Model Answers Complexity Theory

tms41, dl516

last updated: 16.5.2023, 22:22

Question (Exercise Sheet 1, Question 1)**.** *In the lecture, a proof was sketched showing a $\Omega(n \log n)$ lower bound on the complexity of the sorting problem. It was also stated that a similar analysis could be used to establish the same bound for the Travelling Salesman Problem. Give a detailed sketch of such an argument. Can you think of a way to improve the lower bound?*

Answer. The computation tree with branches, binary decisions and $n!$ leaves is the same as with sorting. Also the derivation why $\log(n!) = \Theta(n \log n)$ can be found in the lecture notes of Complexity Theory [page 5]. What remains to be verified that indeed all $n!$ leaves are needed. Each leaf corresponds to a particular permutation ρ of $\{1, 2, \ldots, n\}$, and it suffices that for each particular permutation there is an input graph so that the only optimal tour is ρ.

Actually, due to the nature of the TSP, every tour is equivalent to $n-1$ other tours. This means that we in fact only need $(n-1)!$ different branches (each branch corresponds to the selection of a cycle of length n). Now for each i, j so that i and j are adjacent on the cycle, define the costs $c(i,j) := 1$ and $c(j,i) := 1$. All other costs are defined as $c(i,j) := 2$. It is clear that for this particular TSP instance, only the TSP tour (=cycle) which is formed by the 1-cost edges is optimal, and all other TSP tours are sub-optimal. Hence any correct TSP algorithm must have at least $(n-1)!$ different leaves.

To improve the argument, note there is a crucial difference between the input to the sorting problem and the input to the TSP problem in that the TSP problem requires $\binom{n}{2}$ different edge costs (so it is order $\Theta(n^2)$, as opposed to an input of size n for the sorting problem). Now consider as possible TSP inputs all possible graphs with a cost function so that exactly one edge has cost 1 and all other edges have cost 2. This means that in order to find the minimum TSP tour, the TSP algorithm has to identify the 1-cost edge, which clearly requires scanning through all edges, so we get a lower bound of $\Omega(\binom{n}{2}) = \Omega(n^2)$. □

Question (Exercise Sheet 1 Question 3)**.** *Consider the language* UNARY-PRIME *in the one letter alphabet $\{a\}$ defined by* UNARY-PRIME $= \{a^n \mid n \text{ is prime}\}$*. Show that this language is in P.*

Answer. We can simply check whether any of the numbers $2, \ldots, n-1$ divides n. The school division algorithm can be implemented in $\mathcal{O}(n^2)$ time. Hence, the entire check can be done in $\mathcal{O}(n^3)$ time, which is polynomial in the input size n.

Note 1: We can improve the running time to $\mathcal{O}(\sqrt{n} \cdot \text{polylog}(n))$, by checking divisors $2, \ldots, \lfloor \sqrt{n} \rfloor$ (we don't need to check larger values) and using an efficient $\mathcal{O}(\log n (\log \log n))$ time for division.

Note 2: We can even use one of the polynomial time algorithms for the binary prime problem (which is in P) to get an even better time complexity.

□

Question (Exercise Sheet 1 Question 4)**.** *Suppose $S \subseteq \mathbb{N}$ is a set of natural numbers and consider the language* UNARY-S *in the one letter alphabet $\{a\}$ defined by* UNARY-$S = \{a^n \mid n \in S\}$*, and the language* BINARY-S *in the two letter alphabet $\{0, 1\}$ consisting of those strings* *starting with a* 1 *which are the binary representation of a number in* S*. Show that if* UNARY-S *is in* P *then* BINARY-S *is in* $\mathsf{Time}(2^{cn})$ *for some constant* c*.*

Answer. We can convert a number with n bits from its binary representation to its unary representation by just appending a's (at most 2^{2n} of them) in $\mathcal{O}(2^{2n})$ time. Since UNARYS is in P, for any input of length k we can determine if it is in the language in time $\mathcal{O}(k^c)$ for some constant $c > 0$. Hence, we can determine if $n \in$ BINARYS in time $\mathcal{O}((2^{2n})^c) = \mathcal{O}((2^{2cn})$. □

Question (Exercise Sheet 1 Question 5). *We say that a propositional formula* ϕ *is in* 2-CNF *if it is a conjunction of clauses, each of which contains exactly* 2 *literals. The point of this problem is to show that the satisfiability problem for formulas in* 2-CNF *can be solved by a polynomial time algorithm.*

First note that any clause with 2 *literals can be written as an implication in exactly two ways. For instance* $(p \vee \neg q)$ *is equivalent to* $(q \to p)$ *and* $(\neg p \to \neg q)$*, and* $(p \vee q)$ *is equivalent to* $(\neg p \to q)$ *and* $(\neg q \to p)$*.*

For any formula ϕ*, define the directed graph* G_ϕ *to be the graph whose set of vertices is the set of all literals that occur in* ϕ*, and in which there is an edge from literal* x *to literal* y *if and only if, the implication* $(x \to y)$ *is equivalent to one of the clauses in* ϕ*.*

(a) *If* ϕ *has* n *variables and* m *clauses, give an upper bound on the number of vertices and edges in* G_ϕ

(b) *Show that* ϕ *is unsatisfiable if, and only if, there is a literal* x *such that there is a path in* G_ϕ *from* x *to* $\neg x$ *and a path from* $\neg x$ *to* x*.*

(c) *Give an algorithm for verifying that a graph* G_ϕ *satisfies the property stated in (b) above. What is the complexity of your algorithm?*

(d) *From (c) deduce that there is a polynomial time algorithm for testing whether or not a* 2-CNF *propositional formula is satisfiable.*

(e) *Why does this idea not work if we have* 3 *literals per clause?*

Proof. (a) For each variable x we have two vertices x and $\neg x$. So there can be at most $2n$ vertices. For each clause there can be at most two new edges added, hence a total of at most $2m$.

(b) ($\Rightarrow$) If there is a path from x to $\neg x$ ($x \to p_1 \to \ldots \to p_k \to \neg x$) and a path from $\neg x$ to x ($\neg x \to q_1 \to \ldots \to q_k \to x$), then we will show that there is no satisfying assignment for x:

- **Case 1** [$x = \texttt{T}$]**:** Then by induction each of $p_1, \ldots, p_k, \neg x$ must be `T` and so $x = \neg x$ which is a contradiction.
- **Case 2** [$x = \texttt{F}$]**:** Then by induction each of $q_1, \ldots, q_k, x$ must be `T` and so $x = \neg x$ which is a contradiction.

($\Leftarrow$) This direction is slightly more tricky. Assume that there is no path from $x \to^* \neg x$ and $\neg x \to^* x$ for any variable x. We will now construct a satisfying assignment for this graph.

We start with the claim that if there is a path $p_1 \to \ldots \to p_k$ in the graph, then there is also the path $\neg p_k \to \ldots \to \neg p_1$ (where double negation gives the original variable). This follows from the fact that when $a \to b$ is present in the graph, then so is $\neg b \to \neg a$.

Consider the strongly connected components (SCC) of the graph, i.e., the equivalence classes formed by the bidirectional reachability relation. These form a directed acyclic graph with some of the components being potentially disconnected. We proceed for each unassigned variable and assign the truth value of the component to `T` (and automatically the truth value of the component of its negation to `F`).

We will now argue that this gives a valid truth assignment. To show this we need that there is no path $y \to^* x$ and $y \to^* \neg x$, i.e., that x and $\neg x$ cannot be in the same component. This follows from the above observation since a path from $y \to^* x$ would also imply a path from $\neg x \to^* y$, and so a path from $x \to^* \neg x$, which by assumption does not exist.

(c) A simple algorithm for performing this check is to start a BFS or DFS from each vertex x and check if we can reach $\neg x$. This requires $\mathcal{O}(n \cdot (n+m))$ time. An $\mathcal{O}(n+m)$ time algorithm can be obtained by computing the SCCs in $\mathcal{O}(n+m)$ time and checking if any variable x is in the same component as its negation.

(d) Both of the above algorithms run in time polynomial to the size of the input.

(e) The above idea does not work because if we know that x is `F` in the clause $(x \vee y \vee z)$, then we cannot decuce anything about y and z individually.

□

Question (Exercise Sheet 1 Question 7)**.** *We define the complexity class of quasi-polynomial-time problems* Quasi-P *by:*

$$\text{Quasi-P} = \bigcup_{k=1}^{\infty} \text{Time}(n^{(\log n)^k}).$$

Show that if $L_1 \leq_P L_2$ and $L_2 \in$ Quasi-P, then $L_1 \in$ Quasi-P.

Answer. We need to design a Turing Machine M_1 such that $L(M_1) = L_1$, and every accepting computation is in quasi-polynomial-time.

By assumption, there is a Turing Machine M_2 with $L(M_2) = L_2$, and for every $x \in L_2$, the accepting computation finishes in time $O(n^{(\log n)^{k_2}})$ for some constant $k_2 \geq 1$. Also there is a polynomial-time computable function $f : \Sigma_1^\star \to \Sigma_2^\star$ such that $x \in L_1$ if and only if $f(x) \in L_2$.

Given $x \in \Sigma_1^\star$, the Turing Machine M_1 first computes $f(x)$. Then it inputs $f(x)$ to the Turing Machine M_2. If M_2 accepts it, then M_1 accepts. Otherwise, M_1 rejects (or runs forever). We need to prove that $L(M_2) = L_2$ and for every accepting configuration, M_2 finishes in time $O(n^{(\log n)^k})$.

Let $L_1 \subseteq \Sigma_1^\star$ and $x \in \Sigma_1^\star$ be of length n. First, assume $x \in L_1$. Then $f(x) \in L_2$ and by definition of M_2, $M_2(f(x))$ accepts. Furthermore, the length of $f(x)$ is $O(n^c)$ for some constant $c > 0$. Hence M_2 will finish the computation in time

$$O((n^c)^{(\log n^c)^{k_2}}) = O(n^{c \cdot c^{k_2} \cdot (\log n)^{k_2}}) \leq O(n^{\log n \cdot (\log n)^{k_2}}) = O(n^{(\log n)^{k_2+1}}).$$

Hence the total time for M_1 is

$$n^d + O(n^{(\log n)^{k_2+1}}) = O(n^{(\log n)^{k_2+1}}),$$

where $O(n^d)$ for some constant $d \geq 1$ is the time used to compute the function f.

Next assume $x \notin L_1$. Then $f(x) \notin L_2$, and $M_2(f(x))$ does not accept (i.e., rejects or runs forever). This proves that $L(M_2)) = L_2$, and also that every accepting computation is in quasi-polynomial time. Therefore, $L_1 \in$ Quasi-P. □

Question (Exercise Sheet 1 Question 8)**.** *In general k-colourability is the problem of deciding, given a graph $G = (V, E)$, whether there is a colouring $\chi : V \to \{1, \ldots, k\}$ of the vertices such that if $\{u, v\} \in E$, then $\chi(u) \neq \chi(v)$. That is, adjacent vertices do not have the same colour.*

1. *Show that there is a polynomial time algorithm for solving 2-colourability.*
2. *Show that, for each k, k-colourability is reducible to $(k+1)$-colourability. What can you conclude from this about the complexity of 4-colourability?*

Answer. First recall that as explained on page 29 of the notes, k is fixed and not part of the input.

1. A polynomial-time algorithm for 2-colourability can be designed based on, e.g., BFS and colouring each vertex that is explored from the current vertex alternately. (Details and a formal proof are omitted here, but a formal correctness proof would need to exploit that a graph is 2-colourable if and only if the graph is bipartite (which is equivalent to having no cycles of odd length).

2. Given any graph $G = (V, E)$, we need to give a polynomial-time[1] construction of another graph G' (depending on G) such that G is k-colourable if and only if G' is $(k+1)$-colourable. To this end, let $G = (V, E)$ be given. Construct $G' = (V', E')$ by adding a single vertex z, i.e., $V' = V \cup \{z\}$. With regards to the edges, keep all edges in G and additionally connect z to all other vertices, i.e., $E' = E \cup \{\{z, u\} : z \in V\}$ (see Figure 1 for an illustration).

   First, we note that it is clear that the construction of G' can be done in time polynomial in the size of the input (which is the representation of the graph G). Secondly, we will prove the equivalence. First assume G is k-colourability. Then, by colouring z with an extra colour, we obtain a colouring of G' with $k + 1$ colours. For the other direction, assume G' is $(k + 1)$-colourable. Then, since z is connected to all other vertices, its colour must be unique. Hence for the set $V' \setminus \{z\} = V$ only k colours are used, and since the colouring is valid for G', it follows that the same colouring also works for V, proving that G is k-colourable.

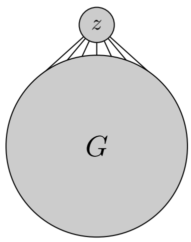

Figure 1: Illustration of the construction of G'.

   Regarding the additional question about the complexity of 4-colourability, it was shown in the lectures that 3-colourability is NP-complete (which means it is NP-hard, i.e., any problem in NP can be reduced to it, and additionally, it is also in NP). By the polynomial-time reduction above, we can reduce every problem in NP first to 3-colourability, and then to 4-colourability. This proves that 4-colourability is NP-hard. It also easy to see that 4-colourability is in NP (for example, simply guess non-deterministically a 4-colouring).

□

[1]The question did not explicitly ask for a polynomial time reduction, but this is needed for our conclusion about the complexity of 4-colourability.