The Theory of Computation

Bernard M.E. Moret

Addison-Wesley-Longman, 1998

Solution Manual
Chapter 7, Problem 7.55

This problem offers a small glimpse into a large area of structure theory: the relationship between the complexity of problems and their density (or sparseness), that is, the number of distinct instances of the same size in the set. By its nature, a tally set is at one extreme of the density spectrum: it has at most one string of each length, since each element of a tally set is entirely determined by its length. Thus the exercise asks to show that very sparse sets cannot be NP-complete (unless, of course, P equals NP). We prove that, if a tally set is NP-complete, then we must have P = NP. The proof is somewhat tricky, although its basic engine is the self-reducibility of an NP-complete problem, here *SAT*: we shall consider partial truth assignments and use the induced versions of *SAT*, storing some partial results in order to run the entire algorithm in polynomial time.

Our algorithm will be the following self-reduction, where t_i denotes a partial truth assignment wherein the first i variables are set (the ordering of the variables is fixed):

```
if i=n
   then if the induced instance is empty
            then return "yes", else return "no"
look up in the table at index h(t_i)
if a value is found for h(t_i), return it
otherwise assign "true" to the (i+1)st variable and recurse;
if the returned valued was "no"
   then assign "false" to the (i+1)st variable and recurse;
if either recursive call returned "yes"
   then return "yes", else return "no"
in either case, store the returned value under h(t_i)
```

The function h acts much as a hashing function; its goal is to avoid unnecessary evaluation of subfunctions that have already been evaluated. Thus, whenever $h(t_i) = h(t'_j)$, it must be the case that the two induced instances are either both satisfiable or both unsatisifiable. It is also desirable that h be a good hash function in that its range is (very) small compared to its domain. This is where the reduction to a tally set comes in: the mapping used in the reduction will provide just the h we need.

Let f be a mapping from SAT to our tally set T. We can assume that f maps any instance of SAT to a string in $\{a\}^*$ (otherwise, we can remap any string that uses other symbols into a fixed string in $\{a\}^* - T$). Because f runs in polynomial time, the strings produced by f are at most polynomial in length; for an original instance with n variables, any string produced has length $O(n^{O(1)})$, which is very small compared to the number of possible instances of SAT on n variables. Finally, because f is a valid reduction, if must be the case that, whenever it maps two instances of SAT to the same string, then the two instances are both "yes" instances or are both "no" instances. Thus f meets the requirements placed upon h in the self-reduction procedure described above.

It remains to show that the self-reduction procedure described above, with the mapping f used as hash function, runs in polynomial time. Each recursive call takes polynomial time: its most expensive component is the computation of $h(t_i)$, the polynomial-time mapping from SAT to T. Thus we need to bound the number of recursive calls. Given an instance with n variables, our hash table will store at most $p(n)$ values, for some polynomial p. Note that the recursive calls form a binary tree of calls of depth at most n; denote the number of nodes in this tree by $S(n)$. We want to show that $S(n)$ is $O(n^{O(1)})$. We do this rather indirectly by showing that we can pick a subset of internal nodes (not leaves) of the tree of size at least $S(n)/2n$ and such that no two nodes are on the same path in the tree (or, equivalently, such that any two nodes give rise to disjoint sets of possible truth assignments). We can construct such a set by first removing all leaves of the tree (thus leaving $S(n)/2$ nodes), then iteratively applying the following step: select any new leaf node in the tree, add it to the set, and remove it and all of its ancestors from consideration. Since we remove at most n nodes in each iteration and start with $S(n)/2$ nodes, it will take at least $S(n)/2n$ iterations to delete every node in the tree and thus our set will have at least $S(n)/2n$ elements. By construction, none of these elements is a leaf of the original tree; also by construction, no prefix of a selected partial truth assignment can be in the set. We claim that each element of the set must be mapped to a different table location by our function h. Assume that such was not the case and let t_i and t_j be two such elements. Since t_i and t_j are on different paths on the tree, one was generated after the recursive call to the other had completed and returned; but then, since the two have the same h value, the second one would have succeeded in the table look-up and thus would have been a leaf in the original tree—a contradiction. Thus our table has size at least $S(n)/2n$; but we already noted that it has size at most $p(n)$; it follows that we have $S(N)/2n \leq p(n)$ or $S(n) \in O(n^{O(1)})$, as desired.