091M4041H - Assignment Three
Greedy Algorithm

张 帅
201828018670119
网络空间安全学院, UCAS

November 22, 2018

1 是否存在无向图?

1.1 问题描述

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

1.2 基本思想

假设给定的节点及其度数可以构成图，每次从图中选择一个度数最大的节点，将该节点及所有与其相连的边删除，重复这个过程，那么到最后无向图G必然会变为空。

将度数从大到小排序，每次选取度数最大的点（度数为d_i），假设它与随后的$|d_i|$个节点用边连接，将这些边删除，即：随后的$|d_i|$个节点的度数都减1，后面的度数出现-1的情况，说明没有那么多节点与该节点相连，所以就无法构成图；否则，重复该过程，直到遍历完所有的节点为止。

1.3 伪代码

1.4 贪心选择性质

每次选择度数最大的节点删除边，如果该节点都无法满足其度数，那么肯定无法构成图；否则，继续判断。

1.5 最优子结构性质

从上面的算法思想以及贪心选择中我们可以知道，原问题的解与子问题的解(删去第一个节点及所有与其相连的边的之后的子图)相同。

$$OPTD[i \ldots n] = \begin{cases} OPT(D^{'}[i+1 \ldots n]) & if\ condition\ 1 \\ false & otherwise \end{cases}$$

式中$condition1$为：在$[i+1, n]$节点区域中，存在$|D[i]|$个节点可以和节点i连通，如果该条件不成立，则返回false。$D^{'}[i+1 \ldots n]$为与节点i连通的各个节点度数减1之后的度数列表。

PROBLEM 1 Determine whether there exists an undirected graph

INPUT: Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, D, n.

OUTPUT: Determine whether there exists an undirected graph.

```
function IsExistUndirectedGraph(D, n)
    flag ← true
    for i = 1 → n do
        sort D[i:n] in non-increase order
        for j = i + 1 → i + D_i do
            if D_j − 1! = −1 then
                D_j ← D_j − 1
            else
                return false
            end if
            D_i ← 0
        end for
    end for
    return true
end function
```

1.6 正确性证明

假设能构成图$G = (V, E)$，那么依次删除度数最大的节点相连的各条边，删到最后，各个节点的度数必然都为0，所以返回true成立。

而如果存在一个节点的度数要大于子图$G' = (V - D_i, E - E_{D_i})$中节点的个数，则不能构成图。这是因为题目要求，构成的图中，两个节点之间至多只能有一条边，且不存在边的两个端点是同一个节点的情况，所以肯定该节点肯定无法达到其期望的度数，返回false成立。

1.7 复杂度分析

- **时间复杂度:**
  排序时间复杂度为为$T(n) = O(nlogn)$

  最好的情况是搜到第一个就出现返回false的条件，时间复杂度为$T(n) = O(nlogn)$;

  最坏的情况即存在一个简单完全无向图，则度数总和为$n(n-1)/2$，时间复杂度为:

$$\begin{aligned} T(n) &= n * nlogn + c * n(n-1)/2 \\ &= O(n^2 logn) \end{aligned}$$

  而一般情况为$T(n) = O(n^2 logn)$

- **空间复杂度:**
  额外使用的空间是常数量级的，即为$O(1)$。

2 作业调度

2.1 问题描述

There are n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each jop 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 off to a PC for finishing.

2.2 基本思想

首先将作业按照f_i按照非增序排序，排序之后的作业序列即是作业调度的序列，然后记录每个作业的完成时间，返回完成时间的最大值即可。

2.3 伪代码

PROBLEM 2 Job Scheduling

INPUT: Given two lists P and F. And the number of jobs, n.

OUTPUT: Determine the minimum time that overcome these n jobs, m.

```
function JobScheduling(P, F, n)
    sort job by P in non-incresing order
    m ← 0
    cur_sup ← 0 /*current supercomputer's running time */
    for i = 1 → n do
        cur_sup ← cur_sup + P[i]
        m ← Max(m, cur_sup + F[i])
    end for
    return m
end function
```

2.4 贪心选择性质

每次从工作集中选择f_i最大的先执行，然后重复这样的动作。

2.5 最优子结构性质

为了方便计算，我们定义OPT由两部分组成，当前的最晚完成时间$OPT(i)$和当前supercomputer上的运行时间$CurT(i)$。

$$CurT(i) = CurT(i-1) + P[i]$$

$$OPT(i) = max \begin{cases} CurT(i) + F[i] \\ OPT(i-1) \end{cases}$$

2.6 正确性证明

2.6.1 证明1

假设作业k为完成时间最晚的作业，它的完成时间为$\sum_{i=1}^{k} P[i] + F[k]$，现在证明在作业$k$后面不可能出现一个作业$l$，使得$F[l] > F[k]$。对于作业$l$，它的完成时间为$\sum_{i=1}^{l} P[i] + F[l]$。

因为$k < l$，所以很容易得出$\sum_{i=1}^{k} P[i] < \sum_{i=1}^{l} P[i]$,又因为$F[l] > F[k]$，所以$\sum_{i=1}^{k} P[i] + F[k] < \sum_{i=1}^{l} P[i] + F[l]$，这与作业$k$为完成时间最晚的作业相矛盾，所以最晚完成的作业的后面不可能存在一个F值更大的作业。

2.6.2 证明2

假设作业序列$J_1, J_2, \ldots, J_n$满足按F值非增序排列，现在按照这个进行作业调度。若此时出现另一个作业k，使得存在存在这种情况：$F[l] \le F[k] < F[l+1], l \in [1, n]$，将它排在作业调度序列的最后，则其完成时间为$\sum_{i=1}^{n} P[i] + P[k] + F[k]$。很明显这个值可能会是最大值。现在查找有没有一个序列能够使得这个值变得小一些。

由证明1中可以得出，如果存在$F[j] < F[k]$,按照j, k进行的作业调度用时要大于按照k, j进行的作业调度用时。而由已知条件$F[l] \le F[k] < F[l+1], l \in [1, n]$，所以应该将作业$k$放在作业$l$之后，作业$l+1$之前执行。同理，不应该将作业$k$放在$F$值比$F[k]$大的作业之前执行。

下面考虑F值相等的情况。假设$F[l] = F[k]$，则按照k, l进行的作业调度最大用时为：$\sum_{i=1}^{l-1} P[i] + P[k] + P[l] + F[l]$ 按照l, k进行的作业调度最大用时为$\sum_{i=1}^{l} P[i] + P[k] + F[k] = \sum_{i=1}^{l-1} P[i] + P[l] + P[k] + F[k]$，很明显这两个值相等，所以若两个作业的$F$值相等，则其调度的先后顺序不影响最大用时的计算。

2.7 复杂度分析

- **时间复杂度:**
  排序的时间复杂度为$O(nlogn)$, 计算最优解的时间复杂度为$O(n)$,所以总的时间复杂度为：$T(n) = O(nlogn)$
- **空间复杂度:**
  额外使用的空间是常数量级的，即为$O(1)$。

3 是否存在子序列?

3.1 问题描述

Given two strings s and t, check if s is subsequence of t?

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, ” ace” is a subsequence of ” abcde” while ” aec” is not).

3.2 基本思想

如果字符串s的长度大于字符串t的长度，则字符串s不可能是字符串t的子序列，返回false。

否则，对字符串s的每一个字母s_i，每次都在字符串t的搜索串t'中搜索最靠前的与该字母相同的项t_j，如果在搜索串t'中没有找到相同项，则返回false；否则，更新搜索串t'为去掉t_j及其前半部分后剩下的子串，继续重复执行该方法，直到字符串s中的每一项都能在字符串t中按序对应为止。

3.3 伪代码

PROBLEM 3 Is Subsequence?

INPUT: Given two strings s and t

OUTPUT: check if s is subsequence of t

```
function IsSubsequence(s, t)
    n ← s.length()
    m ← t.length()
    if n > m then
        return false
    end if
    i, j ← 0
    for i = 0 → n do
        while j < m and s[i]! = t[j] do
            j++
        end while
        if j == m then
            break
        end if
        j++
    end for
    if i! = n then
        return false
    end if
    return ture
end function
```

3.4 贪心选择性质

每次都从字符串t的子串t'中选择最先与s_i匹配的项当做字符串t的子序列中的一项。

3.5 最优子结构性质

$$OPT(s[i\dots n],t[j\dots m])=\begin{cases}true & if\ i>n\\ OPT(s[i+1\dots n],t[k+1\dots m]) & if\ s[i]==t[k],k\in[j,m]\\ false & otherwise\end{cases}$$

第一个式子中，当$i>n$，意味着字符串s已搜索完毕且其为字符串t的一个子序列；

第二个式子中，k为$t[j\dots m]$中第一个与$s[i]$相同的字母的下标；

如果找不到这样的k，则说明字符串s不可能为字符串t的一个子序列。

3.6 正确性证明

对于$s[i\dots n],t[j\dots m]$ ，现在从字符串$t[j\dots m]$中寻找第一次$s[i]$出现的情况，设其下标为$k$fi$k\in[j,m]$，则再在从字符串$t[k+1\dots m]$中找满足$s[i+1\dots n]$的子序列。假如我们不选第一次$s[i]$出现的情况，选择第2,3,or...次$s[i]$出现时情况，记下标为l,很明显$k<l\leq m$。这样就需要在字符串$t[l+1\dots m]$中找满足$s[i+1\dots n]$的子序列。但是可能会出现这样的一种情况，字符串$s[i+1\dots h],h\in[i+1,n]$只能在$t[k+1\dots l]$找到子序列，而无法在$t[l+1\dots m]$中找到子序列，这样就可能出现错误的结果。

3.7 复杂度分析

- **时间复杂度:**
  获取字符串s和t的长度的时间复杂度分别为$O(n)\Delta O(m)$，检测是否是子序列的过程时间复杂度为$O(n+m)$，因此总的时间复杂度为$O(n+m)$。

- **空间复杂度:**
  额外使用的空间是常数量级的，即为$O(1)$。