Complexity Theory

Games/Logarithmic Space

Daniel Borchmann, Markus Krötzsch

Computational Logic

2015-12-02

Review

Games

Games as Computational Problems

Many single-player games relate to NP-complete problems:

- Sudoku
- Minesweeper
- Tetris
- ...

Decision problem: Is there a solution?

(For Tetris: is it possible to clear all blocks?)

What about two-player games?

Games as Computational Problems

Many single-player games relate to NP-complete problems:

- Sudoku
- Minesweeper
- Tetris
- …

Decision problem: Is there a solution?

(For Tetris: is it possible to clear all blocks?)

What about two-player games?

- Two players take moves in turns
- The players have different goals
- The game ends if a player wins

Decision problem: Does Player 1 have a winnings strategy?

In other words: can Player 1 enforce winning, whatever Player 2 does?

Example: The Formula Game

A contrived game, to illustrate the idea:

- Given: a propositional logic formula φ with consecutively numbered variables $X_1, \ldots X_\ell$.
- Two players take turns in selecting values for the next variable:
  - Player 1 sets X_1 to true or false
  - Player 2 sets X_2 to true or false
  - Player 1 sets X_3 to true or false
  - ...

  until all variables are set.
- Player 1 wins if the assignment makes φ true.
  Otherwise, Player 2 wins.

Deciding the Formula Game

FORMULA GAME

Input: A formula φ.

Problem: Does Player 1 have a winning strategy on φ?

Theorem 12.1

FORMULA GAME *is* $\mathrm{PS{\small PACE}}$*-complete.*

Deciding the Formula Game

FORMULA GAME

Input: A formula φ.

Problem: Does Player 1 have a winning strategy on φ?

Theorem 12.1

FORMULA GAME *is* PSPACE*-complete.*

Proof sketch.

FORMULA GAME is essentially the same as TRUE QBF.

Having a winning strategy means: there is a truth value for X_1, such that, for all truth values of X_2, there is a truth value of X_3, . . . such that φ becomes true.

If we have a QBF where quantifiers do not alternate, we can add dummy quantifiers and variables that do not change the semantics to get the same alternating form as for the Formula Game. □

Example: The Geography Game

A children's game:

- Two players are taking turns naming cities.
- Each city must start with the last letter of the previous.
- Repetitions are not allowed.
- The first player who cannot name a new city looses.

Example: The Geography Game

A children's game:

- Two players are taking turns naming cities.
- Each city must start with the last letter of the previous.
- Repetitions are not allowed.
- The first player who cannot name a new city looses.

A mathematicians' game:

- Two players are marking nodes on a directed graph.
- Each node must be a successor of the previous one.
- Repetitions are not allowed.
- The first player who cannot mark a new node looses.

Decision problem (GENERALISED) GEOGRAPHY:
given a graph and start node, does Player 1 have a winning strategy?

GEOGRAPHY is PSPACE-complete

Theorem 12.2

GENERALISED GEOGRAPHY *is* PSPACE*-complete.*

Proof.

- GEOGRAPHY $\in$ PSPACE:
  Give algorithm that runs in polynomial space.
  It is not difficult to provide a recursive algorithm similar to the one for TRUE QBF or FOL MODEL CHECKING.
- GEOGRAPHY is PSPACE-hard:
  Proof by reduction FORMULA GAME $\leq_p$ GEOGRAPHY.

GEOGRAPHY is PSPACE-hard

Let φ with variables $X_1, \ldots, X_\ell$ be an instance of FORMULA GAME.
Without loss of generality, we assume:

- ℓ is odd (Player 1 gets the first and last turn)
- φ is in CNF

We now build a graph that encodes FORMULA GAME in terms of GEOGRAPHY

- The left-hand side of the graph is a chain of diamond structures that represent the choices that players have when assigning truth values
- The right-hand side of the graph encodes the structure of φ: Player 2 may choose a clause (trying to find one that is not true under the assignment); Player 1 may choose a literal (trying to find one that is true under the assignment).

(see board or [Sipser, Theorem 8.14]) □

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?

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
  $\leadsto$ generalise games to arbitrarily large boards
  - generalised Tic-Tac-Toe is PSPACE-complete
  - generalised Reversi (Othello) is PSPACE-complete

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
  $\leadsto$ generalise games to arbitrarily large boards
  - generalised Tic-Tac-Toe is PSPACE-complete
  - generalised Reversi (Othello) is PSPACE-complete
- Issue 2: (generalised) games where moves can be reversed may require very long matches
  $\leadsto$ such games often are even harder
  - generalised Go is EXPTIME-complete
  - generalised Draughts (Checkers) is EXPTIME-complete
  - generalised Chess is EXPTIME-complete

Logarithmic Space

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 SAT.

Sub-linear space

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

Recall:

$$\mathrm{L} = \textsc{LogSpace} = \textsc{DSpace}(\log n)$$
$$\mathrm{NL} = \textsc{NLogSpace} = \textsc{NSpace}(\log n)$$

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

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.

Examples: Problems in L

Example 12.3

The language $\{0^n 1^n \mid n \geq 0\}$ is in L.

Examples: Problems in L

Example 12.3

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

□

Examples: Problems in L

PALINDROMES

Input: Word w on some input alphabet Σ

Problem: Does w read the same forward and backward?

Example 12.4

PALINDROMES $\in \mathrm{L}$.

Examples: Problems in L

PALINDROMES

Input: Word w on some input alphabet Σ

Problem: Does w read the same forward and backward?

Example 12.4

PALINDROMES $\in \mathrm{L}$.

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: A Problem in NL

REACHABILITY *a.k.a.* STCON *a.k.a.* PATH

Input: Directed graph G, vertices $s, t \in V(G)$

Problem: Does G contain a path from s to t?

Example 12.5

REACHABILITY $\in \mathrm{NL}$.

Example: A Problem in NL

REACHABILITY *a.k.a.* STCON *a.k.a.* PATH

Input: Directed graph G, vertices $s, t \in V(G)$

Problem: Does G contain a path from s to t?

Example 12.5

REACHABILITY $\in \mathrm{NL}$.

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 REACHABILITY

More formally:

```
CanReach(G,s,t) :
   c := |V(G)| // counter
   p := s      // pointer
   while c > 0 :
      if p = t :
         return TRUE
      else :
         nondeterministically select G-successor p′ of p
         p := p′
         c := c − 1
   // eventually, if no success:
   return FALSE
```

Defining Reductions in Logarithmic Space

To compare the difficulty of problems in P or NL, polynomial-time reductions are useless.

Defining Reductions in Logarithmic Space

To compare the difficulty of problems in P or NL, polynomial-time reductions are useless.

Definition 12.6

A log-space transducer $\mathcal{M}$ 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.

$\mathcal{M}$ computes a function $f : \Sigma^* \to \Sigma^*$, where $f(w)$ is the content of the output tape of $\mathcal{M}$ running on input w when $\mathcal{M}$ halts.

f is called a log-space computable function.

Log-Space Reductions and NL-Completeness

Definition 12.7

A log-space reduction from $\mathcal{L} \subseteq \Sigma^*$ to $\mathcal{L}' \subseteq \Sigma^*$ is a log-space computable function $f : \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*$:

$$w \in \mathcal{L} \iff f(w) \in \mathcal{L}'$$

We write $\mathcal{L} \leq_L \mathcal{L}'$ in this case.

Definition 12.8

A problem $\mathcal{L} \in \mathrm{NL}$ is complete for NL if every other language in NL is log-space reducible to $\mathcal{L}$.

Detour: P-completeness

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

Definition 12.9

A problem $\mathcal{L} \in \mathrm{P}$ is complete for P if every other language in P is log-space reducible to $\mathcal{L}$.

We will see some examples in later lectures . . .

An NL-Complete Problem

Theorem 12.10

REACHABILITY *is* NL*-complete.*

Proof idea.

Let $\mathcal{M}$ be a non-deterministic log-space TM deciding $\mathcal{L}$.

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 $\mathcal{M}$ and edges represent possible computational steps of $\mathcal{M}$ 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 $\mathcal{M}$ and w using a log-space transducer:

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

$w \in \mathcal{L}$ iff $\langle G, s, t\rangle \in$ REACHABILITY

(see also Sipser, Theorem 8.25) □

coNL

coNL

As for time, we consider complement classes for space.

Recall Definition 9.6:

For a complexity class C, we define $\text{coC} := \{\mathcal{L} : \overline{\mathcal{L}} \in \text{C}\}$.

Complement classes for space:

- $\text{coNL} := \{\mathcal{L} : \overline{\mathcal{L}} \in \text{NL}\}$
- $\text{coNPSpace} := \{\mathcal{L} : \overline{\mathcal{L}} \in \text{NPSpace}\}$

From Savitch's theorem:

$\text{PSpace} = \text{NPSpace}$ and hence $\text{coNPSpace} = \text{PSpace}$, 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 simutaneously in 1987

The Immerman-Szelepcsényi Theorem

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

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

Theorem 12.11 (Immerman 1987/Szelepcsényi 1987)

$\mathrm{NL} = \mathrm{coNL}$.

Proof.

Show that $\overline{\text{Reachability}}$ is in NL.

Remark: alternative explanations provided by

- Sipser (Theorem 8.27)
- Dick Lipton's blog entry We All Guessed Wrong (link)
- Wikipedia Immerman–Szelepcsényi theorem

Towards Nondeterminsitic Nonreachability

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

Towards Nondeterminsitic Nonreachability

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

Initial idea:

```
NaiveNonReach(G, s, t) :
  for each vertex v of G :
    if CanReach(G, s, v) and v = t :
      return FALSE
  // eventually, if FALSE was not returned above:
  return TRUE
```

Does this work?

Towards Nondeterminsitic Nonreachability

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

Initial idea:

```
NaiveNonReach(G, s, t) :
  for each vertex v of G :
    if CanReach(G, s, v) and v = t :
      return FALSE
  // eventually, if FALSE was not returned above:
  return TRUE
```

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.

Towards Nondeterminsitic Nonreachability

Things would be different if we knew
the number *count* of vertices reachable from s:

```
CountingNonReach(G, s, t, count) :
  reached := 0
  for each vertex v of G :
    if CanReach(G, s, v) :
      reached := reached + 1
      if v = t :
        return FALSE
  // eventually, if FALSE was not returned above:
  return (count = reached)
```

Problem: how can we know *count*?

Counting Reachable Vertices – Intuition

Idea:

- Count number of vertices reachable in at most *length* steps
  - we call this number $count_{length}$
  - then the number we are looking for is $count = count_{|V(G)|-1}$
- Use a limited-length reachability test:
  CanReach($G, s, v, length$): "t reachable from s in G in $\leq$ *length* steps"
  (we actually implemented CanReach(G, s, v) as CanReach($G, s, v, |V(G)| - 1$))
- Compute the count iteratively, starting with $length = 0$ steps:
  - for $length > 0$, 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 $count_{length}$ was computed before)

Counting Reachable Vertices – Algorithm

The count for *length* $= 0$ is 1. For *length* > 0, we compute as follows:

```
01 CountReachable(G, s, length, count_{length-1}) :
02    count := 1 // we always count s
03    for each vertex u of G such that u ≠ s :
04       reached := 0
05       for each vertex v of G :
06          if CanReach(G, s, v, length − 1) :
07             reached := reached + 1
08             if G has an edge v → u :
09                count := count + 1
10                GOTO 03 // continue with next u
11       if reached < count_{length-1} :
12          REJECT // whole algorithm fails
13    return count
```

Completing the Proof of NL = coNL

Putting the ingredients together:

```
NonReachable(G, s, t) :
   count := 1 // number of nodes reachable in 0 steps
   for ℓ := 1 to |V(G)| − 1 :
      count_prev := count
      count := CountReachable(G, s, ℓ, count_prev)
   return CountingNonReach(G, s, t, count)
```

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