Lecture 13: Part I Strong NP-Completeness and Pseudo-Polynomial Time Algorithms

Informally, a decision problem π is strongly NP-complete if the language corresponding to a "unary" encoding of it, which is reasonable except that each integer k is recorded using k +1 symbols instead of O(log k) symbols, is NP-complete (to be very precise we would want to specify more exactly the encoding we were considering).

A pseudo-polynomial time algorithm for π runs in time which is polynomial in the size of the input under such an encoding. Such an algorithm shows that the language corresponding to a unary encoding of π is in P.

We showed that TSP is strongly NP-complete by reducing Hamilton Cycle to our special encoding of TSP (see page 479 of Kleinberg and Tardos). Since the only numbers in the TSP instance are 1 and 2, we see that we can think of this reduction as a reduction to a unary encoding, as even under the unary encoding we use at most 3 symbols for any of our integers.

In contrast, we presented a pseudo-polynomial time algorithm for Subset Sum. Which is that given in Section 6.4 of the text. We noted that we could also use this algorithm to solve Knapsack, as discussed in Section 6.4.

Lecture 13: Part 2 Dealing With NP-Complete Problems.

Determining that a problem is NP-Complete does not make it go away. Much of the rest of the course is devoted to studying how to handle NP-complete problems algorithmically. Approaches we investigate will include:

(1) algorithms which run in polynomial time and give approximately optimal solutions,

(2) algorithms which while they do not run in polynomial time on all inputs, run in polynomial time on average, and

(3) algorithms which run in polynomial time for a restricted class of inputs.

Pseudo-polynomial time algorithms are an example of the third kind of algorithm, they run in polynomial time on those inputs for which the value of the integers in the instance are bounded by a polynomial in the (classical not unary) input size.

For some strongly NP-Complete problems, we can get polynomial time algorithms by imposing even more draconian restrictions on the input. Thus, for example, while Clique is strongly NP-complete, if we insist that the size k of the clique we are looking for is a constant then a brute force algorithm which checks for each set S of k vertices whether or not all the edges between these vertices are present solves the problem in $O(n^k)$ time. Thus the decision problem k-Clique in which an instance is a graph and we are asked if it contains a clique of size k, can be solved in polynomial time. In contrast 3-Colourability is NP-complete even though this is a restriction of the general Colourability problem to the 3 colour case.

Other problems can be solved quickly if we restrict our attention to specially structured inputs. For example, as discussed in an earlier lecture and in Section 10.2 of Kleinberg and Tardos, maximum weight stable set can be solved in linear time in trees even though it is NP-complete in general graphs.

We then turned to approximation algorithms for optimization problems. We observed that we can greedily construct in linear time a maximal matching which must be within a factor of 2 of optimal. We then discussed an approximation algorithm for Knapsack. We had noted earlier in the lecture that we could swap the roles of weight and value in the pseudo-polynomial dynamic programming algorithm for Knapsack. Rather than recording the highest value subset of each given weight in the table, we could record the lowest weight subset of each given value. Using this algorithm and rounding the values we obtained the approximation algorithm for Knapsack discussed in section 11.8 of Kleinberg and Tardos.