1 A few reminders

Definition (Function Space-constructible). A function $f : \mathbb{N} \to \mathbb{N}$ is said to be space-constructible if $\forall n \in \mathbb{N}$, $f(n) > \log(n)$ and there exists a deterministic Turing machine that computes $f(|x|)$ in space $O(f(|x|))$ given $x$ as input.

We also recall the definition of hardness and completeness w.r.t. $\text{NL}$.

Definition (NL-hardness and completeness). Consider a decision problem $A$.

- It is hard for $\text{NL}$ if for every problem $B$ in $\text{NL}$, there is a log-space reduction from $B$ to $A$.

- It is $\text{NL}$-complete if $A \in \text{NL}$ and $A$ is hard for $\text{NL}$.

Also, recall that the decision problem $\text{PATH}$ is $\text{NL}$-complete where $\text{PATH}:

- Input: a graph $G$ and two vertices $s,t$.

- Output: Yes if and only if there is a path from $s$ to $t$.

Finally, recall that this problem is also in $\text{co-NL}$.

Theorem 1 (Immerman-Szelepcsenyi). $\text{PATH}$ is in $\text{co-NL}$.

2 Exercises

Exercise 1 (Inclusions of complexity classes). Show that for a space-constructible function $f$,

$$\text{NSPACE}(f(n)) \subseteq \text{DTIME}(2^{O(f(n))})$$

Solution 1. Consider a Turing Machine $M$ running in space $f$ (that is space-constructible) with $Q$ as set of states, $k$ tapes and an alphabet $\Gamma$. Consider also an input $x$ such that $n := |x|$.

- A configuration of the Turing machine $M$ on $x$ consists in a state (in $Q$), $k$ sequences of at most $f(n)$ letters in the alphabet $\Gamma$ on the working tapes with the $k$ locations of the head in the working tapes plus the location of the reading head in the (read-only) input tape. Hence, there are at most:

$$|Q| \cdot |\Gamma|^{k \cdot f(n)} \cdot f(n)^k \cdot n$$

different possible configurations. Hence, the size of the configuration graph is thus in $2^{O(f(n))}$.
In order to decide if \( x \) is accepted by \( M \), we just have to use a breadth first search algorithm on the configuration graph to check whether \( C_{\text{initial}} \) and \( C_{\text{accept}} \) are connected. This algorithm is polynomial in the size of the graph, so we finally do have a time complexity in \( 2^{O(f(n))} \).

**Exercise 2** (One-minute-long exercise). At which conditions is a language \( A \subset \{0,1\}^* \) hard for \( \text{NL} \) under polynomial-time reductions?

**Solution 2.** Consider a language \( A \subseteq \{0,1\}^* \). Consider another language \( B \) in \( \text{NL} \) and let us try to reduce an instance \( x_B \) of \( B \) into an instance \( x_A \) of \( A \) via a polynomial time reduction such that \( x_B \in B \iff x_A \in A \). By the previous exercise, we have \( \text{NL} \subseteq \text{P} \), hence, it can be decided in polynomial time if \( x_B \in B \). If so, \( x_A \) needs to be such that \( x_A \notin A \), otherwise we need \( x_A /\notin A \). As positive and negative instances exists in some languages in \( \text{NL} \), it follows that \( A \) needs (and it is sufficient) to be non-trivial, i.e. \( \emptyset \subsetneq A \subseteq \{0,1\}^* \).

**Exercise 3** (A few \( \text{NL} \)-complete problems). Show that the following problems are \( \text{NL} \)-complete.

1. Deciding if a non-deterministic automaton \( \mathcal{A} \) accepts a word \( w \).
2. Deciding if a directed graph is strongly connected.
3. Deciding if a directed graph has a cycle.

**Solution 3.**  

1. The problem is in \( \text{NL} \) as we can simply guess the path in the automaton corresponding to the word and arrive in an accepting state. To be in logspace, we do however guess the whole path at once, but guess transition by transition while keeping a counter to the current state.

   Then, to establish \( \text{NL} \)-hardness, given a graph \( G \) and two vertices \( s \) and \( t \), we label all the edges with the letter \( \epsilon \) in order to create an automaton \( \mathcal{A} \), with initial state \( s \) and final state \( t \). Now, \( (G,s,t) \in \text{PATH} \) is equivalent to having \( \mathcal{A} \) accepts \( \epsilon \). The reduction is straightforwardly logspace.

2. The problem is in \( \text{co–NL} \) (\( = \text{NL} \)), as we can simply guess two nodes and check that they are not connected. We once again reduce \( \text{PATH} \), so we are given \( (G,s,t) \). We construct \( G' \) by copying \( G \), and for every vertex \( v \), we add an edge from \( t \) to \( v \) and from \( v \) to \( s \). This reduction is logspace as we only need one counter for the loop on the vertices. Furthermore, \( G' \) is strongly connected if and only if there is a path from \( s \) to \( t \) in \( G \) (and \( G' \)).

3. The problem is in \( \text{NL} \), given \( G \) we guess an edge \( (x,y) \) of the cycle and run \( \text{PATH} \) on \( (G,y,x) \).

Consider now the \( \text{NL} \)-hardness. Given an instance of \( (G,s,t) \) of \( \text{PATH} \), we may create \( G' \) by first adding an edge between \( t \) and \( s \), creating a cycle inside \( G' \) if \( s \) and \( t \) are connected in \( G \). This is not enough, because \( G \) may have other cycles and the equivalence would not hold. Thus, we must first eliminate all the cycles in \( G \). Let \( m \) be the number of nodes in \( G \). We create \( m \) copies of \( G \), which can be seen as \( m \) levels. For every edge from \( i \) to \( j \) in \( G \), we draw an edge from node \( i \) at each level to node \( j \) at the next level. Additionally, we draw an edge from each node \( i \) at each level to node \( i \) at the next level. We call \( s' \) the \( s \) of the first level, and \( t' \) the \( t \) of the last level. Now, there is a path from \( s \) to \( t \) in \( G \) if and only if there is path from \( s' \) to \( t' \) in \( G' \). Moreover, in \( G' \) path are only "going up" into the levels, so there
cannot be any circle. Thus if we add an edge from \( t' \) to \( s' \), we now have that there is a path from \( s \) to \( t \) in \( G \) if and only if there is a cycle in \( G' \). We can check that this operation can be done in logarithmic space.

**Exercise 4** (Graph representation and why it does not matter). In the definition of \textsc{PATH}, we did not specify which graph representation is used. In fact, this does not matter as the two usual representations (by adjacency list or matrix) are log-reducible from one another.

Let \( \Sigma = \{0, 1, \ldots, \#\} \) with \( \# \) the end-of-word symbol. For a directed graph \( G = (V, E) \) with \( V = [0, n - 1] \) for some \( n \in \mathbb{N} \) and \( E \subseteq V \times V \), we consider the following two representations of \( G \) by a word in \( \Sigma^* \):

- By its adjacency matrix \( m_G \in \Sigma^* \):
  \[
m_G \overset{\text{def}}{=} m_{0,0}m_{0,1} \ldots m_{0,n-1} \bullet \ldots \bullet m_{n-1,0} \ldots m_{n-1,n-1} \#
\]
  where for all \( 0 \leq i, j < n \), \( m_{i,j} \) is 1 if \( (i, j) \in E \), 0 otherwise.

- By its adjacency list \( l_G \in \Sigma^* \):
  \[
l_G \overset{\text{def}}{=} k_0^0 / \ldots / k_{m_1} \bullet \ldots \bullet k_{n-1}^0 / \ldots / k_{m_n-1}^0 \#
\]
  where for all \( 0 \leq i < n \), \( k_i^0, \ldots, k_i^{m_i} \) are binary words listing the (codes of) right neighbors of vertex \( i \).

1. Show that it is possible to check in logarithmic space that a word \( w \in \Sigma^* \) is a well-formed description of a graph (for any of the two representations).

2. Describe a logarithmic space bounded deterministic Turing machine taking as input a graph \( G \), represented by its adjacency matrix, and computing the adjacency list representation of \( G \).

**Solution 4.** We make two preliminary observations.

- Firstly, it is easy to implement a binary counter on an empty worktape \( T_B \). E.g., for little-endian representation we increment by reading \( T_B \) left to right, replacing 1s by 0s, stopping at the first 0 and replacing it with a 1. If we reach the end of \( T_B \) without seeing a 0, we add a 1 to the right. Then, we reset the TM head to the start of the tape, and the incrementation is over.

- Secondly, comparing two numbers written in binary (with no extra 0s on the right) just needs logspace. If the counters are on different worktapes one needs no extra space and only linear time: one checks that the counters have different sizes and deduces which is larger, and if they have same size one checks if they coincide or reading from the right, finds the first bit where they differ and deduces which counter is the largest. If the counters are on the same worktape (usually the input tape) one just needs logarithmic space, e.g., to store pointers to the extremities of the counters before performing the comparison, or to copy them on different auxiliary worktapes and reuse the previous method.

In the following, \( N \) will denote the size of the input.

1. First, consider the adjacency matrix case. Checking that \( w \in \Sigma^* \) consists in sequences of 0s and 1s separated by the symbol \( \bullet \) and ending with the symbol \( \# \) is straightforward\(^1\). But we also have to check that between any two consecutive \( \bullet \)s

---

\(^1\)This is a “regular” constraint and any regular language is in \( \text{TIME}(n) \cap \text{SPACE}(0) \).
(and also before the first ⋅) there are exactly \( m + 1 \) symbols, where \( m \) is the number of ⋅s. To do so, we may use an additional tape \( T_B \) as a counter, incrementing it each time the symbol ⋅ or # is seen, so that in the end \( T_B \) stores \( m + 1 \). Note that \( m + 1 \) written in binary takes \( O(\log N) \) bits. Then, we consider another working tape \( T_W \) where, for each sequence in \( \{0,1\}^* \) we increment a counter until we reach the next ⋅ symbol. Then, we can check that it is equal to the number \( m + 1 \) on \( T_B \).

The case of adjacency list is analogous, but instead of checking the number of symbols between two ⋅s, we have to check that every number written (between /) is smaller than the value of the counter written on the tape \( T_B \).

2. We first use the Turing Machine we described in the previous question to check that the word considered is well-formed. Then, we consider a Turing Machine \( M \) with a tape \( T_i \) where a counter will loop between 0 and \( k - 1 \) (where \( k \) is the number of ⋅ symbol plus 1) assuming the word we consider is well formed. The counter \( i \) is incremented whenever a new symbol 0 or 1 is seen and reinitialized at each ⋅ symbol. Furthermore, whenever ⋅ is seen, ⋅ is written on the output tape and whenever a 1 is seen, the counter \( i \) is copied on the output tape, preceded by symbol / if it is not the first time the counter \( i \) is copied since the last time a ⋅ symbol was written. The space taken by this TM is in \( O(\log N) \) since we only use a counter whose value, written in binary, is at most \( k \) and the logarithmic space taken by the TM described in the first question. Then, we conclude with the speedup theorem.

Exercise 5 (Restrictions in the definition of \( \text{SPACE}(f(n)) \)). In the course, we restricted our attention to Turing machines that always halt, and whose computations are space-bounded on every input. In particular, remember that \( \text{SPACE}(f(n)) \) is defined as the class of languages \( L \) for which there exists some deterministic Turing machine \( M \) that always halts (i.e. on every input), whose computations are \( f(n) \) space-bounded (on every input), such that \( M \) decides \( L \).

Now, consider the following two classes of languages:

- \( \text{SPACE}'(f(n)) \) is the class of languages \( L \) such that there exists a deterministic Turing machine \( M \), running in space bounded by \( f(n) \), such that \( M \) accepts \( x \) iff \( x \in L \). Note that if \( x \notin L \), \( M \) may not terminate.

- \( \text{SPACE}''(f(n)) \) is the class of languages \( L \) such that there exists a deterministic Turing machine \( M \) such that \( M \) accepts \( x \) using space bounded by \( f(n) \) iff \( x \in L \) (\( M \) may use more space and not even halt when \( x \notin L \)).

1. Show that for a space-constructible function \( f \), \( \text{SPACE}'(f) = \text{SPACE}(f) \)

2. Show that for a space-constructible function \( f \), \( \text{SPACE}''(f) = \text{SPACE}(f) \)

Solution 5. 1. First, \( \text{SPACE}(f(n)) \subseteq \text{SPACE}'(f(n)) \). Conversely, let there be \( M \) as specified, with \( a = |Γ| \), \( M \) terminates in times \( O(a^{f(n)}) \) or does not terminate (she cannot run more than the number of possible configurations). We construct \( M_0 \) from \( M \) by adding a counter \( B \) (basically, a timeout) incremented by 1 at every steps of \( M \). \( B \) is written in base \( a \), and as soon as \( B \) takes more bits than necessary to write the number of possible configurations, we reject (the maximum size of \( B \) can be set in space \( f(n) \) because \( f \) is space constructible). This uses \( O(f(n)) \) space, we conclude with the speed-up theorem.
2. First, $\text{SPACE}(f(n)) \subseteq \text{SPACE}''(f(n))$. Conversely, we want to force $M$ to only use space $f(n)$ on every input. We construct $M_0$, which starts by writing $f(n)$ 0 on a tape $B$ (f space constructible). $M_0$ also contains the tapes of $M$, and copy $B$ on those tapes, adding a special symbol # at the end of each of them. $M_0$ then simulates $M$, but rejecting if it goes over #. If $x$ is in $L$, $M$ runs in space $f(n)$ and $M_0$ accepts. If $x \notin L$, $M_0$ cannot accept $x$. So $M_0$ accepts $x$ iff $x \in L$, and $M_0$ runs in $O(f(n))$ space, so $M_0 \in \text{SPACE}'(O(f(n))) \subseteq \text{SPACE}(f(n))$ by 1.