SAT modulo Graphs: Acyclicity

Jussi Rintanen

Department of Information and Computer Science
Aalto University, Finland
(Also affiliated with Griffith University, Brisbane, Australia, and the Helsinki Institute of Information Technology, Finland.)

Joint work with *Martin Gebser and Tomi Janhunen*

September 2014

Motivation for the Work

Acyclicity is required in solutions of several important problems that can be reduced to the propositional satisfiability problem SAT.

- partial-order methods in planning (Rintanen et al. 2006) and bounded LTL model-checking
- well-foundedness of inductive definitions
- rule dependencies in answer set programming
- Bayesian networks, Markov networks (the structure learning problem) (Cussens 2008; Corander et al. 2013)

More General Motivation

Many graph-related concepts difficult to encode as propositional formulas (size, efficiency).

- Which nodes reachable from the source node?
- Which nodes on a simple path between a source node and sink node?

Application in e.g. networked systems' diagnosis, control, design: telecom, electricity, water, transport

A New Class of SAT Modulo Theories Problems

SAT modulo Graphs (SMG)

standard SAT problem (a set of clauses) $+$

- set of nodes
- set of edges/arcs
- mapping from edges/arcs (n, n') to propositional variables $a_{n,n'}$
- property satisfied by subgraph consisting of true edges/arcs

In general, the property is identified with a single propositional variable that may be assigned true or false, but in this work the property is fixed to *true*.

This Work

- We will focus on acyclicity: how to handle SAT+acyclicity efficiently?
- space consumption linear in $|E| + |V|$
- strong propagations
- Outperforms other representations of acyclicity in our experiments.

Approach

1. Run a SAT solver based on Conflict-Driven Clause Learning (CDCL).
2. When an arc variable is assigned *true*,
   - Check whether the graph contains a cycle $n_1 \to n_2 \to \cdots \to n_m \to n_1$. If it does,
     1. generate a clause $\neg a_{n_1,n_2} \vee \neg a_{n_2,n_3} \vee \cdots \vee \neg a_{n_m,n_1}$,
     2. continue as with any false clause (learn an asserting clause, ...).
   - Check if there is an almost-cycle $n_1 \to n_2 \to \cdots \to n_m \to n_1$ with all arcs *true* but one $a_{n_j,n_{j+1}}$. If so,
     1. generate a clause $c = \neg a_{n_1,n_2} \vee \neg a_{n_2,n_3} \vee \cdots \cdots \neg a_{n_m,n_1}$,
     2. add it to the clause database,
     3. add $\neg a_{n_j,n_{j+1}}$ to the propagation queue with c as its reason clause, and
     4. continue propagation.

     Notice that there may be multiple such almost-cycles, each yielding a different literal.

Approach

1. Run a SAT solver based on Conflict-Driven Clause Learning (CDCL).
2. When an arc variable is assigned *true*,
   - Check whether the graph contains a cycle $n_1 \to n_2 \to \cdots \to n_m \to n_1$. If it does,
     1. generate a clause $\neg a_{n_1,n_2} \vee \neg a_{n_2,n_3} \vee \cdots \vee \neg a_{n_m,n_1}$,
     2. continue as with any false clause (learn an asserting clause, ...).
   - Check if there is an almost-cycle $n_1 \to n_2 \to \cdots \to n_m \to n_1$ with all arcs *true* but one $a_{n_j,n_{j+1}}$. If so,
     1. generate a clause $c = \neg a_{n_1,n_2} \vee \neg a_{n_2,n_3} \vee \cdots \cdots \neg a_{n_m,n_1}$,
     2. add it to the clause database,
     3. add $\neg a_{n_j,n_{j+1}}$ to the propagation queue with c as its reason clause, and
     4. continue propagation.

     Notice that there may be multiple such almost-cycles, each yielding a different literal.

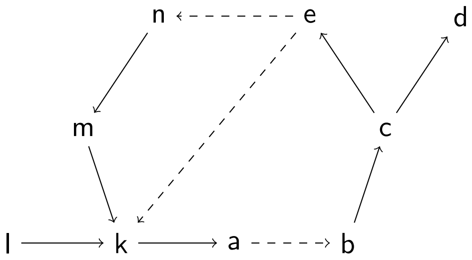
Approach

1. Run a SAT solver based on Conflict-Driven Clause Learning (CDCL).
2. When an arc variable is assigned *true*,
   - Check whether the graph contains a cycle $n_1 \to n_2 \to \cdots \to n_m \to n_1$. If it does,
     1. generate a clause $\neg a_{n_1,n_2} \vee \neg a_{n_2,n_3} \vee \cdots \vee \neg a_{n_m,n_1}$,
     2. continue as with any false clause (learn an asserting clause, ...).
   - Check if there is an almost-cycle $n_1 \to n_2 \to \cdots \to n_m \to n_1$ with all arcs *true* but one $a_{n_j,n_{j+1}}$. If so,
     1. generate a clause $c = \neg a_{n_1,n_2} \vee \neg a_{n_2,n_3} \vee \cdots\cdots \neg a_{n_m,n_1}$,
     2. add it to the clause database,
     3. add $\neg a_{n_j,n_{j+1}}$ to the propagation queue with c as its reason clause, and
     4. continue propagation.

   Notice that there may be multiple such almost-cycles, each yielding a different literal.

The Propagator
Example

Variable corresponding to $a \rightarrow b$ is made *true*.
Perform search forward from b (*true* arcs only).
Perform search backward from a (*true* arcs only).
Infer negations of arcs from brown to purple.
...deleting them.



The Propagator
Example

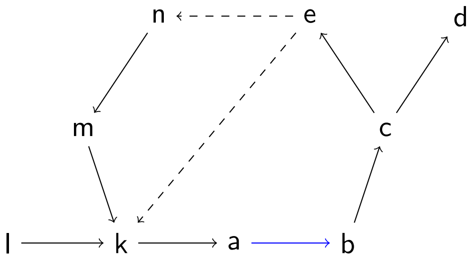
Variable corresponding to $a \to b$ is made *true*.

Perform search forward from b (*true* arcs only).

Perform search backward from a (*true* arcs only).

Infer negations of arcs from brown to purple.

...deleting them.



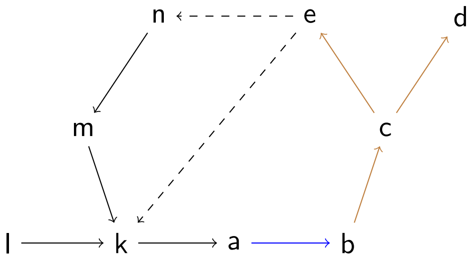
Variable corresponding to $a \to b$ is made *true*.

Perform search forward from b (*true* arcs only).

Perform search backward from a (*true* arcs only).

Infer negations of arcs from brown to purple.

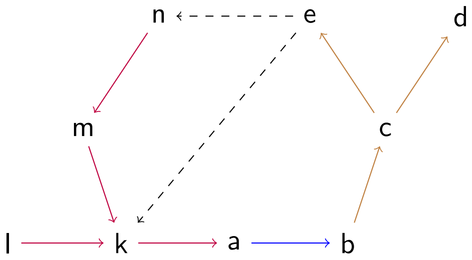
...deleting them.



The Propagator

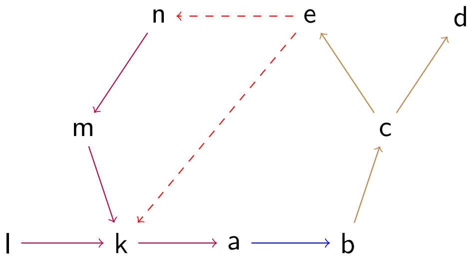
Example

Variable corresponding to $a \rightarrow b$ is made *true*.
Perform search forward from b (*true* arcs only).
Perform search backward from a (*true* arcs only).
Infer negations of arcs from brown to purple.
...deleting them.



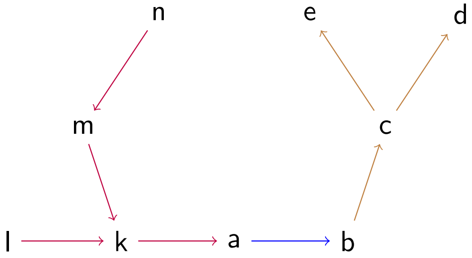
The Propagator
Example

Variable corresponding to $a \rightarrow b$ is made *true*.
Perform search forward from b (*true* arcs only).
Perform search backward from a (*true* arcs only).
Infer negations of arcs from brown to purple.
...deleting them.



The Propagator
Example

Variable corresponding to $a \rightarrow b$ is made *true*.
Perform search forward from b (*true* arcs only).
Perform search backward from a (*true* arcs only).
Infer negations of arcs from brown to purple.
...deleting them.



Implementation

- embedding in MiniSAT and Glucose SAT solvers
- a couple of dozens of lines of C++
- Propagator run after every decision in CDCL.
- Runtime overhead typically ≤ 5 per cent, even for largish graphs (10000+ nodes).

Integration with CDCL much simpler than with typical SMT theories such as linear arithmetics.

Acyclicity as CNF Constraints

Enumerative encoding

For every $n_1 \to \cdots \to n_m \to n_1$ have clause $\neg a_{n_1,n_2} \vee \cdots \vee \neg a_{n_m,n_1}$.
Size: $\mathcal{O}(v^v)$

Transitive closure

$a_{x,y} \to t_{x,y}$ $\quad$ $a_{x,y} \wedge t_{y,z} \to t_{x,z}$ $\quad$ $a_{x,y} \to \neg t_{y,x}$
Size: $\mathcal{O}(ev)$

Acyclicity as CNF Constraints

Tree reduction

Assign each node inductively the maximum distance of the any of its children from a leaf. If all distances are finite, there is no cycle.
Size: $\mathcal{O}(ev)$

Topological sort

Every node $n \in N$ implies a binary number $i(n)$.
For every arc $(n, n') \in A$ have $a_{n,n'} \to (i(n) < i(n'))$.
Size: $\mathcal{O}(v \log v + e \log v)$

Acyclicity as CNF Constraints

Properties: detection of cycles, inferring forbidden arcs

INC	Is inconsistency (a cycle) detected with UP (Unit Propagation) after all arcs forming a cycle are enabled?
BACK	For an enabled path $n_1, \ldots, n_k$, is arc (n_k, n_1) disabled by UP?

encoding	size	propagation
Enumerative	$\mathcal{O}(v^v)$	INC, BACK
Transitive Closure	$\mathcal{O}(ev)$	INC, BACK
Tree Reduction	$\mathcal{O}(ev)$	INC
Topological Sort	$\mathcal{O}(v \log v + e \log v)$	-

See also our paper in KR'14.

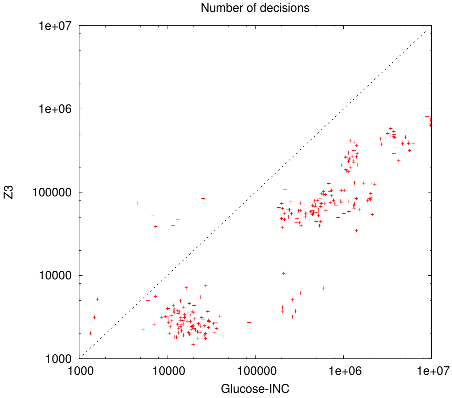
Linear inequalities & Difference Logic

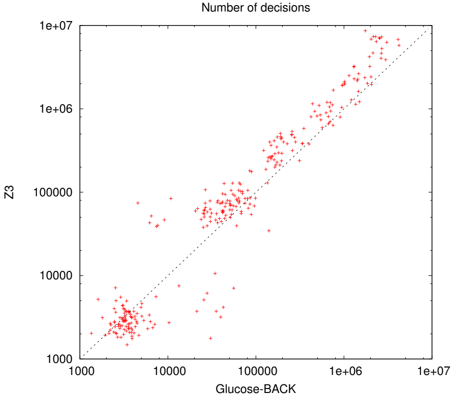
- Acyclicity can be easily encoded as the $<$ relation of integers/reals/rationals in SMT with linear real arithmetics and inequalities.
  1. Numeric variable n for every node n.
  2. For each arc variable $a_{n,n'}$ we have formula $a_{n,n'} \rightarrow (n < n')$.
- Some SMT solvers solve sets of simple inequalities like the above by graph-based algorithms, potentially detecting acyclicity efficiently and inferring forbidden arcs.
- However, in practice they are clearly outperformed by our SAT+acyclicity solver.

Runtime Comparison

Problem	Size	Glucose-INC	Glucose-BACK	MiniSAT-INC	MiniSAT-BACK	Glucose-SAT	MiniSAT-SAT	Lingeling-SAT	Clasp-SAT	Clasp-ASP	Z3-SMT
Hamilton	100	0.21	0.07	**0.03**	0.04	224.14	275.00	2419.63	2600.90	0.95	2.45
	150	0.13	0.15	**0.10**	0.12	3440.00	3172.54	3536.02	—	20.16	50.64
Acyclic	25	0.08	0.05	0.05	**0.03**	2406.60	2934.30	1.61	1282.49	0.12	0.29
	50	2.34	**0.28**	1.64	0.29	3147.91	2988.30	17.09	—	0.76	7.61
	75	682.86	8.09	856.47	**4.76**	3241.00	3276.92	99.60	—	282.01	167.74
	100	2180.98	964.28	2172.01	**647.13**	3170.48	3176.70	2760.52	1984.10	831.33	2278.63
Forest	25	**0.59**	0.64	0.75	0.72	118.70	139.88	3.10	3.59	4.09	4.54
	50	**301.46**	304.44	466.56	498.00	1165.53	1438.49	667.24	1125.86	1039.26	1205.63
	75	**909.15**	1006.73	1011.05	920.43	2597.99	2708.27	1019.68	1470.12	1501.76	1755.28
	100	1349.29	1418.25	1271.86	**1269.47**	2882.20	2853.03	2131.73	2597.71	1632.94	2690.67
Tree	25	0.80	0.74	**0.67**	0.83	72.93	6.12	3.17	4.12	4.37	4.75
	50	**301.81**	315.83	564.05	544.43	815.09	1230.09	685.38	1126.76	1193.09	1208.36
	75	**947.61**	999.07	976.40	1025.02	2646.64	2749.26	1044.51	1633.95	1495.32	1726.56
	100	1348.91	1414.68	1330.81	**1224.28**	2882.36	2861.33	2239.12	2621.82	1995.19	2538.20

Explaining Performance of SMT / Difference Logic Solvers





Other Graph Properties

- Graph properties important in many applications:
  - s-t-reachability: node t reachable from s (directed, undirected)
  - simple paths: a node is on a simple path between s and t
  - cycles: acyclicity, cyclicity
  - chordality: graph consists of triangles
- Devising efficient (linear-time) propagators often a challenge. ($\leadsto$ Explains why compact and efficient CNF-encodings hard to come by.)

Conclusion

- Proposed a framework SAT modulo Graphs.
- Presented efficient and simple implementation of SAT + Acyclicity.

- Future work: implementation MAXSAT modulo Graphs
- Future work: other graph properties
- Future work: applications SAT + Graphs
- Remaining performance differences to ASP solvers such as Clasp?