Complexité avancée - TD 2

Benjamin Bordais

September 30, 2020

Exercise 1: Dyck's language

- Let A be the language of balanced parentheses – that is the language generated by the grammar $S \to (S)|SS|\epsilon$. Show that $A \in \mathsf{L}$.
- What about the language B of balanced parentheses of two types? that is the language generated by the grammar $S \to (S)|[S]|SS|\epsilon$

Solution:

- We describe an algorithm running in logarithmic space. We read the input from left to right while maintaining a "balancing" counter on a working tape with initial value zero, and increment (resp. decrement) it when reading '(' (resp. ')'). We reject if the counter ever becomes negative, and accept if the counter is zero at the end of the input. Since the counter can never exceed the input length n, it is a $log_2(n)$-bit number.
- Let us show that $B \in \mathsf{L}$. We say that each symbol has a type, either round or square, and that each symbol is a left or right bracket regardless of the type. Each left bracket has a right bracket which is its partner, and our goal is to check that every left bracket's partner is of the same type. To find its partner we use a counter as in question 1 above. First, we check that the word is in the bracket language of question 1 if we ignore round vs. square, so that every left bracket has a right partner.

  There now remains to check that every opening bracket is partnered with a closing bracket of the same type (round or square). For this we observe that if $x[i]$ is an opening bracket matched at position j by a closing bracket, the factor $x[i+1{:}j-1]$ is a well-balanced word. So it is enough to loop over all $i = 1, \ldots, n$ such that $x[i]$ is an opening bracket and run the algorithm for well-balanced words starting from position $i+1$. When the "balancing" counter first becomes negative, this indicates that we have arrived at the position j matching i. We now check that the closing parenthesis at j has the same type as its partner at i. This uses two counters: one for looking at all positions i and the "balancing" counter used by the subprogram on the $x[i+1{:}j]$ factor.

  Then, to check that partners match, we use the following pseudocode:

```
i = 1
do until i exceeds the length of the input {
```

```
move i-1 steps from the left end of the input
read the input symbol a    // a = w(i)
if a is a left bracket {
c = 1
do until c = 0 {    // find w(i) partner
move right and read the next input symbol b
if b is a left bracket, increment c
if b is a right bracket, decrement c
}
if a and b are of different types, reject
}
}
accept
```

Exercise 2: Restrictions of the SAT problem

1. Let 3-SAT be the restriction of SAT to clauses consisting of at most three literals (called 3-clauses). In other words, the input is a finite set S of 3-clauses, and the question is whether S is satisfiable. Show that 3-SAT is NP-complete for logspace reductions (assuming SAT is).
2. Let 2-SAT be the restriction of SAT to clauses consisting of at most two literals (called 2-clauses). Show that 2-SAT is in P, using proofs by resolution.
3. Show that 2-UNSAT (i.e, the unsatisfiability of a set of 2-clauses) is NL-complete.
4. Conclude that 2-SAT is NL-complete. You may use that coNL = NL.

Solution:

1. First, the problem is in NP as a sub case of SAT. We now must be able to transform any instance of SAT into an instance of 3-SAT by using only logarithmic space. The idea is the following: consider a clause $C = l_1 \vee l_2 \vee L$ where L is a non-empty subclause. Then, C and $C_x \wedge C_{\neg x}$ are equisatisfiable where $C_x = l_1 \vee l_2 \vee x$ and $C_{\neg x} = \neg x \vee L$ for some fresh variable x. Indeed, consider a valuation (that does not contain x) satisfying C. Then if $l_1 \vee l_2$ is satisfied, C_x also is and we set x to false to satisfy $C_{\neg x}$. If L is satisfied, we set x to true so that C_x also is. Reciprocally, if $C_x \wedge C_{\neg x}$ is satisfied for some valuation (containing x), by a case disjonction on the truth value of x, we can With this transformation, we have a new 3-clause C_x, and a new clause $C_{\neg x}$ that is not necessarily a 3-clause but it is a clause with one less literal than C. Hence, from a SAT formula $\varphi = \wedge_{i=1}^{n} C_i$, if n_i refers to the number of literals in clause C_i, we have to do $k = \sum_{i=1}^{n} l_i - 3$ such transformations to obtain an equisatisfiable formula with only 3-clauses.

   Let us show that this transformation can be done in logarithmic space. We denote by N the size of the input. We first read the input to obtain the number n of variables, and write $n+1$ (in binary) on a work tape B. Then, we treat each clauses one after the other. For a clause C_i, we keep a pointer on the subclause L_i, we copy the first two literals, we add the fresh variable x (by copying the content of the tape B) and we write $\neg x$, then we increment the counter on the tape B. While the

pointer we keep on L is followed by at least three literals, we copy the first literal and copy a new fresh variable.

The counter written in binary in the tape B equals at most $n + k = O(\log N)$. It can then be represented in logspace, as can the pointer on the remaining subclause. Overall, this can be done in logarithmic space.

2. From the formula φ, we construct a graph G_φ where the nodes are all the variables in φ and their negation. For every clause $C = l_1 \vee l_2$ we create an edge from $\neg l_1$ to l_2 and one from $\neg l_2$ to l_1. We denote the fact that there is a directed path from t to s in G_φ by $t \to^* s$. Then, we claim that φ is satisfiable if and only if there is no variable x such that $x \to^* \neg x \to^* x$. First, it is straightforward to show that if we have path $l \to^* l'$ in G_φ for two literals l and l', then we have $\varphi \Rightarrow (l \Rightarrow l')$ [Proof idea: by induction on the length of the path, the base case uses the definition of edges from φ]. Finally, if G_φ has a cycle $x \to^* \neg x \to^* x$ for some variable x then φ entails $(x \Rightarrow \neg x) \wedge (\neg x \Rightarrow x)$, i.e., φ entails a contradiction and must be uniformly false, i.e. not satisfiable. That proves direction " $\Longrightarrow$ " of the above claim.

   Let us now prove the reverse direction. We assume that there is no such literal and must show that φ is satisfiable. We define by induction on $1 \leq i \leq n$ the valuation of x_i and a graph G_i as follow. We set $G_0 = G_\varphi$, and for $1 \leq i \leq n$, if we have $\neg x_i \to^* x_i$ in G_{i-1} then we set x_i to true and $G_i = G_{i-1}$. Otherwise, x_i is set to false and G_i is obtained from G_{i-1} by adding an edge between x_i and $\neg x_i$. By induction, we show that for all $0 \leq i \leq n$, the graph G_i does not contain a cycle going through a literal and its negation. It is true for G_0 by hypothesis. Now, consider $i \geq 1$ assume this holds for G_{i-1}. The case $G_i = G_{i-1}$ is obvious. Assume that G_i is obtained from G_{i-1} by adding an edge between x_i to $\neg x_i$. Assume that there is a cycle going through a literal and its negation in G_i. Since there is none in G_{i-1}, it must use the arc $x_i \to \neg x_i$. Therefore, there is a path from $\neg x_i$ to x_i in G_i, which also exists in G_{i-1}. Hence the contradiction since we consider the case where $\neg x_i \to^* x_i$ does not hold. By construction we also have that, for all variable x, in G_n, either $x \to^* \neg x$ or $\neg x \to^* x$.

   Now, consider the valuation $v : \{x_i \mid 1 \leq i \leq n\} \to \{T, F\}$ that we have constructed. We have that, for all literal l, $v(l) = T \Leftrightarrow (\neg l \to^* l)$ and $v(l) = F \Leftrightarrow (l \to^* \neg l)$. Let us show that the valuation v satisfies φ. Consider a clause $C = (l_1 \vee l_2)$. If $v(l_1) = T$, C is satisfied. Now, assume that $v(l_1) = F$. Then, we have $l_1 \to^* \neg l_1$. Furthermore, by construction of $G_\varphi = G_0$, we have $\neg l_2 \to l_1$ and $\neg l_1 \to l_2$. Hence, we have the path $\neg l_2 \to^* l_2$. That is, $v(l_2) = T$ and the clause is satisfied. This holds for all clauses of φ.

   It follows that $2 - \mathsf{SAT}$ is in P since the construction of the graph can be done in polynomial time and and so can checking the existence of a cycle going through a literal and its negation.

3. In the previous question, we have established that a 2-SAT formula φ is satisfiable if and only if, in the graph G_φ, there exists a variable x so that $x \to^* \neg x \to^* x$. Hence, the unsatisfability of a 2-SAT formula φ can be decided in logarithmic space. Note that the graph G_φ is never constructed in its entirety. We only keep a pointer on the current vertex and we check the formula φ to determine which are the possible successors. By doing, we can guess a variable x and call $REACH(G_\varphi, x, \neg x)$ and $REACH(G_\varphi, \neg x, x)$.

Let us prove that the UNSAT is NL-hard. Consider an instance (G, s, t) of REACH. We construct a 2-SAT formula φ from G by adding a 2-clause $\neg u \vee v$, i.e. "$u \Rightarrow v$", for every edge (u, v) in G. We also add two unit clauses: s and $\neg t$. Then, if $s \to^* t$ in G, φ entails the implication $s \Rightarrow t$ (or $\neg s \vee t$), hence φ is false since it also contains s and $\neg t$ as clauses. On the other hand, assuming that t is not reachable from s, we can construct a valuation satisfying φ: we assign true to every variable reachable from s in G, and false to the unreachable ones (in particular, t is set to false). Then, the clauses s and $\neg t$ are satisfied. Furthermore, consider a clause $(\neg u \vee v)$ of φ. If u is set to false, the clause is satisfied, otherwise u is set to true, which means that it is reachable from s, and so is v. Hence v is set to true and the clause is satisfied. Finally we have shown that φ is not satisfiable if and only if (G, s, t) is a positive instance of REACH, i.e., our reduction is correct. Since this transformation can be done in logarithmic space, we can conclude that UNSAT is NL-complete.

4. $\overline{\text{2-SAT}} = \text{2-UNSAT}$ is in $\text{NL} = \text{coNL}$. Hence, $\overline{\text{2-SAT}}$ is in NL. Furthermore, for any language $A \in \text{NL}$, $\overline{A} \in \text{coNL} = \text{NL}$ can be reduced in logspace to 2-UNSAT that is NL-complete. The same reduction can be used to reduce $A = \overline{\overline{A}}$ to $\overline{\text{2-UNSAT}} = \text{2-SAT}$.