```

Figure 4: task3a_schedule

O(n$\hat{3}$) is the classic complexity of the Hungarian algorithm but since steps 3 and 4 will be repeated until number of minimum lines is n that makes it undefined number of iterations (in reality for n 10 it is 1 - 3 iterations, but i couldnt predict why) .

Step 3 (getMinimumLines) will be O(n^2) because we need to check each zero in matrix and the worst case will be when we need n lines to cover all zeroes in each of such iteration (for each line) we have couple loops (total of O(n)) where we calculate the best line to draw to achieve minimum drawn lines.

Since step 3 has the biggest complexity of O(n^2) and classic Hungarian algorithm is told to be O(n^3) i can suggest that that loop for undefined iterations will have complexity of O(n) but there are still no visible arguments for this to be respectable statement.

Except Hungarian algorithm itself I had to find out how to choose such order that will be best, i found a way in which i try all 4 possible ways to start taking cells in rows (from top left, top right, bottom left, bottom right), to test each of them it takes O(n^2) which is no bigger than other functions, it is not nested so the total complexity of lookin up the assignment order is O(n^2)

```cpp
void hungarianAlgo(vector<vector<int>>& matrix, vector<vector<int>>& initMatrix) {
    int n, rows, cols, lines_cnt = 0;
    vector<Line> lines;

    n = rows = cols = matrix.size() - 1;

    // maximization of profit case
    reductionByMax( &: matrix);                    // O(n^2)
    // step 1
    rowReduction( &: matrix);                      // O(n^2)
    // step 2
    columnReduction( &: matrix);                   // O(n^2)
    // print matrix after all reductions
    cout << "Matrix after reductions: " << endl;
    print_matrix( &: matrix);                      // O(n^2)
    // step 3 - check if minimum of lines that cover all zeroes is equal to number of rows/columns
    while (lines_cnt != n) {                        // O(?)
        lines = getMinimumLines( &: matrix);       // O(n^2)
        lines_cnt = lines.size();
        cout << "Minimum number of lines is " << lines_cnt << endl;
        //step 4 - add additional zeroes
        if (lines_cnt != n) {
            matrix = addZeroes( &: matrix, &: lines);   // O(n^2)
            print_matrix( &: matrix);
        }
    }
```

Figure 5: Hungarian algorithm

```cpp
for (int i = 1; i <= rows; i++) {   // O(n^2)
    for (int j = 1; j <= cols; j++) {
        if (bannedCols[j] != -1 && matrix[i][j] == 0) {
            profit += initMatrix[i][j];
            res[i] = j;
            bannedCols[j] = -1;
            assigned += 1;
            break;
        }
    }
}
```

Figure 6: One of four possible ways to find assignment order that is the best O(n^2)