数据结构与算法第五次作业

Shuxin Chen, 2015013229

2017 年 4 月 29 日

1. 通过由顶点覆盖问题对其进行归约，证明：集合覆盖问题的判定版本是 NP 完全的。

   我们把集合覆盖问题重新表述为一个判定问题，即确定一个实例 $(X, \mathscr{F})$ 是否具有一个给定规模为 k 的集合覆盖。作为一种语言，我们定义

   SET-COVER $= \{\langle X, \mathscr{F}, k\rangle : X$ 中有一个规模为 k 的集合覆盖 $\}$

   首先我们证明 SET-COVER $\in$ NP。给定一个实例 $(X, \mathscr{F})$ 以及一个包含集合覆盖的集合 $\mathscr{F}_0 \in \mathscr{F}$，容易验证 $|\mathscr{F}_0| = k$。对于每一个在 X 中的元素，打上一个标记，依次遍历 $\mathscr{F}_0$ 中的集合，将每一个集合包含的元素的标记删除。最终如果所有 X 中的元素均没有标记，那么说明这个实例是正确的。显然这可以在多项式时间之内完成，因此我们证明了 SET-COVER $\in$ NP。

   接着我们通过证明 VERTEX-COVER $\leq_P$ SET-COVER 来证明集合覆盖问题是 NP 难度的。给定任意一个图 $G = (V, E)$，我们构造一个集合覆盖的实例 $(X, \mathscr{F})$，其中 $|X| = |E|$，即将图中的每一条边抽象成一个 X 中的元素；并且 $|\mathscr{F}| = |V|$，即将图中的每一个顶点抽象成一个 $\mathscr{F}$ 中的集合，设顶点 V_0 抽象成了集合 $\mathscr{F}_0$，那么 X 中的元素 $ELEM \in \mathscr{F}_0$ 当且仅当 $ELEM$ 对应的图中的边 E_{ELEM} 与 V_0 相关联。这样的归约是比较直观的，它将一个图中顶点与边的关系做了一个“对称”。

   下面来说明这确实是一个归约过程，即 $G = (V, E)$ 有一个 k 顶点覆盖当且仅当 $(X, \mathscr{F})$ 有一个 k 集合覆盖。接下来的证明过程及推导都是双向（即可逆）的，因此我们只证明从左边到右边的情况。

   若 $G = (V, E)$ 有一个 k 顶点覆盖，设我们选的点集合为 V'，$V' \in V$ 且 $|V'| = k$。在 $(X, \mathscr{F})$ 中，我们选取 V' 中每一个顶点对应的集合，这些集合的集合记为 $\mathscr{F}'$，$\mathscr{F}' \in \mathscr{F}$ 且 $|\mathscr{F}'| = k$。因为 V' 是一个顶点覆盖，那么 $\forall E_i \in E$，$\exists V_0 \in V'$ 使得 E_i 与 V_0 相关联，那么对于 E_i 抽象成的在 X 中的元素 $ELEM_i$，和 V_0 抽象成的在 $\mathscr{F}'$ 中的集合

$\mathscr{F}_0$，有 $ELEM_i$ 与 $\mathscr{F}_0$ 相关联。由 E_i 的任意性可知每个元素都与某一个 $\mathscr{F}'$ 中的集合相关联，因此 $\mathscr{F}'$ 是一个 k 集合覆盖。

综上，我们证明了 SET-COVER 问题是 NP 完全的。

2. 说明如何实现 GREEDY-SET-COVER，使其运行时间为 $O\left(\sum\limits_{s\in\mathscr{F}}|S|\right)$。

   我们首先对每一个 X 中的元素建立一个链表，链表中存储这个元素和集合的关联情况，若元素 $ELEM$ 与集合 $\mathscr{F}_0$ 相关联，那么 $ELEM$ 的链表中存放着 $\mathscr{F}_0$ 的引用。这一部分的构造需要依次遍历每一个集合中的每一个元素，时间复杂度为 $O\left(\sum\limits_{s\in\mathscr{F}}|S|\right)$。

   然后我们继续维护一个数组 sz，数组的大小为 $\max\limits_{s\in\mathscr{F}}|S|$。这个数组的每一个位置都存放着一个链表的指针，位置 i 存放着**当前**大小为 i 的所有集合的引用。同理，每一个集合都存放着它们在这个数组链表中的位置。这一部分的时间复杂度为 $O(\max\limits_{s\in\mathscr{F}}|S|)$。

   最后我们再使用一个数组 $flag$，它表示每一个元素是否被覆盖。

   上述的两个步骤均为预处理。我们接下来开始执行贪心算法。我们倒序访问数组 sz，每一次可以将当前大小最大的那个集合提取出来（如果有相同大小的，就按照链表中存放的次序依次提取）。设当前取出的集合为 $\mathscr{F}_1$，我们访问 $\mathscr{F}_1$ 中每一个未被覆盖的元素 $ELEM_1$，得到它对应的关联链表 $LINK_1$，再依次访问 $LINK_1$ 中的每一个集合，将这些集合的**当前**大小减去 1（即元素 $ELEM_1$ 已经被覆盖，其它包含它的集合拥有的未覆盖元素数量都要减去 1）。此时，这些集合在 sz 中的位置也应该发生变化，由于我们已经存放了每一个集合在数组链表中的位置，因此数组链表的维护操作（添加和删除，这里是将位置 i 的某个集合删除，再添加到位置 $i-1$）的时间复杂度均为 $O(1)$。贪心算法的时间复杂度可以根据访问集合的次数来估计，对于每一个元素，它有且仅有被访问一次（由数组 $flag$）保证，此时会访问所有和它相关联的集合。从另一个角度来考虑。对于每一个集合，它会被它其中的每一个元素有且仅有修改一次，因此时间复杂度为 $O\left(\sum\limits_{s\in\mathscr{F}}|S|\right)$。

   将上述三部分的时间复杂度相加，由于，$\max\limits_{s\in\mathscr{F}}|S| \leq \sum\limits_{s\in\mathscr{F}}|S|$，因此总的时间复杂度为 $O\left(\sum\limits_{s\in\mathscr{F}}|S|\right)$。

3. 节省矩阵乘法中的临时空间

(a) 给出改进算法的伪代码。

```
P-MATRIX-MULTIPLY(A, B, C)
n = A.size
if n == 1
    C11 = A11 + B11
else
    partition A, B, C into n/2 × n/2 sub-matrices:
    A11, A12, A21, A22, B11, B12, B21, B22, C11, C12, C21, C22
    spawn P-MATRIX-MULTIPLY(C11, A11, B11)
    spawn P-MATRIX-MULTIPLY(C12, A11, B12)
    spawn P-MATRIX-MULTIPLY(C21, A21, B11)
    spawn P-MATRIX-MULTIPLY(C22, A21, B12)
    sync
    spawn P-MATRIX-MULTIPLY(C11, A12, B21)
    spawn P-MATRIX-MULTIPLY(C12, A12, B22)
    spawn P-MATRIX-MULTIPLY(C21, A22, B21)
    spawn P-MATRIX-MULTIPLY(C22, A22, B22)
    sync
return C
```

(b) 工作量：$T_1(n) = 8T_1(n/2) + \Theta(1)$，解得 $T_1(n) = \Theta(n^3)$

持续时间：$T_\infty(n) = 2T_\infty(n/2) + \Theta(1)$，解得 $T_\infty(n) = \Theta(n)$

(c) 并行度：$\frac{T_1(n)}{T_\infty(n)} = \frac{\Theta(n^3)}{\Theta(n)} = \Theta(n^2)$

当 $n = 1000$ 时，并行度为 $1000000 = 10^6$，而原始算法的并行度为 10^7。由于绝大多数并行计算机的处理器数目也小于 100 万，因此改进算法和原始算法在并行度上比较几乎没有区别。