Chapter 16 Solusion

github.com/frc123/CLRS

1/10/2022

16.1

16.1-1

```
void OutputAux(const std::vector< std::vector<int> >& dp_selection,
    int i, int j, std::list<int>& output)
{
    if (dp_selection[i][j] > 0)
    {
        OutputAux(dp_selection, i, dp_selection[i][j], output);
        output.push_back(dp_selection[i][j] - 1);
        OutputAux(dp_selection, dp_selection[i][j], j, output);
    }
}

// assume intervals are sorted by finish time
std::list<int> DpActivitySelector(const std::vector<Activity>& intervals)
{
    int n, i, j, l, k, l_size;
    n = (int)(intervals.size());
    // dp_size index start by 1
    std::vector< std::vector<int> > dp_size(n + 2, std::vector<int>(n + 2, 0)),
        dp_selection(n + 2, std::vector<int>(n + 2, -1));
    // compute
    for (l = 2; l <= n + 1; ++l)
    {
        for (i = 0; i <= n + 1 - l; ++i)
        {
            j = i + l;
            for (k = i + 1; k <= j - 1; ++k)
            {
                if ((i == 0 || intervals[k - 1].s >= intervals[i - 1].f) &&
                    (j == n + 1 || intervals[k - 1].f <= intervals[j - 1].s))
                {
                    l_size = dp_size[i][k] + dp_size[k][j] + 1;
                    if (dp_size[i][j] < l_size)
                    {
                        dp_size[i][j] = l_size;
                        dp_selection[i][j] = k;
                    }
                }
            }
        }
    }
    // output
    std::list<int> output;
    OutputAux(dp_selection, 0, n + 1, output);
    return output;
}
```

The dynamic-programming algorithm runs in $O(n^3)$.

16.1-2

```
// assume intervals are sorted by start time
std::list<int> GreedyActivitySelector(const std::vector<Activity>& intervals)
{
    int k, m, n;
    std::list<int> activities;
    n = (int)(intervals.size());
    activities.push_front(n - 1);
    k = n - 1;
    for (m = n - 2; m >= 0; --m)
    {
        if (intervals[k].s >= intervals[m].f)
        {
            activities.push_front(m);
            k = m;
        }
    }
    return activities;
}
```

Claim 1. Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the latest start time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof. Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the latest start time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the last activity in A_k to start, and $s_m \geq s_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. □

16.1-3

1. selecting the compatible activity of least duration

i	1	2	3
s_i	1	3	4
f_i	4	5	7

By this approach, the solution will be $\{a_2\}$. However, the optimal solutiopn is $\{a_1, a_3\}$.

2. selecting the compatible activity that overlaps the fewest other remaining activities

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	1	1	1	2	3	4	5	5	5	6
f_i	2	3	3	3	4	5	6	7	7	7	7

By this approach, the solution will include a_7. However, the optimal solution is $\{a_1, a_6, a_8, a_{11}\}$, and a_7 is not compatible with the optimal solution.

3. selecting the compatible activity with the earliest start time

i	1	2	3
s_i	2	3	1
f_i	3	4	5

By this approach, the solution will be $\{a_3\}$. However, the optimal solutiopn is $\{a_1, a_2\}$.

16.1-4

```
struct Element
{
    int interval;
    std::list< std::list<int> >::iterator list;

    Element(int interval, std::list< std::list<int> >::iterator list)
        : interval(interval), list(list) {}
};
```

```

// assume intervals are sorted by finish time
std::list< std::list<int> > IntervalGraphColoring(const std::vector<Activity>& intervals)
{
    int i, n;
    n = (int)(intervals.size());
    std::list< std::list<int> > collection;
    auto heap_cmp = [&intervals](const Element& a, const Element& b) {
        return intervals[a.interval].s < intervals[b.interval].s;
    };
    std::priority_queue<Element, std::vector<Element>, decltype(heap_cmp)> heap(heap_cmp);
    std::list< std::list<int> >::iterator curr_list;
    collection.emplace_front();
    curr_list = collection.begin();
    curr_list->push_front(n - 1);
    heap.emplace(n - 1, curr_list);
    for (i = n - 2; i >= 0; --i)
    {
        if (intervals[i].f <= intervals[heap.top().interval].s)
        {
            curr_list = heap.top().list;
            curr_list->push_front(i);
            heap.pop();
        }
        else
        {
            collection.emplace_front();
            curr_list = collection.begin();
            curr_list->push_front(i);
        }
        heap.emplace(i, curr_list);
    }
    return collection;
}
```

16.1-5

This algorithm is actually a revision from 16.1-1.

```
// assume intervals are sorted by finish time
std::list<int> DpActivitySelector(const std::vector<Activity>& activities)
```

```
{
    int n, i, j, l, k, l_size;
    n = (int)(activities.size());
    // dp_size index start by 1
    std::vector< std::vector<int> > dp_size(n + 2, std::vector<int>(n + 2, 0)),
        dp_selection(n + 2, std::vector<int>(n + 2, -1));
    // compute
    for (l = 2; l <= n + 1; ++l)
    {
        for (i = 0; i <= n + 1 - l; ++i)
        {
            j = i + l;
            for (k = i + 1; k <= j - 1; ++k)
            {
                if ((i == 0 || activities[k - 1].s >= activities[i - 1].f) &&
                    (j == n + 1 || activities[k - 1].f <= activities[j - 1].s))
                {
                    l_size = dp_size[i][k] + dp_size[k][j] + activities[k - 1].v;
                    if (dp_size[i][j] < l_size)
                    {
                        dp_size[i][j] = l_size;
                        dp_selection[i][j] = k;
                    }
                }
            }
        }
    }
    // output
    std::list<int> output;
    OutputAux(dp_selection, 0, n + 1, output);
    return output;
}
```

Updating...