Exercise V, Computational Complexity 2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. Solve as many problems as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

Space Complexity

1 Basics:

(a) Prove that TQBF $\in$ PSPACE by designing a polynomial-space algorithm for it.
(b) Prove that NL $\subseteq$ P. *(Hint: Construct the whole configuration graph for an* NL *algorithm.)*

Solution:

(a) Let $\phi = Qx_1\,\phi'(x_1)$ be a TQBF instance where ϕ' is a (smaller) TQBF instance parametrized by x_1. Observe that:

$$\exists x_1\,\phi'(x_1) \iff \phi'(0) \vee \phi'(1)$$
$$\forall x_1\,\phi'(x_1) \iff \phi'(0) \wedge \phi'(0)$$

This gives us a recursive scheme to test the truth of ϕ. Furthermore, the space used for computing $\phi'(0)$ can be completely re-used for the computation of $\phi'(1)$. The Turing machine that implements this has space complexity $s : \mathbb{N} \to \mathbb{N}$ described by $s(n) = s(n-1) + O(n)$ and $s(1) \in O(1)$ so that $s(n) \in O(n^2)$ and thus TQBF $\in$ PSPACE

(b) Fix a language $L \in$ NL and let N be a non-deterministic Turing machine computing L in log-space. Observe that the configuration graph $G_{N,x}$ of N on input x with $n = |x|$ has size poly(n) and can be computed in time poly(n). Note that by the correctness of N, $x \in$ L if and only the initial and accepting configurations are connected in $G_{N,x}$. This can be tested in time poly(n) using breadth-first search.

2 Consider the following single-player *pebble game* played on an directed acyclic graph $G = (V, E)$ with a unique *sink vertex* $v^* \in V$ (out-degree 0). There is a set of pebbles that can be placed on the vertices in V. The game proceeds in rounds. At start, the graph is empty of pebbles. In each round we perform either of the following pebbling moves:

(a) We may place a pebble on any vertex whose predecessors are pebbled. In particular, we may always place a pebble on a *source vertex* (in-degree 0).
(b) We may always remove a pebble from any vertex.

The goal is to pebble the sink v^* while minimising the number of pebbles used (maximum number of pebbles in any configuration during the game).

In the Pebble problem, we are given $\langle G, k\rangle$ as input and want to determine if the sink can be pebbled using at most k pebbles. Show that Pebble $\in$ PSPACE. Do you think Pebble $\in$ NP?

Solution:

Consider the non-deterministic Turing machine N that makes non-deterministic pebbling moves according to rules (a) and (b) subject to the constraint that no more than k pebbles can be used. Each branch must remember the location of at most k pebbles, hence the memory is bounded by $s(n) = O(kn)$. (The machine can also maintain an $s(n)$-bit counter so that all branches halt after at most $2^{s(n)}$ steps.) This Turing machine is indeed a decider for PEBBLE. If the graph admits a pebbling strategy using less than k pebbles, then *some* branch will accept. Conversely, no branch can accept a graph that requires more than k pebbles.

This shows that PEBBLE $\in$ NPSPACE and so PEBBLE $\in$ PSPACE by Savitch's theorem. It turns out that PEBBLE is PSPACE-complete (this is not easy to show). PEBBLE $\in$ NP would thus imply PSPACE = NP, an unlikely collapse.

3 Show that the following parsing problems are in L.

(a) The input is a string of parentheses $x \in \{(,)\}^*$. We wish to know whether x is *properly nested*; for instance, `(())()` and `()()()` are allowed but `())` is not.

(b) The input is a string $x \in \{(,),[,]\}^*$. We again wish to know whether x is properly nested; for instance, `([])[]` and `[()[]]` are allowed but `[(])` is not.

(Hint: Show that one only needs to check a simple condition for each substring of x.)

Solution:

(a) Maintain a counter c (initially set to 0) by processing the input from left to right. Whenever an opening parenthesis is seen, increment c by one and decrease c if the parenthesis is closing. A string is well-parenthesized if and only if at all point $c \geq 0$ and $c = 0$ at the end. Since maintaining the counter takes $\log(c)$ space and $c \leq n$, the problem is indeed in L.

(b) The algorithm maintains two counters c and c'. In the first stage, like in part (a), we check, using a single counter c only, that the input string is properly nested when we ignore the *type* (round or square) of parenthesis. In the second stage, we check that the types match: Looping over all $c = 1, \ldots, n$, if the c-th symbol is an opening parenthesis, we find the corresponding closing one (or decide none exists) by scanning trough the string and maintaining a counter c', initialised to 0, similarly to part (a). The update to the counter c' is made irrespective of the type of bracket. The corresponding closing bracket is the first one encountered with $c' = 0$. If the type matches, then we may increment c and move to considering the next symbol.

4 In the context of the previous problem, suppose the TM is only allowed to scan the input x once from left to right—much like a deterministic finite automaton. Show that (a) can still be done with memory $O(\log n)$, but (b) requires memory $\Theta(n)$.

(Hint: What does the TM need to remember when it sweeps past the middle symbol of x?)

Solution:

(a) The above algorithm still works for this restricted type of Turing machine.

(b) Let $L = \{"(","["\}^{n/2}$ and $R = \{")","]"\}^{n/2}$. Observe that each $\ell \in L$ has a unique $r \in R$ such that the string ℓr is well-parenthesized. Suppose now toward contradiction that there

exists some left-to-right Turing machine M solving the problem using memory $s < n/2$ on all instances of size n. Because $|L| \geq 2^s$, there exists two different strings $\ell_1, \ell_2 \in L$ for which M is in the *same* configuration after having read $n/2$ bits. Now, let $r_1 \in R$ be the unique string such that $\ell_1 r_1$ is well-parenthesized. Since M is correct, M accepts $\ell_1 r_1$ but since ℓ_1 and ℓ_2 have the same configuration after reading $n/2$ bits, M must also accept $\ell_2 r_1$: a contradiction.

5 The *alternating path* game is played by two competing players, Alice and Bob, on a directed graph $G = (V, E, v^*)$ where v^* is a distinguished vertex. The players alternate in constructing a directed path $v_0, v_1, \ldots$ starting at $v_0 = v^*$: Alice chooses v_1 as one of the out-neighbours of v_0, then Bob chooses v_2 as one of the out-neighbours of v_1, and so on the players alternate. The first player who cannot choose a previously unvisited vertex loses the game—that is, the player is forced to choose a previously visited vertex, or the path has terminated at a sink.

In the AltPath problem, the input is the graph G and the goal is to decide if Alice has a winning strategy for the alternating path game on G. Show that AltPath is PSPACE-complete by giving a reduction from TQBF.

(Hint: Suppose $\exists x_1 \forall x_2 \cdots Q_n x_n \colon \varphi(x)$ is the TQBF instance where φ is a 3-CNF. Construct G so that it has one vertex for every literal x_i and $\bar{x}_i$, and connect them by directed edges so that Alice (resp. Bob) gets to choose, for each existentially (universally) quantified x_i, whether to visit vertex labelled x_i or $\bar{x}_i$. After a path has determined a truth assignment to x by visiting appropriate literal vertices, check whether it satisfies the CNF formula by including in G one node each clause in φ.)

Solution: We argue that AltPath $\in$ PSPACE by showing that there is a way to solve it recursively and re-use memory. Let $\{v_1, v_2, \ldots, v_q\}$ be the direct neighbors of v^* and G' be a copy of G with vertex v removed. Observe that:

$$(G, v^*) \in \text{AltPath} \iff \forall i \in [q] : (G', v_i) \notin \text{AltPath}$$

This observation can be turned into a space-efficient implementation by re-using memory. The space complexity of this implementation obeys $s(n) = s(n-1) + O(n)$ with $s(1) = O(1)$. Hence, $s(n) \leq O(n^2)$. Let us now show that AltPath is PSPACE-hard by specifying a poly-time reduction from TQBF. Without loss of generality, we may assume that the TQBF instance is of the following form:

$$\exists x_1 \, \forall x_2 \, \exists x_3 \, \ldots \, \forall x_{n-1} \, \exists x_n \, C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

That is, the inner formula is in CNF form (this can be done in poly-time much like in the reduction form SAT to 3-SAT), the quantifiers alternates and finally, the starting and ending quantifiers are existential ones (this can be done by adding at most n dummy variables).

The corresponding AltPath graph consists of two parts. In the first part, Alice (which holds the existential quantifiers) and Bob (which has the universal ones) commit to variable values in turn. Once all the variables have been set, Bob chooses a clause to challenge Alice on. Alice responds by selecting some literal of that clause as a proof that the clause is satisfied. This literal vertex is connected only to its corresponding literal in the first part of the graph so that Bob cannot move if that literal was selected earlier. If the literal was not selected earlier, then its negation was and thus Alice is the blocked one. See figure 1 for an example of the transformation. This graph can be constructed in time polynomial and we now prove that the reduction preserves

solutions.

Suppose first that Alice has a winning strategy in the graph. Then applying this strategy to the existential variables of the TQBF instance is such that no matter how the universal variables are set, all clause are satisfied (recall that Bob is free to challenge any clause). Now, if the TQBF instance is true, this also gives a way for Alice to win in the graph.

Remark: This problem is also called "generalized geography".

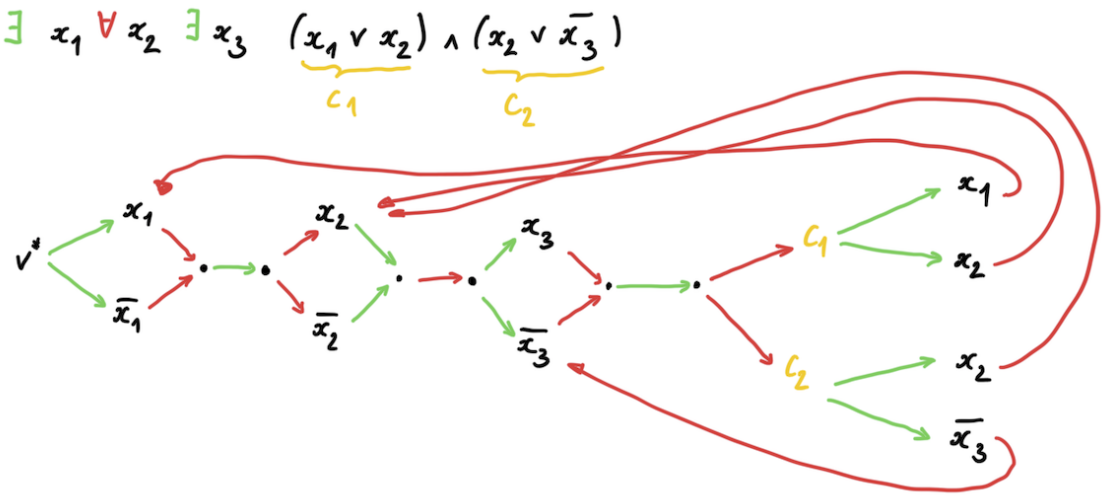

Figure 1. Transforming a TQBF instance into an AltPath one. Green edges are the one that Alice can select and Bob has the red ones