Assignment 7

Algorithms & Complexity (CIS 522-01)

Javier Arechalde

April 20, 2018

1. PC Manufacturer

Problem model

In this problem, we are a consulting company that works for a company that manufactures PC equipment. This company will have a projected supply for each week $S = [s_1, s_2, ..., s_n]$. To deliver each weeks supply we can choose between two different carriers. One of them charges r per pound of supply. Another one has a fixed rate c per week, but if we choose this carrier we will have to choose for 4 consecutive weeks. We will have to find the best combination possible of carriers to pay the least money possible.

Input:

- Price per pound on Carrier 1: r
- Price per week on Carrier 2: c
- Projected supply for each week: S

Output:

- The optimal schedule: Sch

Class

This algorithm belongs to P algorithms, as it is a decision problem that can be easily implemented in polynomial time.

Algorithm

At the beginning of our algorithm, we will start by only choosing company A, until we reach week 4, once we reach week 4, we will start checking what is more expensive, if choosing company A for the past 3 weeks and that week, or choosing the flat rate of company B. In case using company B is the better choice, we will update the best cost for that week with the best cost four weeks ago plus the flat rate multiplied by the four weeks. Otherwise we will add the cost per pound multiplied for the projected supply for that week to the previous week best cost.

In the end, we will have an array containing the best cost possible for each of the weeks, and the schedule.

Pseudocode

Algorithm 1 Carrier Selection Pseudocode

```
1: We set $Cost_0 = 0$
2: for $i$ in range $0 \to n$ do
3:     if $i < 3$ then
4:         $Cost_i = Cost_{i-1} + s_i * r$
5:     else
6:         if $Cost_{i-1} + s_i * r > 4 * c$ then
7:             $Cost_i = Cost_{i-4} + 4 * c$
8:         else
9:             $Cost_i = Cost_{i-1} + s_i * r$
10:        end if
11:    end if
12: end for
13: return Best cost possible $Cost_n$
```

Implementation

An example of how this algorithm works can be seen by running *Problem1.py*

Time complexity

The running time of our implementation will be $O(n)$, as we have to go through the n weeks to find the lowest cost possible to deliver our product supply.

2. Processes scheduling

Problem Model

In this problem, we have n processes on a system, each one of them being capable of running multiple jobs concurrently. Some jobs can't be scheduled at the same time because they both may need the same resource. We want to schedule in the next k steps of the system all the jobs to run in at least one of the processes.

Class

This problem belongs to the *NP-Complete* class of problems, as it can be recognized as a partitioning problem, in which we have to divide in this case the jobs into different subsets, the processes, in a way that each one of the jobs appears in exactly one of the processes. More specifically, we can relate this problem to the graph coloring problem, as there exist conflicts between some of the jobs.

Complexity of the problem

We will follow the following strategy to prove thato our problem is NP-Complete.

- Prove that $X \in NP$
- Choose a problem Y that is known to be NP-Complete
- Probe that $Y \leq_p X$

For our given problem, the certifier B will prove that all the jobs are scheduled in at least one process, and at the same time that not two jobs are scheduled at the same time.

Then we choose a problem Y, in this case the graph coloring problem, that we know that is NP-Complete.

In the case of graph coloring, we have k colors, and a Graph G, we want to find out if there is a possible k-coloring for that graph. There is also conflicts between objects, which means that conflicting objects can't go into the same set.

In the case of our problem, we have n processes, and k steps of the system. We want to schedule the different jobs in our system to run in at least one of the processes, while there is also conflicts between our jobs, because some of them need the same resource, so they can't run at the same time.

So we can relate the colors to the number of steps in our problem, so if we can color the grap with k colors, we can also schedule the processes in k steps of the system. And also the incompatibilities between some objects that can't go into the same set can be seen as the incompatibilities between two jobs that can't be scheduled at the same time because they need to access the same resource.

3. Database analysis

In this case, we have two databases, each one of them containing n values, we assume that not two values are the same. We want to find the median of this $2n$ values, using the mimimum number of queries as possible. To find this value, we will have to access the data in the databases. By specifying k to a database, this database will return the k^{th} smallest value.

Class

This problem belongs to the P class, as it can be easily implemented in Polynomial time.

Problem Model

We will name each one of the databases D_1 and D_2, each one of these databases contains n values, and the query $D_i(k)$ will return the k^{th} smallest value in that database. We will also have two iterators c_1 and c_2, each one of them for one of the available databases we have.

Algorithm

What our algorithm will do is iterate through the two databases, we will first compare the lowest values of both databases, if the lowest value is in the first database, we will increase the counter of the first database, else we will increase the other counter. We will continue to compare values, until we do $n + 1$ comparisons. Whenever we reach the $(n + 1)^{th}$ comparison, the value resulting from that comparison plus the value from the last comparison divided by 2 will be the median, as we have an even number of values $2n$.

Pseudocode

Algorithm 2 Carrier Selection Pseudocode

```
1: We initiallize $c_1 = c_2 = 0$
2: We initiallize $val_n = val_{n1} = 0$
3: for $i$ in range $0 \to n+1$ do
4:     $val_1 = D_1(c_1)$
5:     $val_2 = D_2(c_2)$
6:     $val_n = val_{n1}$
7:     if $val_1 > val_2$ then
8:         $val_{n1} = val_1$
9:         $c_1 = c_1 + 1$
10:    else
11:        $val_{n1} = val_2$
12:        $c_2 = c_2 + 1$
13:    end if
14: end for
15: return $median = (val_n + val_{n1})/2$
```

Implementation

An example of how this algorithm works can be seen by running *Problem3.py*

Time Complexity

This algorithm can find the median of the values in the two databases in $O(n)$ time, as it only needs to compare $n+1$ values from the two databases.

4. Photocopying Service

Problem Model

In this problem we will have different customers, each one of them having a job that takes t_i to complete. Also, each one of these jobs has a weight w_i that is the importance of that customer to the business.

We want to find the order of jobs that minimizes the weigthed sum of the completion times: $\sum i = 1nw_iC_i$.

Input:

- List of the time that takes to complete each client's job: $T = [t_1, t_2, ..., t_n]$
- List of the weight for each client's job: $W = [w_1, w_2, ..., w_n]$

Output

- List containing the optimal order of jobs: $Schedule = [job_x, job_y, ...]$

Class

This problem belongs to P, as it can be solved in polynomial time. We will prove this later on.

Algorithm

We will solve this problem using a greedy approach, first we will sort the jobs by decreasing weight w_i using quicksort algorithm, then we will schedule this jobs, starting by the ones that have higher w_i first. This way, we can minimize the weighted sum of completion times.

Pseudocode

Algorithm 3 Job scheduling

```
First we sort the jobs by weight using QuickSort
function QUICKSORT(array)
    if Array length is 1 then
      return array
    end if
    if Array length > 1 then
        $pivot == array[0]$
        for Element in array do
            if $element < pivot$ then
                We add the element to the lowerlist
            end if
            if $element > pivot$ then
                We add element to the upperlist
            end if
            if d then
                Append element to the pivotlist
            end if
        end for
        upperlist = QUICKSORT(upperlist)
        lowerlist = QUICKSORT(lowerlist)
    end if
    Return lowerlist + pivotlist + upperlist
end function

Now we will start scheduling the jobs by decreasing weight order
We initialize start time of jobs: $s_{job} = 0$
while We didn't schedule all jobs do
    Get the timings for the corresponding job

    Job start and finish
    $s_{job} = f_{previousjob}$
    $f_{job} = s_{job} + t_i$

    We return the start and finish time for each one of the jobs
end while
```

Implementation

An example of how this algorithm works can be seen by running *Problem4.py*

Time Complexity

The time complexity of the scheduling part of the algorithm is $O(n)$, as we only need to go through the list once to schedule the jobs, but prior to this we need to sort our jobs by using quicksort algorithm, which has a time complexity of $O(n \log n)$. Then, as $O(n \log n)$ is upper bound of $O(n)$, the time complexity of our implementation will be $O(n \log n)$.

5. Communication network

Problem model

In this problem, we have a communication network, modeled as a directed grapg $G = (V, E)$, a source node s and a sink node t. There are c users that want to make use of this network. Each one of these users will reserve a specific path P_i that goes from s to t, and we have the following restriction: if two users i and j request a path P_i and P_j, then P_i and P_j cannot share any edges.

We want to be able to accomodate as many users as possible from c to use the communication network, following the given prerequisite.

What we will do is to model the communication network as the directed graph G, and give all the edges a capacity of 1, this way, we can reduce the problem to a problem of finding the maximum flow. Then if the maximum flow equals c, we can accomodate all the users on the network, otherwise the network capacity is not sufficient.

Class

The *Ford-Fulkerson* algorithm for finding the maximum flow in a directed graph can be implemented in Polynomial time, so this problem belongs to P.

Time Complexity

The *Ford-Fulkerson* algorithm can be used to find a maximum flow in our directed graph in $O(mn)$ time as we dont have any duplicate edges in our graph, and each edge has unit capacity. Where m is the number of edges, and n is the number of nodes.

6. Project selection

In this problem we are a student in the end of the semester, and we have n final projects. Each one of these projects will give us some points for our final performance. Also, some of these projects have prerequisites, which means that we can only work on that project if we completed the prerequisite projects first. Our goal is to select a set of projects that will maximize the utility points we get.

Problem model

We will model this problem as a network-flow problem.

In this case, the nodes will be the different projects, and each one of these projects will have associated to it the points that we can get from each of this project. Then for each one of the prerequisites for a project, we will create and edge that goes from that project to the prerequisite project. We won't assign any capacities for the edges just yet.

Then we will model this problem, as the project selection problem. We will start by adding a source s and a sink t. Then for each project that has positive points, we will create and edge that goes from the source, and the node representing that project,this edge having the number of points as its capacity. For each project that has negative points, we will create and edge that goes from that node to the sink, that edge having the capacity equal to that number of points. For the preexisting edges, we will set their capacity to ∞, which means that that edge has no upper bound.

Then, we will find the maximum number of points that we can achieve by working on different projects, by finding the minimum cut of the graph, by using *Ford-Fulkerson* algorithm.

Class

This problem can be reduced to a problem of finding the maximum flow in a graph, then it can be solved using *Ford-Fulkerson* algorithm which can be implemented in Polynomial time, so this problem belongs to P.

Time complexity

The time complexity of this algorithm will be $O(mn)$, as its based on *Ford-Fulkerson* algorithm. Where m is the number of edges, that will be equal to all the prerequisites plus the number of projects, and n will be the number of nodes, that in this case will be the number of projects plus the source node, and the sink node.

7. Summer sports camp

In this problem, we are helping organize a summer sports camp. The camp needs to hire at least one counselor who is skilled at each of the n sports the camp offers. They have applications from m potential counselors, for each of the n sports, a subset of the m applicants is qualified in that sport. We want to find out if we will be able to hire a number of counselors $k < m$, and have at least one counselor qualified in each one of the n sports.

Problem model

We will model this problem as bipartite matching problem, in which we will have two sets, X, and Y. We first will model the graph as a bipartite graph G, which is an undirected graph, which has the property that every edge, has one end in X, and the other one in Y.

Then we will construct a flow network G' from G. First we will direct all edges in G from X to Y. Then we will create a source, and add one edge for every node in X that goes from source s to that node. After that, we will create a sink t, and for every node in Y we will ad an edge that goes from Y to t. In the end, we will give each edge in G' a capacity of 1.

Then we will compute the maximum flow in G', and the maximum flow, will equal the maximum matching in G.

Class

The *Ford-Fulkerson* algorithm for finding the maximum matching in a bipartite graph can be implemented in Polynomial time, so this problem belongs to P.

Time complexity

The *Ford-Fulkerson* algorithm can be used to find a maximum matching in a bipartite graph in $O(mn)$ time, where m is the number of edges, and n is the number of nodes, in this case the number of nodes will be the number of counselors plus the number of sports offered in the camp and the source and sink node, and the number of edges will be equal or less to the number of counselors multiplied to the number of nodes plus the number of counselors and the number of sports.