Assignment 3 **Wen Han Xu**
CS711008Z ID: 201818018670073
Date: 2018/11/16

1 Node Degrees

Given a list of n natural numbers $d_1, d_2, ..., d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, ..., d_n$. G should not contain multiple edges between the same pair of nodes, or loop edges with both endpoints equal to the same node.

1.1 Algorithm Description

We firstly sort the set of node degrees so that we have $d_1 \geq d_2 ... \geq d_n$. We can construct the desired graph G starting from node with the largest degree (d_1). When node d_i is placed into the graph G, we expect to connect it with nodes that haven't yet joined G. That is, we subtract the degrees of nodes ranging in $[d_{i+1}, d_{i+d_i}]$ by 1. We will later prove that, by recursively executing this step, a graph G can be constructed if the degree of the last node satisfies $d_n = 0$ at the end of the loop.

We further noticed that after the subtraction in $[d_{i+1}, d_{i+d_i}]$, their orders are not disrupted, thus to restore order in $[d_{i+1}, d_n]$, we only need to modify the position of d_{i+d_i} and d_{i+d_i+1}. A singly linked array can be used to store the sorted $(d'_1, d'_{i+1}, ..., d'_n)$, thus every operation to restore order only takes constant time.

1.1.1 Pseudo-Code

We describe the algorithm *NodeDegrees* in pseudo-code. It takes $d_1, d_2, ..., d_n$ as its input, and outputs whether there exists an undirected graph G as described above.

Algorithm 1 Node Degrees

```
1:  procedure NodeDegrees(d_1, d_2, ..., d_n)
2:      SinglyLinkeList(d'_1, d'_{i+1}, ..., d'_n) ← Quicksort(d_1, d_{i+1}, ..., d_n)(descending)
3:      for i ← 1 to n do
4:          for j ← i + 1 to i + d_i do
5:              d'_j = d'_j − 1
6:          Restore order in (d'_i, d'_{i+1}, ..., d'_{i+d_i})(descending)
7:      if d'_n = 0 then
8:          return True
9:      else
10:         return False
```

1.1.2 Greedy-Choice Property

To construct the desired graph, we put the node with the largest degree into the graph at each step.

1.2 Proof of Correctness

When there exists a descending sequence $(d_{i+1}-1, ..., d_{i+d_i}-1, ..., d_n)$ that forms a desired graph G'. One can easily join node d_i into the graph by connecting it with nodes $(d_{i+1}-1, ..., d_{i+d_i}-1)$. Thus, a graph can be constructed by processing node with the largest degress at each step.

1.3 Time Complexity

The initial *Quicksort* operation takes $O(\log n)$ time. The outer loop runs for exactly n times. We use $m = \sum_{i=1}^{n} d_i$ to denote In each loop, d_i subtraction is performed. A constant time operation is performed to restore the list order. Thus, the overall time complexity of Algorithm 1 is $O(\log n + m)$.

2 Job Scheduling

There are n distinct jobs, labeled $J_1, J_2, ..., J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC. Since there are at least n PCs available on the premises, the finishing of the jobs can be performed on PCs at the same time. However, the supercomputer can only work on a single job a time without any interruption. For every job, as soon as the preprocessing is done on the supercomputer, it can be handed over to a PC for finishing.

Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earlist time at which all jobs have finished processing on the PCs. Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

2.1 Algorithm Description

We sort the n distinct jobs according to their finishing time f_i by descending order to get a new Job sequence $S = (J'_1, J'_2, ..., J'_n)$ $(f'_1 \geq ... \geq f'_n)$. The following algorithm solves the problem by scheduling jobs with the longest finishing time first.

2.1.1 Pseudo-Code

Algorithm 2 Job Scheduling

```
1: procedure JobScheduling(J_1, J_2, ..., J_n)
2:     (J'_1, J'_2, ..., J'_n) ← Quicksort(J_1, J_2, ..., J_n)(descending on f_i)
3:     return (J'_1, J'_2, ..., J'_n) is the job schedule that has the smallest completion time
```

2.1.2 Greedy-Choice Property

It's easy to notice that $\sum_{i=1}^{n} p_i$ must be spent on the supercomputer, so our intuition when designing the scheduling algorithm is to maximize the parallel processing capability of n PCs, that is, to make as many PCs busy as possible.

However, we later notice that, for a job sequence $J'_1, J'_2, ..., J'_n$, the time until job J_k finished is $c_k \sum_{i=1}^{k} p_i + f_k$, which means that if we schedule jobs based on their preprocessing time and deal with jobs that have smaller p_i first, in some special cases, we may encounter a job J_m that

has both the longest preprocessing time and finishing time. If we process J_m at last, then it will finish at $c_m = \sum_{i=1}^{n} p_i + f_m$, and because $\sum_{i=1}^{n} p_i$ is the time necessary for all jobs to be done on the supercomupter, c_m is obviously not a optimal solution to this problem.

Thus, the greedy-choice property to this problem would be to process those jobs that have longer finishing times first (or to process the jobs .

2.2 Proof of Correctness

Here we want to prove that for any given schedule S, there is a method to construct another schedule $S' \neq S$ such that S' has completion time less than or equal to S.

Consider two consecutive job J_i, J_j in the job sequence of S, suppose that $f_i < f_j$. Then we can optimize this schedule by simply swapping J_i with J_j, note that this operation do not effect effect the completion time of any other jobs in the sequence. As J_j is feed into the supercomputer earlier than the original schedule, it also completes earlier. As J_j is larger than J_i, swapping these two won't result in a longer completion time of the entire schedule.

Thus, one can eventually get a optimal schedule by swapping jobs to get a descending order. The greedy-choice schedule is optimal, algorithm 2 is correct

2.3 Time Complexity

Sorting the job sequence takes $O(n \log n$ time, which is also the overall time complexity of the algorithm.