12. PSPACE

Goal: Understand the computational problems that can be solved using polynomial space.

Roughly:
- PSPACE is about alternation
  - ↳ Finding winning strategies against an opponent in turn-based board games
  - ↳ Handling quantifiers in propositional logic.
- PSPACE is also about storage:
  - ↳ Boolean variables in programs.
  - ↳ Exponential counting (less common).
- PSPACE is also about problems on languages.

12.1 QBF is PSPACE-Complete

Definition:

The quantified Boolean formula problem (QBF) is defined as follows:

Given: A quantified Boolean expression

$$F = Q_1 x_1 \dots Q_n x_n . G(x_1, \dots, x_n)$$

where $Q_i \in \{\forall, \exists\}$ and G is a Boolean formula with free variables $x_1, \dots, x_n$.

Question: Is F true?

Note:
- SAT (NP-complete) is a special case of QBF where all quantifiers are $\exists$.
- VALID (also called TAUTOLOGY) (co-NP-complete) is the case where all quantifiers are $\forall$.

Goal: Show that QBF is PSPACE-complete.

Problem: This is our first PSPACE-hard problem.
Have to give a reduction from every other problem in PSPACE.

Idea: Adapt the Cook-Levin-Ladner construction.

Problem: Will be too large.

Solution: Add Savitch.

Theorem (Stockmeyer & Meyer '73):
QBF is PSPACE-complete under logspace reductions.

Proof:

QBF ∈ PSPACE (membership):

- Assign values to the variables and
recursively evaluate the truth of the formula
for those values
(similar to *split* in Davis-Putnam).
- The depth of the recursion is given by the number of variables.
For each variable we store a Boolean value.

⇒ The algorithm runs in space
linear in the number of variables.
Again, this requires a sketch of the procedure
and a careful estimation.

QBF is PSPACE-hard:

- Let $A \in$ PSPACE be decided by a TM M in space n^k.
We give a logspace reduction to QBF.

The reduction maps an input string w to a formula F so that

F is true iff M accepts w.

- To construct F, we solve a more general problem:
  - Given collections of variables denoted c_1 and c_2 (that are supposed to represent two configurations)
  - and given $t > 0$,
  - we construct a formula $F_{c_1, c_2, t}$.

If we assign c_1 and c_2 actual configurations, we have

$F_{c_1, c_2, t}$ is true iff M can go from c_1 to c_2 in at most t steps.

Hence,

$F_{\text{start}, \text{accept}, 2^{d \cdot n^k}}$ is the formula we are looking for.

- Technically, the formula encodes the contents of cells in a configuration as in the proof of Ladner's theorem:
  - Every cell j has several variables:
    P_j^a for every symbol a.
    Q_j^p for every state p.
  - Since each configuration has n^k cells, we have $O(n^k)$ variables per configuration.

- If $t = 1$, we can construct $F_{c_1, c_2, 1}$.
The formula says that is either $c_1 = c_2$ or $c_1 \to c_2$.

For the first case, say that each of the variables representing c_1 contains the same Boolean value as the corresponding variable for c_2.
For the second case, use the technique from Ladner's proof.

- If $t > 1$, we construct $F_{c_1, c_2, t}$ recursively.

↳ First, an idea that doesn't quite work:

$$F_{c_1, c_2, t} := \exists \underbrace{m_1}_{\text{Sequence of } O(n^k) \text{ variables.}} \left(F_{c_1, m, t/2} \wedge F_{m, c_2, t/2} \right)$$

Pro: The formula has the correct value:

$F_{c_1, c_2, t}$ is true iff M can go from c_1 to c_2 in $\leq t$ steps.

Con: The formula is too big:

- Every recursive step cuts t in half
- but doubles the size of the formula.
- Hence, we end up with a formula of size $t = 2^{d \cdot n^k}$.

↳ To reduce the size of the formula, use the $\forall$-quantifier:

$$F_{c_1, c_2, t} := \exists m : \forall (c_3, c_4) \in \{(c_1, m), (m, c_2)\} : F_{c_3, c_4, t/2}.$$

Essentially, we fold the two recursive subformulas into a single subformula.
We can easily replace the construct

$\forall x \in \{y, z\} [\dots]$ by $\forall x : (x \leftrightarrow y \vee x \leftrightarrow z) \rightarrow \dots$

↳ To calculate the size of $F_{start, accept, 2^{d \cdot n^k}}$
note that each level of the recursion
adds a portion to the formula that is $O(n^k)$.
The number of levels is

$$\log 2^{d \cdot n^k} = O(n^k).$$

Hence, the size of the formula is $O((n^k)^2)$.

↳ It remains to argue that the formula is logspace computable
(clearly, it is polynomial-time computable).
Essentially, all we need to do is to count to n^k,
which can be done in $\log n^k = k \cdot \log n$.

□

12.2 Winning Strategies for Games

Definition:

- A <u>two-person perfect information game</u>
is a triple $G = (Boards, Move, s)$,
where
  - $(Boards, Move)$ is a directed graph and
  - $s \in Boards$ is a <u>start node</u>.

- The game starts in $s_0 = s$.
The players alternately take turns, with Player I moving first.
Player I chooses $s_1 \in Boards$ so that $(s_0, s_1) \in Move$.
Then Player II chooses $s_2 \in Boards$ with $(s_1, s_2) \in Move$.
Then Player I ...

- A player wins by forcing the opponent into a position from which no move is possible, typically called a deadlock, in the case of games a checkmate.

Examples:

- Most games are of this form: Chess, Go, ...
- For Chess:

  Boards := Legal Chessboards × {white, black}
  (whose turn.)

  Move := Legal Chess moves.

- For Geography (next country starts with last letter of previous country: Norway, Yemen, Nicaragua, ...)
  (do not name a country twice)

  Boards := $\mathcal{P}$(Countries) × Letter

  Move := $(M, x) \to (M \setminus \{c\}, y)$,

  where c is a country starting with x and ending in y.

Definition (Winning positions and strategies):

- Define

  Checkmate := $\{ y \in Boards \mid \forall z \in Boards : \neg(y, z) \in Move \}$.

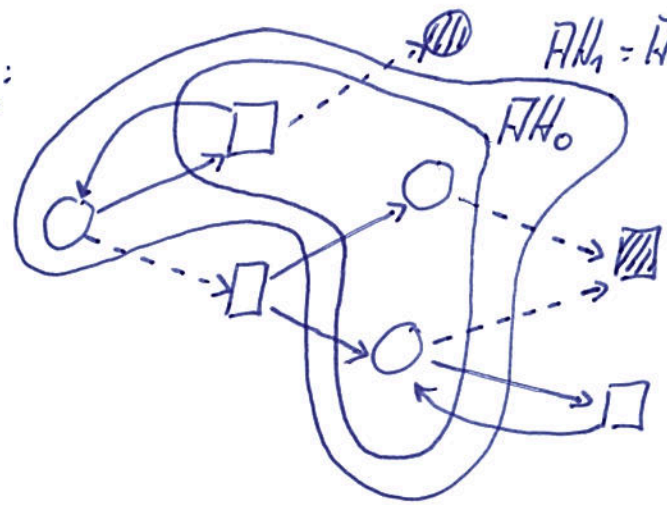
- The set of winning positions (for the player who has the turn) is the least solution to the recursion:

$$X = \{ x \in Boards \mid \exists y : (x,y) \in Move \wedge (y \in Checkmate \vee \forall z : (y,z) \in Move \rightarrow z \in X \}.$$

The solution exists by Knaster & Tarski's theorem.
It is computed using a Kleene iteration,
in this setting also called <u>attractor construction</u>.

- Intuitively, a position is <u>winning</u>
  if the player can win the game
  no matter what moves the opponent does.
- If a position is winning, we also say
  the player has a <u>winning strategy</u> from that position.
  A winning strategy is a strategy of choosing the moves
  in order to win against every possible play of the opponent.
- The winning strategy can be extracted
  from the fixed point computation.

<u>Illustration:</u>



○ := Player <u>I</u>
□ := Player <u>II</u>
/// := Checkmate
---> := Winning strategy

<u>Goal:</u> . Study the complexity of deciding
whether a position is winning for Player I.

Note: We do asymptotic complexity,
so we need a way to generalize
games to arbitrary size.
What does it mean to play Chess on an $n \times n$ board?
This has been done for many games.

Definition (Generalized Geography):
An instance of Generalized Geography
is a triple (V, E, s), where (V, E) is a directed graph
with
↳ V = countries, $s \in V$ = initial country
↳ E = edges,
↳ Even stages = Player I
Odd stages = Player II.
No player may move to a vertex
that has already been visited.

Theorem (Stockmeyer & Chandra '79):
Generalized Geography is PSPACE-complete under $\leq_m^{\log}$.

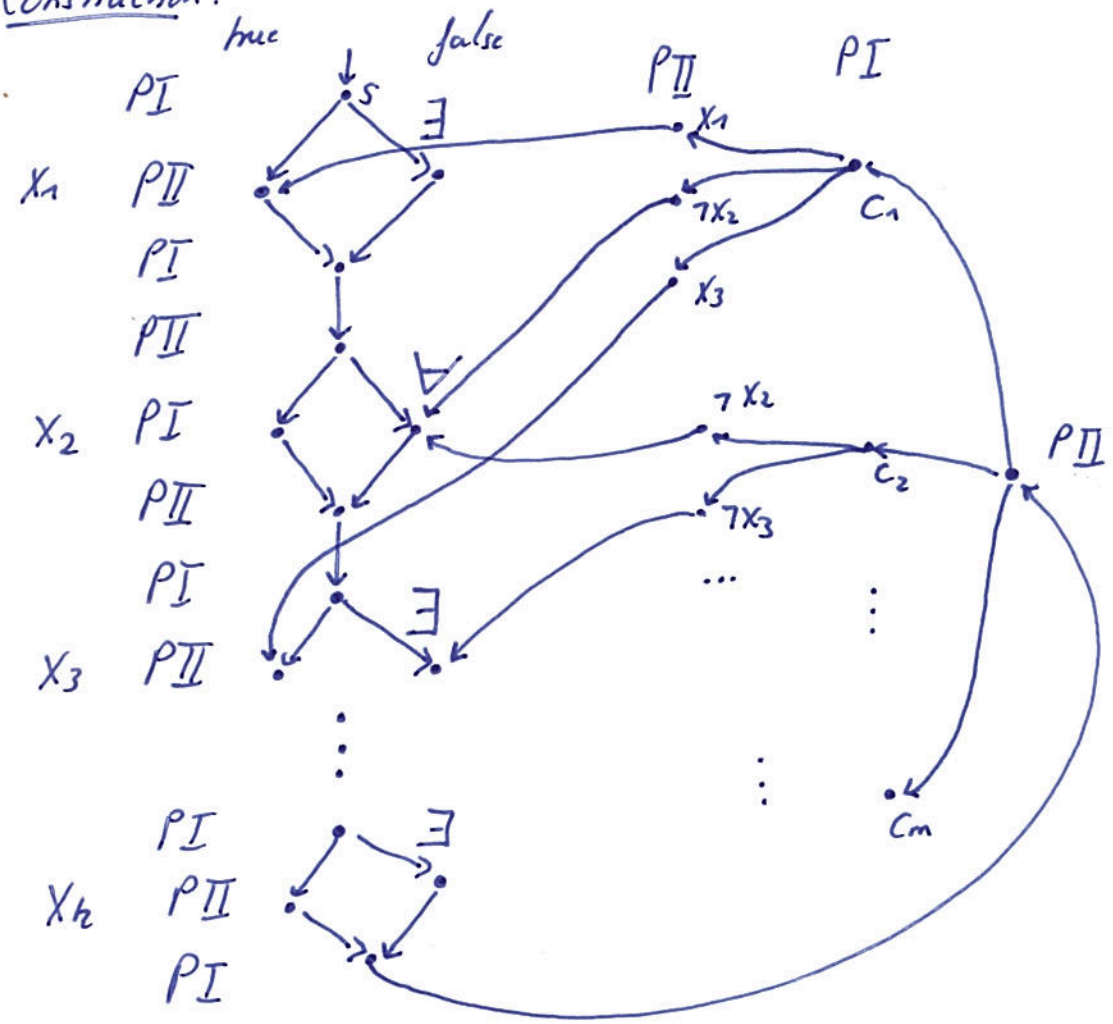
Proof:
PSPACE-hardness:
We reduce QBF to Generalized Geography.
We assume the given QBF instance takes the shape

$$F = \underbrace{\exists x_1 \forall x_2 \dots \exists x_k}_{\text{Begin and end w.r.t. } \exists,\ \text{strictly alternate.}} . \underbrace{G}_{\text{Assumed to be in CNF.}}$$

The game instance that we construct is such that

F is true iff Player I has a winning strategy from the initial position.

Construction:



where $G = \underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{C_1} \wedge \underbrace{(\neg x_2 \vee \neg x_3 \vee \ldots)}_{C_2} \wedge \ldots \wedge \underbrace{(\ldots)}_{C_m}$

Membership in PSPACE:

The following recursive algorithm decides whether Player I has a winning strategy:

```
Bool solve (input (G, b)):
    if b has outdegree 0 then
        return false; // Player I loses
```

```
else
    G' := G with b and all arrows connected to it removed;
    Bool result := true;
    for all b1 ... bm that b pointed to do
        result := result ∧ solve(G', bi);
    end for all
    if result = true then                  // Player II has a winning strategy
        return false;                      // for each bi
    else
        return true;
```

Space requirement:
Recursion stack with each level holding one node
(we do not create copies of G).

$\Rightarrow$ Linear space. □

Note: We do not have methods to quantify the complexity of finite games.

12.3 Language-theoretic Problems

Goal: Show that interesting problems on regular languages are PSPACE-complete.

Theorem:
Universality (Given an NFA N, does $L(N) = \Sigma^*$ hold?)
is PSPACE-complete under logspace reductions.

Corollary:

- Equivalence (Given NFAs A and B, does $L(A) = L(B)$ hold?) is PSPACE-complete.
- Inclusion (Given NFAs A and B, does $L(A) \subseteq L(B)$ hold?) is PSPACE-complete.

Proof:

Universality $\in$ PSPACE:

Idea: Guess a word that is not accepted.

Technically:
- Start with a pebble on the initial state of A.
- Guess a symbol a and move the current pebbles to all states reachable via a.
- We accept if we get to a configuration with no pebble on an accepting state.

Essentially: The procedure determinizes A along the guessed word. We do not need to remember the word explicitly, just the states that A could be in.

Complexity: The procedure shows
Universality $\in$ co-NPSPACE = PSPACE

Each configuration is a subset of the states
$\Rightarrow$ Bitvector of linear size.

Universality is PSPACE-hard:

Let N be a deterministic, n^k-space-bounded TM.
Assume N has a unique accept configuration.

Idea: Given input x, we construct an NFA M with $O(n^k)$ states accepting all strings that are

not accepting computations of N on x.

Then

$L(M) = \Sigma^*$ iff N does not accept x.

(So we reduce $L(N)$ to $\{M \mid L(M) \neq \Sigma^*\}$).

Construction:

- An accepting computation of N on input x of length $|x| = n$ is a string of the form

$$\#\alpha_0\#\alpha_1\#\ldots\#\alpha_m\# \qquad (1)$$

where

↳ each α_i is a string of length n^k over some alphabet Δ (tape symbols and tape symbols plus TM states)

↳ and each α_i encodes a configuration of N on input x.

Moreover, the sequence satisfies

(2) α_0 is the start configuration of N on input x.

(3) α_m is the accept configuration of N on input x.

(4) Each α_{i+1} is a successor of α_i according to the transition relation of N.

- If a sequence is not an accepting computation, it is not of the form (1), or one of the conditions (2), (3), (4) is violated.
- The NFA guesses which of these failures to check.

Checking (1):

Means to check that the word

is not in $(\# \Delta^{n^k})^* \#$ + some formal checks
(one state per configuration,
begin marker etc.)

This needs $O(n^k)$ states for A.

Checking (2) and (3):

Means checking that the word
begins and ends with a certain string of length n^k.
This can also be encoded into A with $O(n^k)$ states.

Checking (4):

- Recall from Ladner's theorem
that there is a finite set of local conditions
involving
  - the $(j-1)$st, jth, and $(j+1)$st symbol of α_i
  - and the jth symbol of α_{i+1},

so that
(4) holds iff these conditions are met for all $1 \leq j \leq n^k$.

- Hence, to check that (4) is violated,
automaton A scans across the input
and guesses non-deterministically where the violation occurs:
  - ↳ Remembers the next three symbols in its finite control.
  - ↳ Skips over the next n^k symbols.
  - ↳ Accepts if the following symbol is not correct.

The construction can be done with logarithmic space. □