Problem-set 03

Mostafa Touny

Contents

lectures 5, 6, 7 sipser 9.1, 7.2

Ex. 1

The proof is already mentioned in sipser. We can easily reprove it using the diagonalization argument.

Ex. 2

Enumerate all the two choices of a node colored in (red or blue), or colored in yellow, on all nodes. Consider the two subgraphs separately; If the yellow subgraph contains any edge reject the instance. Check if the other subgraph is 2-colorable. Only if yes, accept as the whole graph as 3-colorable.

The complexity is justified, Since checking 2-colorable is polynomially solved, on each instance of all two choices, on all nodes.

Clearly, If the graph is 3-colorable, then the algorithm catches the instance corresponding to nodes correctly colored in yellow and others in either red or blue. On the other hand, If the algorithm found a solution, Then the graph is 3-colorable, As the solution can easily be constructed.

Ex. 3

Observe the cases of x_i and x_j of the binary relation $x_i \leq x_j$.

- (1) Both are assigned by a given condition
- (2) One is assigned 0 and the other is equal or less
- (2) One is assigned 1 and the other is equal or greater
- (3) Both are not assigned
- (4) One is assigned 0 and the other is equal or greater
- (4) One is assigned 1 and the other is equal or less

We give an algorithm that costs exactly one linear scan. Scan all binary relations $x_i \leq x_j$, If

- of type (1), Check whether assigned values conform to the relation, and reject satisfiability if not.
- of type (2), Assign 0 and 1, Correspondingly, So that values conform to the relation. If a conflict is faced, where there's a prior assignment, that precludes from assigning what satisfies the relation, reject.

To see why the algorithm is correct, We construct a valid solution, Given what the algorithm had already verified. For case - (3), assign $x_i = 0$ and x_j arbitrarily 0 or 1 - (4), assign arbitrarily 0 or 1

Clearly, cases 3 and 4, do not conflict with cases 1 or 2, Since the algorithm has already checked and assigned what satisfies cases 1 and 2. Case 3 doesn't conflict with 4, As 4 allows any assignment. Remaining x_i with no conditions at all, can be arbitrarily assigned as well.

Ex. 4

From *hw02-2-b*, We are given a procedure *haltMachine(T, f(n))* that produces a Turing Machine $T_{f(n)}$, which is exactly the same as machine T, but halts within $f(n)$ steps; If T terminates within $f(n)$, Then $T_{f(n)}$ produces the same output; Otherwise rejects. Note $T_{f(n)}$ is multi-tab, whereby at each step simulated of T, a counter on a specific tab is increased by one.

a

For any polynomial time machine T, We know it runs in time at most kn^k. T_{kn^k} simulates T upto kn^k steps which suffices to ensure it produces the same output.

b

Since T is polynomial, and at each step, counter increase is polynomial, The total resulting complexity is polynomial.

c

Observe to construct $T_{f(n)}$, We would need to integrate a sub-routine that increases counter by one, where every state points to it after completing its one-step operation. For every state q_i, we create a new state $q_{i-counter}$ such that - q_i transitions to $q_{i-counter}$, instead of q_r as in T, exactly after one-step. - $q_{i-counter}$ transitions to q_r, what q_i transitions to in T, after completing all steps required for increasing counter by one.

Clearly the transformation is linear in time.

d

Alice can basically check for the sub-routine that increases counter by one, and check for the state that terminates the machine, upon the counter reaching $f(n)$.

The algorithm of checking polynomiality is clear, if the reader is convinced by the procedure of transformation.