Dynamic Reasoning

Neil Immerman

College of Information and Computer Sciences

UMass Amherst

people.cs.umass.edu/~immerman/

Logic in Computer Science

- Descriptive Complexity

Logic in Computer Science

- Descriptive Complexity
- Dichotomy

Logic in Computer Science

- Descriptive Complexity
- Dichotomy
- Dynamic Complexity

Logic in Computer Science

- Descriptive Complexity
- Dichotomy
- Dynamic Complexity
- SAT Solvers

Logic in Computer Science

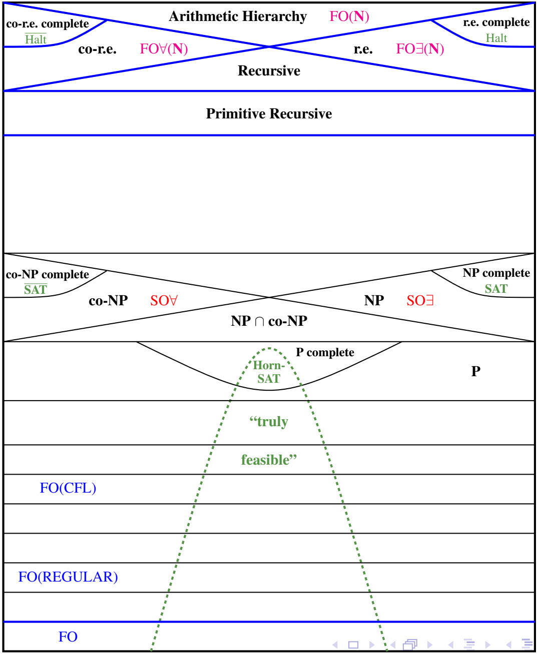
- Descriptive Complexity
- Dichotomy
- Dynamic Complexity
- SAT Solvers
- Computer Software: Crisis and Opportunity

Logic in Computer Science

- Descriptive Complexity
- Dichotomy
- Dynamic Complexity
- SAT Solvers
- Computer Software: Crisis and Opportunity

Personal perspective

$$P = \bigcup_{k=1}^{\infty} \mathrm{DTIME}[n^k]$$

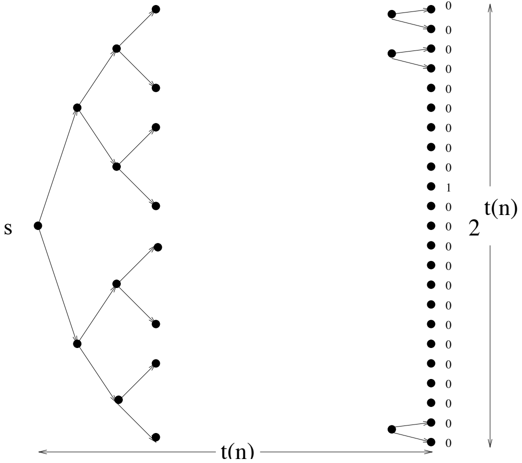


P is a good mathematical wrapper for "truly feasible".

NTIME[$t(n)$]: a mathematical fiction

input w

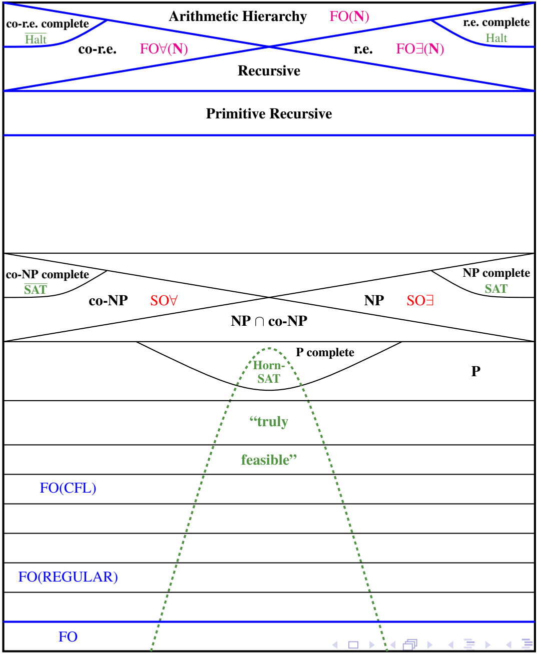
$|w| = n$



b_1 b_2 b_3 $\cdots$ $b_{t(n)-1}$

$$\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}[n^k]$$

Many optimization problems we want to solve are NP complete.



Descriptive Complexity

Query $\mapsto$ **Computation** $\mapsto$ **Answer**

$q_1\ q_2\ \cdots\ q_n$ $\qquad$ $a_1\ a_2\ \cdots\ a_i\ \cdots\ a_m$

Descriptive Complexity

Query $\mapsto$ **Computation** $\mapsto$ **Answer**

$q_1 \; q_2 \; \cdots \; q_n \quad\quad a_1 \; a_2 \; \cdots \; a_i \; \cdots \; a_m$

Restrict attention to the complexity of computing individual bits of the output, i.e., **decision problems**.

Descriptive Complexity

Query $\mapsto$ **Computation** $\mapsto$ **Answer**

$q_1 \; q_2 \; \cdots \; q_n$ $\qquad$ $a_1 \; a_2 \; \cdots \; a_i \; \cdots \; a_m$

$\cdots \; S_i \; \cdots$

Restrict attention to the complexity of computing individual bits of the output, i.e., **decision problems**.

How hard is it to **check** if input has property S ?

Descriptive Complexity

Query ↦ **Computation** ↦ **Answer**

$q_1\ q_2\ \cdots\ q_n$ ↦ ↦ $a_1\ a_2\ \cdots\ a_i\ \cdots\ a_m$

$\cdots\ S_i\ \cdots$

Restrict attention to the complexity of computing individual bits of the output, i.e., **decision problems**.

How hard is it to **check** if input has property S ?

How rich a language do we need to **express** property S?

Descriptive Complexity

Query $\mapsto$ **Computation** $\mapsto$ **Answer**

$q_1\ q_2\ \cdots\ q_n$ $\qquad\qquad$ $a_1\ a_2\ \cdots\ a_i\ \cdots\ a_m$

$\cdots\ S_i\ \cdots$

Restrict attention to the complexity of computing individual bits of the output, i.e., **decision problems**.

How hard is it to **check** if input has property S ?

How rich a language do we need to **express** property S?

There is a constructive isomorphism between these two approaches.

Interpret Input as Finite Logical Structure

Graph $\quad G = (\{v_1, \ldots, v_n\}, E, s, t)$

Binary String

$$\mathcal{A}_w = (\{p_1, \ldots, p_8\}, S)$$
$$S = \{p_2, p_5, p_7, p_8\}$$
$$w = 01001011$$

Vocabularies: $\tau_g = (E^2, s, t), \quad \tau_s = (S^1)$

First-Order Logic

input symbols:	from τ
variables:	$x, y, z, \ldots$
boolean connectives:	$\wedge, \vee, \neg$
quantifiers:	$\forall, \exists$
numeric symbols:	$=, \leq, +, \times, \min, max$

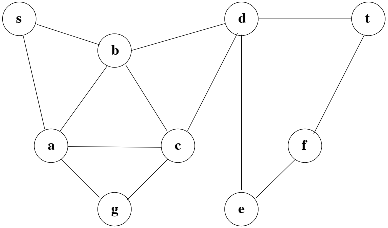
$$\alpha \equiv \forall x \exists y (E(x,y)) \in \mathcal{L}(\tau_g)$$

$$\beta \equiv \exists x \forall y (x \leq y \wedge S(x)) \in \mathcal{L}(\tau_s)$$

$$\beta \equiv S(\min) \in \mathcal{L}(\tau_s)$$

Second-Order Logic

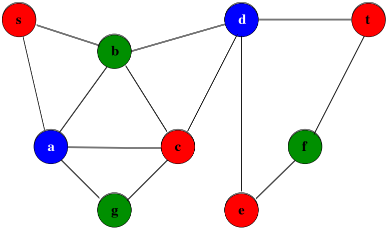
$$\Phi_{3-\mathrm{color}} \equiv \exists R^1\, G^1\, B^1\, \forall x\, y\, ((R(x) \vee G(x) \vee B(x)) \wedge (E(x,y) \rightarrow (\neg(R(x) \wedge R(y)) \wedge \neg(G(x) \wedge G(y)) \wedge \neg(B(x) \wedge B(y)))))$$



Second-Order Logic

Fagin's Theorem: NP = SO∃

$$\Phi_{3-\text{color}} \equiv \exists R^1 G^1 B^1 \forall x\, y\, ((R(x) \vee G(x) \vee B(x)) \wedge$$
$$(E(x,y) \rightarrow (\neg(R(x) \wedge R(y)) \wedge \neg(G(x) \wedge G(y))$$
$$\wedge \quad \neg(B(x) \wedge B(y)))))$$



Addition is First-Order

$$Q_+ : \mathrm{STRUC}[\tau_{AB}] \to \mathrm{STRUC}[\tau_s]$$

$$\begin{array}{llllll} A & & a_1 & a_2 & \ldots & a_{n-1} & a_n \\ B & + & b_1 & b_2 & \ldots & b_{n-1} & b_n \\ \hline S & & s_1 & s_2 & \ldots & s_{n-1} & s_n \end{array}$$

Addition is First-Order

$$Q_+ : \mathrm{STRUC}[\tau_{AB}] \to \mathrm{STRUC}[\tau_s]$$

$$\begin{array}{llcccccc} A & & a_1 & a_2 & \ldots & a_{n-1} & a_n \\ B & + & b_1 & b_2 & \ldots & b_{n-1} & b_n \\ \hline S & & s_1 & s_2 & \ldots & s_{n-1} & s_n \end{array}$$

$$C(i) \equiv (\exists j > i)\Big(A(j) \wedge B(j) \wedge (\forall k.j > k > i)(A(k) \vee B(k))\Big)$$

Addition is First-Order

$$Q_+ : \mathrm{STRUC}[\tau_{AB}] \to \mathrm{STRUC}[\tau_s]$$

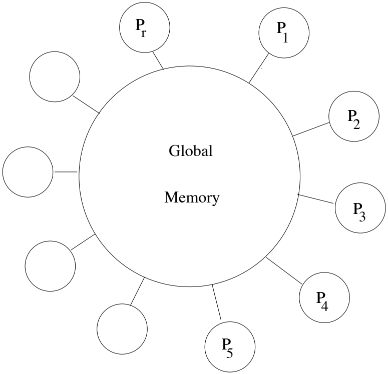
$$\begin{array}{llccccc} A & & a_1 & a_2 & \ldots & a_{n-1} & a_n \\ B & + & b_1 & b_2 & \ldots & b_{n-1} & b_n \\ \hline S & & s_1 & s_2 & \ldots & s_{n-1} & s_n \end{array}$$

$$C(i) \equiv (\exists j > i)\Big(A(j) \wedge B(j) \wedge (\forall k.j > k > i)(A(k) \vee B(k))\Big)$$

$$Q_+(i) \equiv A(i) \oplus B(i) \oplus C(i)$$

Parallel Machines:

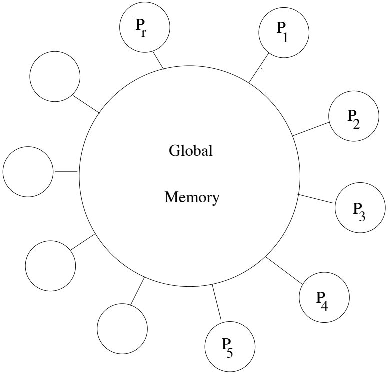
$\text{CRAM}[t(n)] \; = \text{CRCW-PRAM-TIME}[t(n)]\text{-HARD}[n^{O(1)}]$



Parallel Machines: Quantifiers are Parallel

$\text{CRAM}[t(n)] \; = \text{CRCW-PRAM-TIME}[t(n)]\text{-HARD}[n^{O(1)}]$

Assume array $A[x] : x = 1, \ldots, r$ in memory.

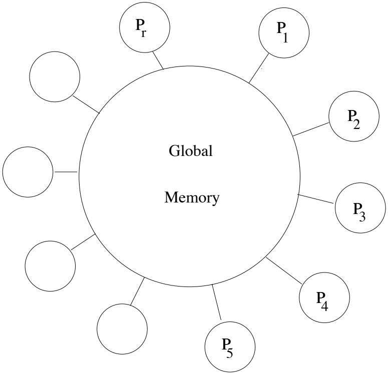


Parallel Machines: Quantifiers are Parallel

$\text{CRAM}[t(n)] = \text{CRCW-PRAM-TIME}[t(n)]\text{-HARD}[n^{O(1)}]$

Assume array $A[x] : x = 1, \ldots, r$ in memory.

$$\forall x(A(x)) \equiv \textbf{write}(1);\ \text{proc } p_i : \textbf{if } (A[i] = 0) \textbf{ then } \textbf{write}(0)$$



FO

=

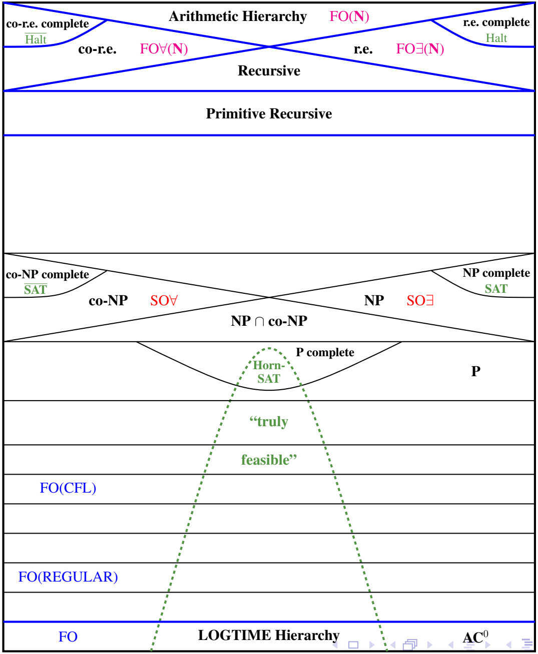
CRAM[1]

=

AC^0

=

Logarithmic-Time Hierarchy



$$\mathrm{CRAM}[t(n)] = \text{concurrent parallel random access machine; polynomial hardware, parallel time } O(t(n))$$

$$\mathrm{IND}[t(n)] = \text{first-order, depth } t(n) \text{ inductive definitions}$$

$$\mathrm{FO}[t(n)] = t(n) \text{ repetitions of a block of restricted quantifiers:}$$

$$\mathrm{QB} = [(Q_1 x_1.M_1)\cdots(Q_k x_k.M_k)]; \quad M_i \text{ quantifier-free}$$

$$\varphi_n = \underbrace{[\mathrm{QB}][\mathrm{QB}]\cdots[\mathrm{QB}]}_{t(n)} M_0$$

parallel time = inductive depth = QB iteration

Thm: For all constructible, polynomially bounded $t(n)$,

$$\mathrm{CRAM}[t(n)] \;=\; \mathrm{IND}[t(n)] \;=\; \mathrm{FO}[t(n)]$$

Thm: For all $t(n)$, even beyond polynomial,

$$\mathrm{CRAM}[t(n)] \quad = \quad \mathrm{FO}[t(n)]$$

For $t(n)$ poly bdd,

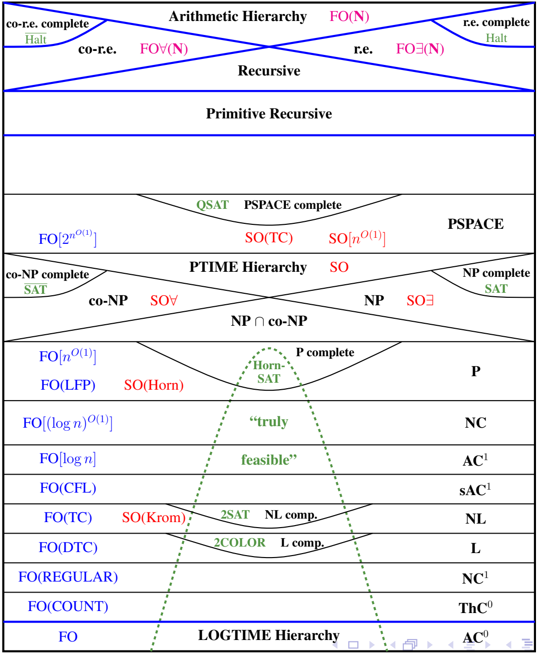
$\mathrm{CRAM}[t(n)]$

$=$

$\mathrm{IND}[t(n)]$

$=$

$\mathrm{FO}[t(n)]$



Recent Breakthroughs in Descriptive Complexity

Theorem [Ben Rossman] Any first-order formula with any numeric relations $(\leq, +, \times, \ldots)$ that means "I have a clique of size k" must have at least $k/4$ variables.

- Creative new proof idea using Håstad's Switching Lemma gives the essentially optimal bound.
- First lower bound of its kind for number of variables with ordering.
- This lower bound is for a fixed formula, if it were for a sequence of polynomially-sized formulas, it would show that $\mathrm{CLIQUE} \notin \mathrm{P}$ and thus $\mathrm{P} \neq \mathrm{NP}$.

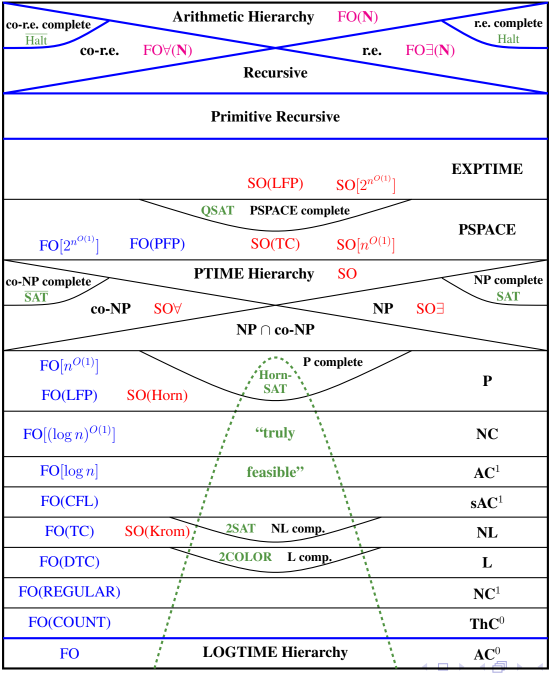
Recent Breakthroughs in Descriptive Complexity

Theorem [Martin Grohe] Fixed-Point Logic with Counting captures Polynomial Time on all classes of graphs with excluded minors.

Grohe proves that for every class of graphs with excluded minors, there is a constant k such that two graphs of the class are isomorphic iff they agree on all k-variable formulas in fixed-point logic with counting.

Thus every class of graphs with excluded minors admits the same general polynomial time canonization algorithm: we're isomorphic iff we agree on all formulas in C_k and in particular, you are isomorphic to me iff your C_k canonical description is equal to mine.

See: **"The Nature and Power of Fixed-Point Logic with Counting"** by **Anuj Dawar** in *SigLog Newsletter*.



Arithmetic Hierarchy FO(N)
co-r.e. complete
$\overline{\text{Halt}}$
co-r.e. FO∀(N)
r.e. complete
Halt
r.e. FO∃(N)
Recursive
Primitive Recursive
EXPTIME
SO(LFP) $SO[2^{n^{O(1)}}]$
QSAT PSPACE complete
PSPACE
$FO[2^{n^{O(1)}}]$ FO(PFP) SO(TC) $SO[n^{O(1)}]$
PTIME Hierarchy SO
co-NP complete
$\overline{\text{SAT}}$
co-NP SO∀
NP complete
SAT
NP SO∃
NP ∩ co-NP
$FO[n^{O(1)}]$
P complete
Horn-SAT
FO(LFP) SO(Horn)
P
$FO[(\log n)^{O(1)}]$ "truly
NC
$FO[\log n]$ feasible"
AC^1
FO(CFL)
sAC^1
FO(TC) SO(Krom) 2SAT NL comp.
NL
FO(DTC) 2COLOR L comp.
L
FO(REGULAR)
NC^1
FO(COUNT)
ThC^0
FO LOGTIME Hierarchy
AC^0

Dichotomy

- "Natural" Computational Problems Tend to be Complete for Important Complexity Classes

Dichotomy

- "Natural" Computational Problems Tend to be Complete for Important Complexity Classes
- Isomorphism Theorem: only one such problem in each class: **small handful of naturally occuring decision problems!**

Dichotomy

- "Natural" Computational Problems Tend to be Complete for Important Complexity Classes
- Isomorphism Theorem: only one such problem in each class: **small handful of naturally occuring decision problems!**
- Not true for "unnatural problems": Ladner's Delayed Diagonalization

Dichotomy

- "Natural" Computational Problems Tend to be Complete for Important Complexity Classes
- Isomorphism Theorem: only one such problem in each class: **small handful of naturally occuring decision problems!**
- Not true for "unnatural problems": Ladner's Delayed Diagonalization
- Schaefer; Feder-Vardi: CSP Dichotomy Conjecture

Dichotomy

- "Natural" Computational Problems Tend to be Complete for Important Complexity Classes
- Isomorphism Theorem: only one such problem in each class: **small handful of naturally occuring decision problems!**
- Not true for "unnatural problems": Ladner's Delayed Diagonalization
- Schaefer; Feder-Vardi: CSP Dichotomy Conjecture
- Tremendous progress using Universal Algebra. (Solved for domains of size 2 and 3, and for undirected graphs.) See: **"Constraint Satisfaction Problem and Universal Algebra"** by **Libor Barto** in *SigLog Newsletter*.

Dynamic Complexity

Static

1. Read entire input
2. Compute boolean query **Q**(input)
3. Classic Complexity Classes are static: FO, NC, P, NP, . . .

Dynamic Complexity

Static

1. Read entire input
2. Compute boolean query **Q**(input)
3. Classic Complexity Classes are static: FO, NC, P, NP, . . .
4. What is the fastest way **upon reading the entire input**, to compute the query?

Dynamic Complexity

Static

1. Read entire input
2. Compute boolean query **Q**(input)
3. Classic Complexity Classes are static: FO, NC, P, NP, . . .
4. What is the fastest way **upon reading the entire input**, to compute the query?

Dynamic

1. Long series of Inserts, Deletes, Changes, and, Queries
2. On **query**, very quickly compute **Q**(current database)
3. Dynamic Complexity Classes: Dyn-FO, Dyn-NC

Dynamic Complexity

Static

1. Read entire input
2. Compute boolean query **Q**(input)
3. Classic Complexity Classes are static: FO, NC, P, NP, . . .
4. What is the fastest way **upon reading the entire input**, to compute the query?

Dynamic

1. Long series of Inserts, Deletes, Changes, and, Queries
2. On **query**, very quickly compute **Q**(current database)
3. Dynamic Complexity Classes: Dyn-FO, Dyn-NC
4. What **additional information** should we maintain? — **auxiliary data structure**

Dynamic (Incremental) Applications

- Databases
- LaTexing a file
- Performing a calculation
- Processing a visual scene
- Understanding a natural language
- Verifying a circuit
- Verifying and compiling a program
- Surviving in the wild

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
	ins(3,S)	

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
0010000	**ins**(3,S)	1

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
0010000	**ins**(3,S)	1
	ins(7,S)	

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
0010000	**ins**(3,S)	1
0010001	**ins**(7,S)	0

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
0010000	**ins**(3,S)	1
0010001	**ins**(7,S)	0
	del(3,S)	

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
0010000	**ins**(3,S)	1
0010001	**ins**(7,S)	0
0000001	**del**(3,S)	1

Parity

Current Database: S	Request	Auxiliary Data: b
0000000		0
0010000	**ins**(3,S)	1
0010001	**ins**(7,S)	0
0000001	**del**(3,S)	1

ins(a,S)

$$S'(x) \equiv S(x) \vee x = a$$

$$b' \equiv (b \wedge S(a)) \vee (\neg b \wedge \neg S(a))$$

del(a,S)

$$S'(x) \equiv S(x) \wedge x \neq a$$

$$b' \equiv (b \wedge \neg S(a)) \vee (\neg b \wedge S(a))$$

Dynamic Examples

Parity

- Does binary string w have an odd number of 1's?
- **Static:** TIME[n], FO[$\Omega(\log n/\log\log n)$]
- **Dynamic:** Dyn-TIME[1], Dyn-FO

Dynamic Examples

Parity

- Does binary string w have an odd number of 1's?
- **Static:** TIME[n], FO[$\Omega(\log n / \log\log n)$]
- **Dynamic:** Dyn-TIME[1], Dyn-FO

REACH_u

- Is t reachable from s in undirected graph G?
- **Static:** not in FO, requires FO[$\Omega(\log n / \log\log n)$]
- **Dynamic:** in Dyn-FO [Patnaik, I]

Dynamic Examples

Parity

- Does binary string w have an odd number of 1's?
- **Static:** TIME$[n]$, FO$[\Omega(\log n/\log\log n)]$
- **Dynamic:** Dyn-TIME[1], Dyn-FO

REACH$_u$

- Is t reachable from s in undirected graph G?
- **Static:** not in FO, requires FO$[\Omega(\log n/\log\log n)]$
- **Dynamic:** in Dyn-FO [Patnaik, I]

Dynamic Examples

Parity

- Does binary string w have an odd number of 1's?
- **Static:** TIME[n], FO[$\Omega(\log n / \log\log n)$]
- **Dynamic:** Dyn-TIME[1], Dyn-FO

REACH$_u$

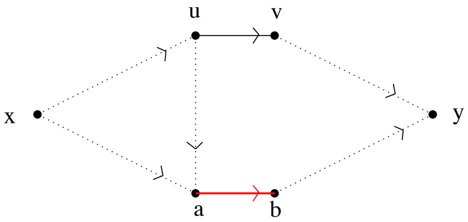
- Is t reachable from s in undirected graph G?
- **Static:** not in FO, requires FO[$\Omega(\log n / \log\log n)$]
- **Dynamic:** in Dyn-FO [Patnaik, I]

Minimum Spanning Trees, ***k*-edge connectivity**, ...

Fact: [Dong & Su] REACH(acyclic) $\in$ DynFO

ins$(a, b, E) : P'(x, y) \equiv P(x, y) \vee (P(x, a) \wedge P(b, y))$

del(a, b, E):



$$P'(x, y) \equiv P(x, y) \wedge \Big[\neg(P(x, a) \wedge P(b, y)) \\ \vee (\exists uv)\big(P(x, u) \wedge E(u, v) \wedge P(v, y) \\ \wedge P(u, a) \wedge \neg P(v, a) \wedge (a \neq u \vee b \neq v)\big)\Big]$$

REACHABILITY Problems

$$\mathrm{REACH} = \{G \mid G \text{ directed, } s \xrightarrow[G]{\star} t\} \qquad \mathrm{NL}$$

$$\mathrm{REACH}_d = \{G \mid G \text{ directed, outdegree} \leq 1\ s \xrightarrow[G]{\star} t\} \qquad \mathrm{L}$$

$$\mathrm{REACH}_u = \{G \mid G \text{ undirected, } s \xrightarrow[G]{\star} t\} \qquad \mathrm{L}$$

$$\mathrm{REACH}_a = \{G \mid G \text{ alternating, } s \xrightarrow[G]{\star} t\} \qquad \mathrm{P}$$

Facts about dynamic REACHABILITY Problems:

Dyn-REACH(acyclic)	$\in$	Dyn-FO	[DS]
Dyn-REACH_d	$\in$	Dyn-QF	[H]
Dyn-REACH_u	$\in$	Dyn-FO	[PI]
Dyn-REACH	$\in$	Dyn-FO(COUNT)	[H]
$\text{Dyn-PAD}(\text{REACH}_a)$	$\in$	Dyn-FO	[PI]

Exciting New Result

Reachability is in DynFO

by Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick and Thomas Zeume

`http://arxiv.org/abs/1502.07467`

They show that Matrix Rank is in DynFO and REACH reduces to Matrix Rank.

Thm. 1 [Hesse] Reachability of functional DAG is in DynQF.

proof: Maintain E, E^*, D (outdegree $= 1$).

Insert $E(i,j)$: (ignore if adding edge violates outdegree or acyclicity)

$$
\begin{aligned}
E'(x,y) &\equiv E(x,y) \lor (x = i \land y = j) \\
D'(x) &\equiv D(x) \lor x = i \\
E^{*\prime}(x,y) &\equiv E^*(x,y) \lor (E^*(x,i) \land E^*(j,y))
\end{aligned}
$$

Thm. 1 [Hesse] Reachability of functional DAG is in DynQF.

proof: Maintain E, E^*, D (outdegree $= 1$).

Insert $E(i,j)$: (ignore if adding edge violates outdegree or acyclicity)

$$\begin{aligned} E'(x,y) &\equiv E(x,y) \lor (x = i \land y = j) \\ D'(x) &\equiv D(x) \lor x = i \\ E^{*\prime}(x,y) &\equiv E^*(x,y) \lor (E^*(x,i) \land E^*(j,y)) \end{aligned}$$

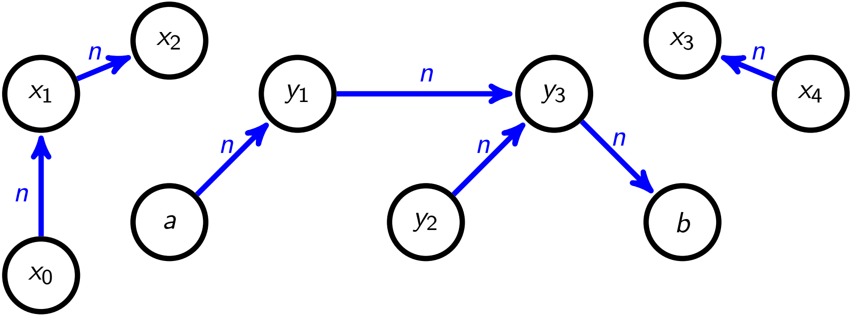
Delete $E(i,j)$:

$$\begin{aligned} E'(x,y) &\equiv E(x,y) \land (x \neq i \lor y \neq j) \\ D'(x) &\equiv D(x) \land (x \neq i \lor \neg E(i,j)) \\ E^{*\prime}(x,y) &\equiv E^*(x,y) \land \neg(E^*(x,i) \land E(i,j) \land E^*(j,y)) \end{aligned}$$

□

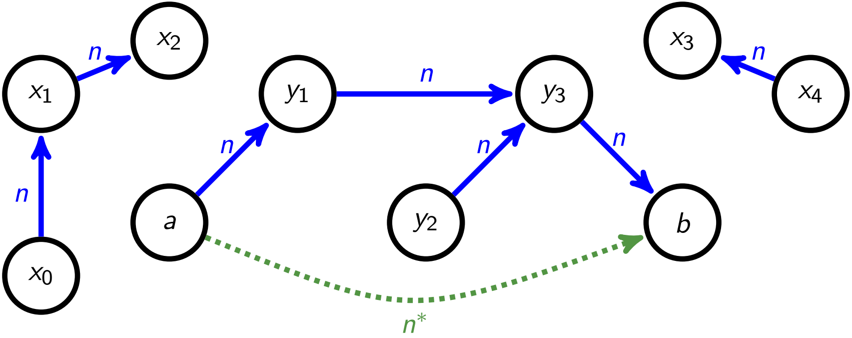
Dynamic Reasoning

Reasoning About reachability – can we get to b from a by following a sequence of pointers – is **crucial for proving that programs meet their specifications**.



Dynamic Reasoning

Reasoning About reachability – can we get to b from a by following a sequence of pointers – is **crucial for proving that programs meet their specifications**.



However reasoning about reachability in general is **undecidable**.

Ideas:

- Can express tilings and thus runs of Turing Machines.
- Even worse, can express **finite path** and thus **finite** and thus **standard natural numbers**. Thus $\mathrm{FO}(\mathrm{TC})$ is as hard as the Arithmetic Hierarchy [Avron].

For the time being, let's restrict ourselves to acyclic fields which thus also generate a linear ordering of all points reachable from a given point.

$$\begin{aligned}
\text{acyclic} &\equiv \forall xy\,(n^*(x,y) \wedge n^*(y,x) \;\rightarrow\; x = y) \\
\text{transitive} &\equiv \forall xyz\,(n^*(x,y) \wedge n^*(y,z) \;\rightarrow\; n^*(x,z))) \\
\text{linear} &\equiv \forall xyz\,(n^*(x,y) \wedge n^*(x,z) \;\rightarrow\; n^*(y,z) \vee n^*(z,y))
\end{aligned}$$

Effectively-Propositional Reasoning about Reachability in Linked Data Structures

- Automatically transform a program manipulating linked lists to an $\forall\exists$ correctness condition.
- Using Hesse's dynQF algorithm for REACH_d, is that these $\forall\exists$ formulas are closed under weakest precondition.
- Using acyclic, transitive and linear axioms, the negation of the correctness condition is equi-satisfiable with a propositional formula.
- use a SAT solver to automatically prove correctness or find counter-example runs, typically in under 3 seconds per program.

Thm. 2 [Hesse] Reachability of functional graphs is in DynQF.

proof idea: If adding an edge, e, would create a cycle, then we maintain relation P – the path relation without the edge completing the cycle – as well as E^*, E and D.

Surprisingly this can all be maintained via quantifier-free formulas, **without remembering which edges we are leaving out** in computing P. □

Thm. 2 [Hesse] Reachability of functional graphs is in DynQF.

proof idea: If adding an edge, e, would create a cycle, then we maintain relation P – the path relation without the edge completing the cycle – as well as E^*, E and D.

Surprisingly this can all be maintained via quantifier-free formulas, **without remembering which edges we are leaving out** in computing P. □

Using Thm. 2, the above methodology has been extended to cyclic deterministic graphs.

- Itzhaky, Banerjee, Immerman, Aleks Nanevski, Sagiv, "Effectively-Propositional Reasoning About Reachability in Linked Data Structures" CAV 2013.
- Itzhaky, Banerjee, Immerman, Lahav, Nanevski, Sagiv, "Modular Reasoning about Heap Paths via Effectively Propositional Formulas", POPL 2014

SAT Solvers

- SAT was the first NP Complete problem, thus **hard**.

SAT Solvers

- SAT was the first NP Complete problem, thus **hard**.
- Through amazing increases of computer speed and memory, plus terrific engineering and algorithmic ideas – clause learning, and good heuristics, SAT solvers are typically incredibly fast – seconds on formulas with a million variables.

SAT Solvers

- SAT was the first NP Complete problem, thus **hard**.
- Through amazing increases of computer speed and memory, plus terrific engineering and algorithmic ideas – clause learning, and good heuristics, SAT solvers are typically incredibly fast – seconds on formulas with a million variables.
- They provably aren't good on all instances, but they do extremely well in practice.

SAT Solvers

- SAT was the first NP Complete problem, thus **hard**.
- Through amazing increases of computer speed and memory, plus terrific engineering and algorithmic ideas – clause learning, and good heuristics, SAT solvers are typically incredibly fast – seconds on formulas with a million variables.
- They provably aren't good on all instances, but they do extremely well in practice.
- Thus we have a general purpose problem solver.

SAT Solvers

- SAT was the first NP Complete problem, thus **hard**.
- Through amazing increases of computer speed and memory, plus terrific engineering and algorithmic ideas – clause learning, and good heuristics, SAT solvers are typically incredibly fast – seconds on formulas with a million variables.
- They provably aren't good on all instances, but they do extremely well in practice.
- Thus we have a general purpose problem solver.
- Very useful for checking the correctness of programs, automatically finding counter-example runs, and for synthesizing good code from specifications.

Software Crisis

- Software is everywhere, controlling more and more of our lives.

Software Crisis

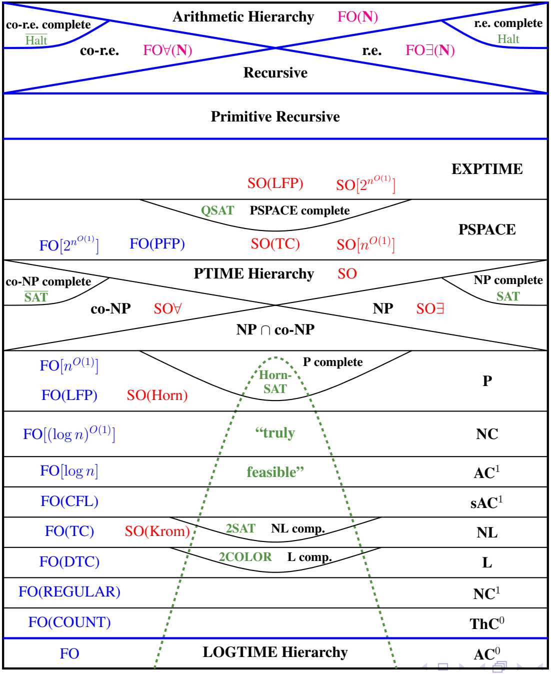
- Software is everywhere, controlling more and more of our lives.
- Software is buggy, insecure, brittle, hard to change.

Software Crisis

- Software is everywhere, controlling more and more of our lives.
- Software is buggy, insecure, brittle, hard to change.
- Logic and its application to automatic model checking and synthesis are – in my opinion – our best hope.

Software Crisis

- Software is everywhere, controlling more and more of our lives.
- Software is buggy, insecure, brittle, hard to change.
- Logic and its application to automatic model checking and synthesis are – in my opinion – our best hope.
- Thank you!



Arithmetic Hierarchy FO(N)
co-r.e. complete
Halt
co-r.e. FO∀(N)
r.e. complete
Halt
r.e. FO∃(N)
Recursive
Primitive Recursive
EXPTIME
SO(LFP) SO[$2^{n^{O(1)}}$]
QSAT PSPACE complete
PSPACE
FO[$2^{n^{O(1)}}$] FO(PFP) SO(TC) SO[$n^{O(1)}$]
PTIME Hierarchy SO
co-NP complete
SAT
co-NP SO∀
NP complete
SAT
NP SO∃
NP ∩ co-NP
FO[$n^{O(1)}$]
P complete
Horn-SAT
FO(LFP) SO(Horn)
P
FO[$(\log n)^{O(1)}$]
“truly
NC
FO[$\log n$]
feasible”
AC^1
FO(CFL)
sAC^1
FO(TC) SO(Krom)
2SAT NL comp.
NL
FO(DTC)
2COLOR L comp.
L
FO(REGULAR)
NC^1
FO(COUNT)
ThC^0
FO
LOGTIME Hierarchy
AC^0