Review: PSpace-complete problems

We have encountered some PSpace-complete problems so far:

- The word problem for polynomially space bounded (N)TMs
- TRUE QBF
- FOL MODEL CHECKING (and SQL query answering)

Several typical PSpace problems are related to the existence of winning strategies in 2-player games:

- FORMULA GAME
- GEOGRAPHY

Review: GEOGRAPHY is PSpace-hard

We consider the formula $\exists X. \forall Y. \exists Z. (X \lor Z \lor Y) \land (\neg Y \lor Z) \land (\neg Z \lor Y)$
More Games

The characteristic of PSpace is quantifier alternation. This is closely related to taking turns in 2-player games.

Are many games PSpace-complete?

• **Issue 1:** many games are finite — that is: computationally trivial
  ~ generalise games to arbitrarily large boards
    – generalised Tic-Tac-Toe is PSpace-complete
    – generalised Reversi (Othello) is PSpace-complete
    – it is not always clear how to generalise a game (Generalised Backgammon?)

• **Issue 2:** (generalised) games where moves can be reversed may require very long matches
  ~ such games often are even harder
    – generalised Go with Japanese ko rule is ExpTime-complete
    – generalised Draughts (Checkers) is ExpTime-complete
    – generalised Chess (without 50-move no-capture draw rule) is ExpTime-complete

Surprisingly, some of these games, e.g. Chess, are known to become even harder — namely ExpSpace-complete — if the exact same board position is not allowed to re-occur in a match. For Go, this case is open.

Logarithmic Space

**Polynomial space**

As we have seen, polynomial space is already quite powerful.

We therefore consider more restricted space complexity classes.

**Linear space**

Even **linear** space is enough to solve $\text{Sat}$.

**Sub-linear space**

To get sub-linear space complexity, we consider Turing-machines with separate input tape and only count working space.

Recall:

$$L = \text{LogSpace} = \text{DSpace}(\log n)$$

$$NL = \text{NLogSpace} = \text{NSpace}(\log n)$$

Problems in L and NL

What sort of problems are in L and NL?

In logarithmic space we can store

• a fixed number of counters (up to length of input)
• a fixed number of pointers to positions in the input string

Hence,

• L contains all problems requiring only a constant number of counters/pointers for solving.
• NL contains all problems requiring only a constant number of counters/pointers for verifying solutions.
Example 11.1: The language \( \{0^n1^n \mid n \geq 0\} \) is in \( L \).

Algorithm:
- Check that no 1 is ever followed by a 0
  Requires no working space (only movements of the read head)
- Count the number of 0's and 1's
- Compare the two counters

Example 11.2: Palindromes \( \in L \).

Algorithm:
- Use two pointers, one to the beginning and one to the end of the input.
- At each step, compare the two symbols pointed to.
- Move the pointers one step inwards.

Example 11.3: Reachability \( \in \text{NL} \).

Algorithm:
- Use a pointer to the current vertex, starting in \( s \)
- Iteratively move pointer from current vertex to some neighbour vertex nondeterministically
- Accept when finding \( t \); reject when searching for too long

An Algorithm for \textbf{Reachability}

More formally:

\begin{verbatim}
01 CanReach(G, s, t) :=
02 c := |V(G)| // counter
03 p := s // pointer
04 while c > 0 :
05    if p = t :
06        return TRUE
07    else :
08        nondeterministically select G-successor \( p' \) of \( p \)
09        p := p'
10        c := c − 1
11    // eventually, if no success:
12    return FALSE
\end{verbatim}
Defining Reductions in Logarithmic Space

To compare the difficulty of problems in P or NL, polynomial-time reductions are useless. Recall the respective result from Lecture 5:

Theorem 5.22: If \( B \) is any language in P, \( B \neq \emptyset \), and \( B \neq \Sigma^* \), then \( A \leq_P B \) for any \( A \in P \).

This also applies to languages in NL (\( \subseteq P \)).

Definition 11.4: A log-space transducer \( M \) is a logarithmic space bounded Turing machine with a read-only input tape and a write-only, write-once output tape, and that halts on all inputs.

A log-space transducer \( M \) computes a function \( f : \Sigma^* \rightarrow \Sigma^* \), where \( f(w) \) is the content of the output tape of \( M \) running on input \( w \) when \( M \) halts.

In this case, \( f \) is called a log-space computable function.

Log-Space Reductions and NL-Completeness

Definition 11.5: A log-space reduction from \( L \subseteq \Sigma^* \) to \( L' \subseteq \Sigma^* \) is a log-space computable function \( f : \Sigma^* \rightarrow \Sigma^* \) such that for all \( w \in \Sigma^* \):

\[ w \in L \iff f(w) \in L' \]

We write \( L \leq_L L' \) in this case.

Definition 11.6: A problem \( L \in NL \) is complete for NL if every other language in NL is log-space reducible to \( L \).

Detour: P-completeness

Log-space reductions are also used to define P-complete problems:

Definition 11.7: A problem \( L \in P \) is complete for P if every other language in P is log-space reducible to \( L \).

We will see some examples in later lectures . . .

Remark: Log-space Reductions for Larger Classes?

Could we use log-space reductions instead of polynomial reductions for defining hardness for other classes, e.g., for NP?

• Some authors do this (prominently Papadimitriou)
• All concrete polynomial reductions we have seen can be computed in logarithmic space

Obvious question: Are the classes “NP-complete problems under polynomial time reductions” and “NP-complete problems under log-space reductions” different?

Today’s answer: Nobody knows (YCTBF)

(at least we have not seen any example of such differences, so it might not matter much in practice)
An NL-Complete Problem

Theorem 11.8: Reachability is NL-complete.

Proof idea: We already showed membership. What remains is hardness. Let $M$ be a non-deterministic log-space TM deciding $L$.

On input $w$:
1. modify Turing machine to have a unique accepting configuration (easy)
2. construct the configuration graph (graph whose nodes are configurations of $M$ and edges represent possible computational steps of $M$ on $w$)
3. find a path from the start configuration to the accepting configuration

NL-Completeness

Proof sketch: We construct $\langle G, s, t \rangle$ from $M$ and $w$ using a log-space transducer:

1. A configuration $(q, w_2, (p_1, p_2))$ of $M$ can be described in $c \log n$ space for some constant $c$ and $n = |w|$.
2. List the nodes of $G$ by going through all strings of length $c \log n$ and outputting those that correspond to legal configurations.
3. List the edges of $G$ by going through all pairs of strings $(C_1, C_2)$ of length $c \log n$ and outputting those pairs where $C_1 \vdash_M C_2$.
4. $s$ is the starting configuration of $G$.
5. Assume w.l.o.g. that $M$ has a single accepting configuration $t$.

$w \in L$ iff $\langle G, s, t \rangle \in$ Reachability

(co also Sipser, Theorem 8.25)

coNL

As for time, we consider complements classes for space.

Recall Definition 9.6:
For a complexity class $C$, we define $\text{co}C := \{L | \overline{L} \in C\}$.

Complement classes for space:
- $\text{coNL} := \{L | \overline{L} \in \text{NL}\}$
- $\text{coNSpace} := \{L | \overline{L} \in \text{NSpace}\}$

From Savitch’s theorem:
$\text{PSpace} = \text{NSpace}$ and hence $\text{coNSpace} = \text{PSpace}$, but merely $\text{NL} \subseteq \text{DSpace} (\log^2 n)$ and hence $\text{coNL} \subseteq \text{DSpace} (\log^2 n)$

The NL vs. coNL Problem

Another famous problem in complexity theory: is NL = coNL?

- First stated in 1964 [Kuroda]
- Related question: are complements of context-sensitive languages also context-sensitive? (such languages are recognized by linear-space bounded TMs)
- Open for decades, although most experts believe NL $\neq$ coNL
The Immerman-Szelepcsényi Theorem

Surprisingly, two independent people resolve the NL vs. coNL problem simultaneously in 1987.

More surprisingly, they show the opposite of what everyone expected:

**Theorem 11.9 (Immerman 1987/Szelepcsényi 1987):** NL = coNL.

**Proof:** Show that *reachability* is in NL. (Why does this suffice?)

Remark: alternative explanations provided by
- Sipser (Theorem 8.27)
- Dick Lipton’s blog entry *We All Guessed Wrong* (link)
- Wikipedia Immerman–Szelepcsényi theorem

Towards Nondeterministic Nonreachability

How could we check in logarithmic space that \( t \) is not reachable from \( s \)?

**Initial idea:** iterate through all reachable nodes looking for \( t \)

**NaiveNonReach** \((G, s, t)\):

\[
\begin{align*}
\text{for each vertex } v \text{ of } G : \\
\text{ if CanReach}(G, s, v) \text{ and } v = t : \\
\text{ return FALSE} \\
\text{// eventually, if FALSE was not returned above:} \\
\text{return TRUE}
\end{align*}
\]

Does this work?

**No:** the check CanReach \((G, s, v)\) may fail even if \( v \) is reachable from \( s \)
Hence there are many (nondeterministic) runs where the algorithm accepts, although \( t \) is reachable from \( s \).

Counting Reachable Vertices – Intuition

**Idea:**
- Count number of vertices reachable in at most \( \text{length} \) steps
  - we call this number \( \text{count}_{\text{length}} \)
  - then the number we are looking for is \( \text{count} = \text{count}_{\text{length}} - 1 \)
- Use a limited-length reachability test: CanReach \((G, s, v, \text{length})\): "\( t \) reachable from \( s \) in \( G \) in \( \leq \text{length} \) steps"
  (we actually implemented CanReach \((G, s, v, \text{length})\) as CanReach \((G, s, v, |V(G)| - 1)\))
- Compute the count iteratively, starting with \( \text{length} = 0 \) steps:
  - for \( \text{length} > 0 \), go through all vertices \( u \) of \( G \) and check if they are reachable
  - to do this, for each such \( u \), go through all \( v \) reachable by a shorter path, and check if you can directly reach \( u \) from them
  - use the counting trick to make sure you don’t miss any \( v \)
  (the required number \( \text{count}_{\text{length}} \) was computed before)
Counting Reachable Vertices – Algorithm

The count for \( \text{length} = 0 \) is 1. For \( \text{length} > 0 \), we compute as follows:

1. \( \text{CountReachable}(G, s, \text{length}, \text{count}_{\text{length}-1}) : \)
   2. \( \text{count} := 1 \) // we always count \( s \)
   3. for each vertex \( u \) of \( G \) such that \( u \neq s \):
      4. \( \text{reached} := 0 \)
      5. for each vertex \( v \) of \( G \):
         6. if \( \text{CanReach}(G, s, v, \text{length} - 1) : \)
            7. \( \text{reached} := \text{reached} + 1 \)
            8. if \( G \) has an edge \( v \rightarrow u \) :
               9. \( \text{count} := \text{count} + 1 \)
      10. \( \text{GOTO 03} // \text{continue with next } u \)
   11. if \( \text{reached} < \text{count}_{\text{length}-1} : \)
      12. \( \text{REJECT} // \text{whole algorithm fails} \)
      13. \( \text{return count} \)

Completing the Proof of NL = coNL

Putting the ingredients together:

1. \( \text{NonReachable}(G, s, t) : \)
   2. \( \text{count} := 1 \) // number of nodes reachable in 0 steps
   3. for \( \ell := 1 \) to \( |V(G)| - 1 : \)
      4. \( \text{count}_{\text{prev}} := \text{count} \)
      5. \( \text{count} := \text{CountReachable}(G, s, \ell, \text{count}_{\text{prev}}) \)
   6. \( \text{return CountingNonReach}(G, s, t, \text{count}) \)

It is not hard to see that this procedure runs in logarithmic space, since we use a fixed number of counters and pointers. \( \square \)

Summary and Outlook

Winning board games that don’t allow moves to be undone is often PSpace-complete

L is the class of problems solvable using only a fixed number of linearly bound counters and pointers to the input

NL is the corresponding non-deterministic class, but we do not know if L = NL

Summary:

\[
\begin{align*}
    \text{L} & \subseteq \text{NL} \subseteq \text{PTime} \subseteq \text{NP} \subseteq \text{PSpace} = \text{NPSpace} \\
    \text{coL} & \subseteq \text{coNL} \subseteq \text{coP} \subseteq \text{coNP} \subseteq \text{coPSpace} = \text{coNPSpace}
\end{align*}
\]

What’s next?

- So many \( \subseteq \)! Will we ever get a strict \( \subset \)?
- More generally: can more resources solve more problems?