Seminar Theoretical Computer science. April 2nd, 2018.

If L is a language over an alphabet Σ, then L^{c} represents the complement $\Sigma^* \setminus L$. Let $\text{co}\mathsf{NP} = \{L^{\mathsf{c}} \colon L \in \mathsf{NP}\}$.

The class coNP

Exercise 1. Prove that if $\mathsf{NP} \neq \text{co}\mathsf{NP}$ then $\mathsf{P} \neq \mathsf{NP}$.

Exercise 2. Show that if $\mathsf{NP} \subseteq \text{co}\mathsf{NP}$ then $\mathsf{NP} = \text{co}\mathsf{NP}$.

Exercise 3. Suppose $L_1, L_2 \in \mathsf{NP} \cap \text{co}\mathsf{NP}$. Show that $L_1 \oplus L_2 = \{x : x \text{ is in exactly one of } L_1, L_2\}$ is in $\mathsf{NP} \cap \text{co}\mathsf{NP}$.

Exercise 4. We say that a language L is coNP-complete if (1) $L \in \text{co}\mathsf{NP}$ and (2) $A \leq_p L$ for all $A \in \text{co}\mathsf{NP}$. Show that 3SAT^{c} is coNP-complete.

The class PSPACE

Exercise 5. Show that if $A \in \mathsf{PSPACE}$, then $A^* \in \mathsf{PSPACE}$.

Exercise 6. Two Boolean formulas are *equivalent* if they describe the same Boolean function. For example, the formulas $x_1 \wedge \overline{x}_2$ and $x_1 \wedge x_1 \wedge \overline{x}_2$ are equivalent. A Boolean formula is *minimal* if there exists no equivalent formula for which the number of $\wedge$ and $\vee$ symbols is strictly smaller. For example, the formula $x_1 \wedge x_1 \wedge \overline{x}_2$ is not minimal, but $x_1 \wedge \overline{x}_2$ is. Let MinFormula be the set of minimal Boolean formulas.

- Show that MinFormula $\in \mathsf{PSPACE}$.
- Explain why this argument fails to show that MinFormula $\in \text{co}\mathsf{NP}$: If $\phi \notin$ MinFormula, then ϕ has a smaller equivalent formula. A non-deterministic Turing machine can verify that $\phi \in \overline{\text{MinFormula}}$.

Exercise 7. Show that if TQBF $\in \mathsf{NP}$, then $\mathsf{NP} = \text{co}\mathsf{NP}$.

Exercise 8. The cat-and-mouse game is played by two players "Cat" and "Mouse" on an arbitrary graph. At a given moment of time, each player occupies a vertex of a connected graph. Players move in turns. At his turn, a player must move to a vertex that is adjacent to the current position.

A special vertex of the graph is called Hole. Cat wins if the two players ever occupy the same node. Mouse wins if it reaches the Hole before the preceding happens. The game is draw if a situation repeats (a situation is determined by the player's positions and the player's turn to move).

The problem Happy Cat:

Input:	A graph (V, E), vertices $c, m, h \in V$ (initial positions of Cat, Mouse and the Hole).
Question:	Does Cat have a winning strategy if Cat moves first?

Prove that this problem is in P.

PSPACE-completeness

Exercise 9. The problem in-space acceptance:

Input:	A Turing machine M and an input x.
Question:	Does M accept x without ever leaving the first $|x| + 1$ cells on the tape?

Prove that the problem in-space acceptance is in PSPACE. Prove that the problem is PSPACE-complete.

Exercise 10. Consider TQBF problem restricted to monotone formulas (no negations). Show (at least) one of the two: monotone-TQBF is in P; monotone-TQBF is PSPACE-complete.

Exercise 11. In this problem a graph (V, E) is given by a Boolean circuit ϕ. This means that V equals the set of m-bit strings for some m, and that two vertices x and y are connected if and only if $\phi(x, y) = 1$. Prove that the following problem is PSPACE-complete.

The problem succinct connectivity:

Input:	A graph G represented by a circuit and two nodes s and t.
Question:	Is there a path from s to t in G?

Solutions

Exercise 1. Recall that $L \in \mathsf{P}$ if and only if $L^{\mathsf{c}} \in \mathsf{P}$. We prove the contrapositive of the statement: if $\mathsf{P} = \mathsf{NP}$ then $\mathsf{NP} = \mathrm{co}\mathsf{NP}$. Assume $\mathsf{P} = \mathsf{NP}$. We need to show that $L \in \mathsf{NP} \Leftrightarrow L \in \mathrm{co}\mathsf{NP}$.

$$L \in \mathsf{NP} = \mathsf{P} \quad \Longleftrightarrow \quad L^{\mathsf{c}} \in \mathsf{P} = \mathsf{NP} \quad \Longleftrightarrow \quad L \in \mathrm{co}\mathsf{NP}.$$

Exercise 3. We prove that $L_1 \oplus L_2 \in \mathsf{NP}$. The proof that the language belongs to $\mathrm{co}\mathsf{NP}$ is similar. For $i \in \{1,2\}$, let V_i be a verifier for language L_i, and let C_i be a verifier for the languages L_i^{c}. Certificates for $x \in L_1 \oplus L_2$ consist of 3 parts:

- The number $i \in \{1,2\}$ that indicates whether $x \in L_1 \setminus L_2$ or $x \in L_2 \setminus L_1$.
- A certificate for the verifier V_i to prove that $x \in L_i$.
- A certificate for the verifier C_{3-i} to prove that $x \notin L_{3-i}$.

It is now easy to construct a verifier that satisfies the conditions.

Exercise 5. The following algorithm $\texttt{InAstar}(x)$ decides A^*. It works recursively in $|x|$. If $|x| = 0$, then accept. Otherwise, for all $k = 1, \ldots, |x|$ check whether the prefix $x_1 x_2 \cdots x_k$ belongs to A using a polynomial space algorithm, and whether the suffix $x_{k+1} \cdots x_{|x|}$ belongs to A^*, using a recursive call $\texttt{InAstar}(x_{k+1} \cdots x_{|x|})$. If this is true for some k, then accept, otherwise reject.

To see that this algorithm runs in polynomial space, note that each recursive call decreases the length of x by at least 1. Hence, the depth of the recursion stack is at most $|x|$. Because each recursive call requires at most polynomial space, storing the whole recursion stack requires polynomial space.

Second solution. The following nondeterministic algorithm decides A^*: nondeterministically guess a splitting of x, accept if each part belongs to A, and otherwise, reject. This shows that $A^* \in \mathsf{NPSPACE}$. Now the statement follows from Savitch's theorem.

Exercise 7. Assume $L \in \mathsf{NP}$. We show that $L \in \mathrm{co}\mathsf{NP}$. By NP-completeness of SAT, there exists a reduction from L to SAT. For any string w, this reductions maps a string w to a Boolean formula ϕ_w. Thus

$$w \in L \quad \Longleftrightarrow \quad \phi_w \in \mathrm{SAT} \quad \Longleftrightarrow \quad \exists x_1 \exists x_2 \ldots \exists x_m \, [\phi_w(x_1, \ldots, x_m)] \in \mathrm{TQBF}.$$

Thus

$$w \in L^{\mathsf{c}} \quad \Longleftrightarrow \quad \forall x_1 \forall x_2 \ldots \forall x_m \, [\mathrm{NOT}\,(\phi(x_1, \ldots, x_m))] \in \mathrm{TQBF}.$$

We have shown that L^{c} reduces to $\mathrm{TQBF} \in \mathsf{NP}$. Thus $L^{\mathsf{c}} \in \mathsf{NP}$ and hence $L \in \mathrm{co}\mathsf{NP}$. Because this is true for every $L \in \mathsf{NP}$, we conclude that $\mathsf{NP} \subseteq \mathrm{co}\mathsf{NP}$. By exercise 2 this implies $\mathsf{NP} = \mathrm{co}\mathsf{NP}$.

Exercise 8. Each configuration of the game is described by a triple $(C, M, p) \in V \times V \times \{\texttt{Cat}, \texttt{Mouse}\}$, where C is the position of the Cat, M is the position of the Mouse, and $p \in \{\texttt{Cat}, \texttt{Mouse}\}$ is the player that moves next. Note that the total number of configurations is $2|V|^2$, which is polynomial in the input length.

The idea of the algorithm is to use dynamic programming to compute a set W of all configurations for which Cat has a winning strategy. Initially, W contains all positions (C, M, p) for which $C = T$. Then we update W by performing the following actions simultaneously:

- Add all configurations $(C, M, \texttt{Cat})$ for which there exists a vertex $C' \in V$ such that $(C, C') \in E$ and $(C', M, \texttt{Mouse}) \in W$.
- Add all configurations $(C, M, \texttt{Mouse})$ such that for all edges $(M, M') \in E$, the configuration $(C, M', \texttt{Cat})$ belongs to W.

(In fact it does not matter whether we perform the actions simultaneously or subsequently, but in the former case the analysis is a bit easier.) We repeat this update until no new configurations are added to W. The algorithm accepts if $(c, m, \texttt{Cat}) \in W$ and rejects otherwise.

This algorithm terminates in polynomial time, because we update W at most $2|V|^2$ times. It remains to prove that W contains precisely the configurations in which Cat has a winning strategy. This is rather obvious, but we present the formal details for the interested reader. (This does not change the final value of W.)

We need some definitions. The *length* of a winning strategy for Cat for a configuration (C, T, p), is the maximal number of moves that are needed such that Cat and Mouse arrive in the same vertex, starting in the configuration (we count the total number of moves, i.e., the sum of the moves of Cat and Mouse). The *catching time* of a configuration is the minimal length of a winning strategy for Cat for the configuration. If a configuration has no winning strategy, its catching time is infinite. In the following claim, the 0th update of W corresponds to the initialization of W. Let $k \geq 0$.

In the kth update of W we add precisely those configurations to W that have catching time k.

We prove the claim by induction on k. The claim is true for $k = 0$, because the configurations with catching time 0 are precisely the configurations (C, M, p) with $C = M$, i.e., the elements of W at its initialization.

Now assume the claim is true for all $k = 0, 1, \ldots, K$. We prove that it is also true for $k = K + 1$. A configuration of type $(C, M, \texttt{Cat})$ has catching time $K + 1$ if and only if K equals the minimal catching time of $(C', M, \texttt{Mouse})$ for a neighbor C' of C. By the induction hypothesis, this is true if and only if the Kth update is the first update in which a configuration $(C', M, \texttt{Mouse})$ for some neighbor C' is added. Finally, this is true if and only if $(C, M, \texttt{Cat})$ is added during the $K + 1$th update.

A configuration $(C, M, \texttt{Mouse})$ has catching time $K + 1$ if and only if K equals to the maximal catching time of the configurations $(C, M', \texttt{Cat})$ for all neighbors M' of M. By the induction hypothesis, this is true if and only if the last such configuration is added during the Kth update. And by construction this is true if and only if the $(C, M, \texttt{Mouse})$ is added during the $K + 1$th update.

Exercise 9. To show that this problem is in PSPACE, consider the following algorithm with inputs M and x. The algorithm simulates M on input x and as soon as M reaches the $(|x| + 2)$nd cell, the simulation is terminated and the input is rejected. Otherwise, if the simulation halts, the input is accepted if the simulation accepts and rejected otherwise.

There is one remaining case left: the simulation might get stuck in an infinite loop. How can we adapt our algorithm? As discussed previously, the simulation goes in an infinite loop if and only if it makes more than $\ell(|Q||\Gamma|)^\ell$ computation steps, where $\ell = |x| + 1$ and $M = (Q, \Gamma, \delta, s)$. Hence, while simulating, we count the number of simulated steps and we reject the input after exceeding the previously mentioned number of steps. We have shown that IN-SPACE ACCEPTANCE is in PSPACE.

Now we prove that the problem IN-SPACE ACCEPTANCE is PSPACE-complete. We reduce TQBF to this problem. To every quantified Boolean formula ψ we associate a machine M_ψ that deletes the input string, writes down ψ, runs the algorithm to check whether ψ is true using a linear amount of space (see Example 8.3 in Sipser's book p332), and accepts if and only if ψ is true. We choose x_ψ to be any string of length $C(1 + |\psi|)$, where C is large enough so that $C(1 + n)$ exceeds the space needed by the algorithm that checks whether ψ is true. The reduction maps ψ to the pair (M_ψ, x_ψ)

We show that this is indeed a reduction. The description of M_ψ is polynomial in the length of ψ. By construction x_ψ has polynomial size. By construction M_ψ uses at most space $|x_\psi|$ and accepts x_ψ if and only if ψ is true.

Exercise 10. We show that MONOTONE-TQBF is in P. To determine whether the formula is true, we consider the formula game from Sipser's book p342. Recall that the $\exists$-player wants to make the formula equal to 1 and the $\forall$-player wants to make the formula 0. An optimal strategy for the $\exists$-player is to always play 1, because the formula is monotone, thus changing a variable from 0 to 1 can never decrease the value of the Boolean function, and this is true regardless of the variables assigned by the $\forall$-player. (This means that the $\exists$-player has an optimal strategy that is *blind*.) For the same reason, the $\forall$-player's optimal strategy is to always play 0. Fixing the values in this way, we can determine the winner by evaluating the Boolean function. Hence, we can determine whether the formula is true.