Computational Complexity

Exercise Session 4

Note: these solutions are (often) merely pointers to the right idea that is needed to solve the problems. These are not fully worked-out solutions. So please do not take these solutions as an example for how to write up your solutions for, e.g., the homework assignments. :-)

Note: These exercises are (likely) too much work to solve all during the exercise session.

Exercise 1. Prove that $\mathsf{L} \subseteq \mathsf{P}$.

Solutions:

- You can use Theorem 4.2 from the book, for example. Since $\mathsf{SPACE}(S(n)) \subseteq \mathsf{DTIME}(2^{O(S(n))})$, taking $S(n) = \log n$, we get the following. Take some $Q \in \mathsf{L}$. Then $Q \in \mathsf{SPACE}(\log n)$. So $Q \in \mathsf{DTIME}(2^{c \log n}) = \mathsf{DTIME}(n^c)$ for some c, and thus $Q \in \mathsf{P}$.
- Alternatively, you can argue that for any problem $Q \in \mathsf{L}$, and for each input x, the configuration graph of a machine deciding whether $x \in Q$ in logarithmic space is of polynomial size in x (and can be constructed in polynomial time), and then checking whether an accepting state is reachable in this graph from the initial configuration is a polynomial-time solvable problem. Thus, this gives a polynomial-time algorithm to solve Q.

Definition 1. We define DP to be the following complexity class:

$$\mathsf{DP} = \{ \ A \cap B \mid A \in \mathsf{NP}, B \in \mathsf{coNP} \ \}.$$

Exercise 2.

(a) Explain the difference between DP and the class $\mathsf{NP} \cap \mathsf{coNP}$.

(b) Prove that $\mathsf{NP} \cup \mathsf{coNP} \subseteq \mathsf{DP}$.

(c) Prove that $\mathsf{P} = \mathsf{DP}$ if and only if $\mathsf{P} = \mathsf{NP}$.

(d) Prove that the following problem $\mathsf{MAX\text{-}SAT}$ is in DP:

> **Instance:** A propositional formula φ in CNF and a positive integer $k \in \mathbb{N}$.
>
> **Question:** Is the maximum number of clauses of φ that can be (simultaneously) satisfied by a truth assignment α exactly k?

Solutions:

(a) Problems in DP are the **intersection of two problems**: a problem $A \in \mathsf{NP}$ and a problem $B \in \mathsf{coNP}$. So by taking $A = \{0,1\}^*$ or $B = \{0,1\}^*$, which is both in NP and in coNP, we get that $\mathsf{NP} \subseteq \mathsf{DP}$ and $\mathsf{coNP} \subseteq \mathsf{DP}$. Problems in $\mathsf{NP} \cap \mathsf{coNP}$ are in the **intersection of the classes**, and thus $\mathsf{NP} \cap \mathsf{coNP} \subseteq \mathsf{NP}$, and $\mathsf{NP} \cap \mathsf{coNP} \subseteq \mathsf{coNP}$.

(b) Let $A \in \mathsf{NP}$. Then take $B = \{0,1\}^*$. Then $B \in \mathsf{coNP}$. Therefore $A \cap B \in \mathsf{DP}$. Since $B = \{0,1\}^*$, we know that $A \cap B = A$, and thus $A \in \mathsf{DP}$. In a similar way we can show that $\mathsf{coNP} \subseteq \mathsf{DP}$, and thus $\mathsf{NP} \cup \mathsf{coNP} \subseteq \mathsf{DP}$.

(c) Suppose that $\mathsf{P} = \mathsf{DP}$. Since $\mathsf{P} \subseteq \mathsf{NP}$, and $\mathsf{NP} \subseteq \mathsf{DP}$, then $\mathsf{P} = \mathsf{NP}$. Conversely, suppose that $\mathsf{P} = \mathsf{NP}$. Then also $\mathsf{P} = \mathsf{coNP}$. We show that $\mathsf{P} = \mathsf{DP}$. Take an arbitrary problem $L \in \mathsf{DP}$. Then $L = A \cap B$ for some $A \in \mathsf{NP} = \mathsf{P}$ and $B \in \mathsf{coNP} = \mathsf{P}$. Since $A \in \mathsf{P}$ and $B \in \mathsf{P}$, then also $L = A \cap B \in \mathsf{P}$. Then, since L was arbitrary, we know that $\mathsf{P} = \mathsf{DP}$.

Exercise 3. Consider the problem SINGLE-CYCLE, where the input consists of an undirected graph $G = (V, E)$, and the question is to decide if G consists of one single cycle. Show that SINGLE-CYCLE is in L.

Solutions:

The algorithm works in two phases. In the first phase, it checks whether each edge in G is connected to exactly two other edges in G. In the second phase, it checks whether every pair of edges is reachable from each other through the edges in the solution.

Phase 1: To check the first property, it iterates over all edges in the graph. For each edge e, it counts the number of edges that share at least one vertex with e (including e itself). This it can do by iterating over all edges, and comparing each edge e' with e. For each edge, it checks if this count is exactly 3. It returns "yes" for this phase if this check passes for all edges.

Phase 2: To check the second property, it iterates over all pairs of edges in the graph. For each pair of edges e_1, e_2 it checks whether e_2 is reachable from e_1. It does so in the following way: it starts at e_1, and finds an edge e that is connected to e_1 (picking an arbitrary one if there are two). Then, it remembers the current edge (initially e) and the previous edge (initially e_1). Then, it keeps finding a new edge connected to the current edge, that is not equal to the previous edge—and stores the current edge as the (new) previous edge, and the newly found edge as the (new) current edge. If it reaches e_2 at some point, it concludes that e_2 is reachable from e_1. If it reaches e_1 again at some point, in concludes that e_2 is not reachable from e_1. This way, it checks for each pair of edges whether they are reachable from each other. It returns "yes" for this phase if this check passes for all pairs of edges.

Both of these phases can be seen as some nested for-loops, iterating over all edges. The algorithm keeps the current value of the variables in these for loops in memory (on the tapes). Since the number of edges are polynomial in the size of the input, storing a single one (or a constant number) of them costs only logarithmic space.