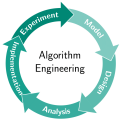
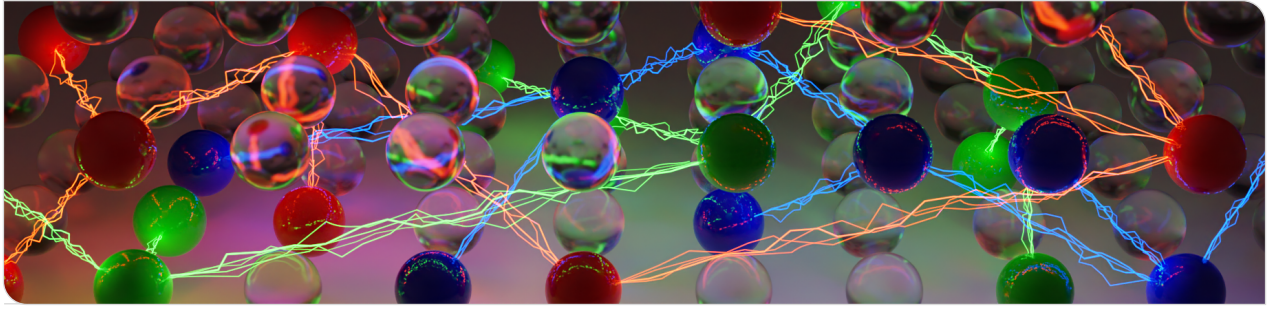
Karlsruhe Institute of Technology



Practical SAT Solving

Lecture 11: Application Highlights

Markus Iser, Dominik Schreiber, Tomáš Balyo | July 1, 2024



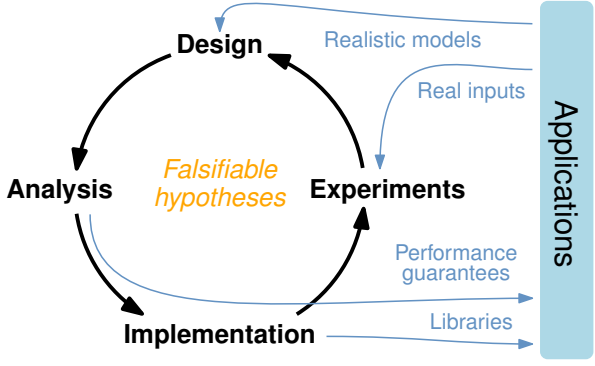
Why consider the Application View? (1/2)

Result, instance family		1st author	Speedups of MallobSat over Kissat-MAB_HyWalk
SAT	Hypertree decomposition	Schidler	3, 5, 5, 5, 5, 7, 8, 9, 12, 13
SAT	Hamilton circle	Heule	4, 4, 7, 11, 17, 20, 21, 22, 24, 31, 33, 36, 42
SAT	Tree decomposition	Ehlers	5, 7, 87
UNSAT	Cellular automatons	Chowdhury	5, 8, 8, 9, 9, 10, 22, 22, 66
			⋮
UNSAT	Relativized pidgeon hole	Elffers	277, 542, 638
UNSAT	Bioinformatics	Bonet	292, 717
UNSAT	Balanced random	Spence	321, 388
SAT	Sum of three cubes	Riveros	384, 509, 1018, 3345
SAT	Circuit multiplication	Shunyang	17, 31, 32, 62, 105, 119, 213, 254, 393, 741, 746, 1401, 2650
UNSAT	Perfect matchings	Reeves	119, 413, 12 439, 108 593, 217 099

3072 cores (128 machines) of SuperMUC-NG · 400 problems from Int. SAT Competition 2021
Only instances with seq. time $\geq$ 60 s · Only families with $\geq$ 2 instances

Why consider the Application View? (2/2)

- Perform sound **algorithm engineering** with realistic applications in the loop
- Understand application-specific solver behavior, needs, shortcomings
- Advance the positive feedback loop between solvers and applications



SAT Competition Benchmarks – “Real” Applications?

5355 problems,
138 families + 72 unknown

Font size $\propto$ # problems

via wordclouds.com,
GBD (benchmark-database.de)

Overview

Selected (!) Application Highlights of SAT

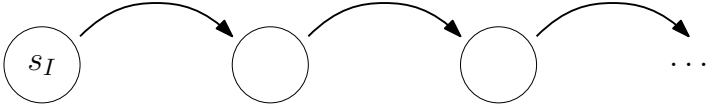
- Recap: Planning
- Bounded Model Checking
- Combinational Equivalence Checking
- Analyzing Cryptographic Building Blocks
- Multi Agent Path Finding
- Explainable AI: Learning decision trees
- Train scheduling with disruptions via MaxSAT

Remark

This slide set contains many literature references—please follow them according to your interest!

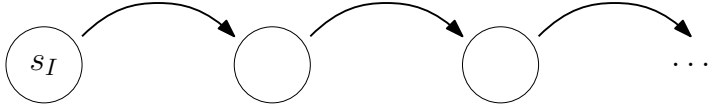
Recap: Planning

- **World state** s: Set of Boolean features
  - Assert initial state s_I via unit clauses for time step 0
  - Assert goal state features g via unit clauses for time step k
- **Action**: Template for valid state transitions $s_x \rightsquigarrow s_{x+1}$
  - Executing an action at step k implies its preconditions at step k
  - Executing an action at step k implies its effects at step $k+1$
- **Plan**: Valid sequence of actions leading from s_I to a goal state
  - Sequence of action variables set to true in satisfying assignment
- Anything else to encode?



Recap: Planning

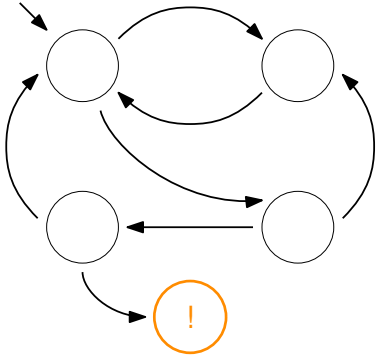
- **World state** s: Set of Boolean features
  - Assert initial state s_I via unit clauses for time step 0
  - Assert goal state features g via unit clauses for time step k
- **Action**: Template for valid state transitions $s_x \rightsquigarrow s_{x+1}$
  - Executing an action at step k implies its preconditions at step k
  - Executing an action at step k implies its effects at step $k+1$
- **Plan**: Valid sequence of actions leading from s_I to a goal state
  - Sequence of action variables set to true in satisfying assignment
- Anything else to encode? Frame axioms – don't let the solver hallucinate causeless state changes!



From Planning to Verification

Let's use our SAT-based planning to check the correctness of a system!

- World state features $\equiv$ State features of our system
- Actions $\equiv$ Valid transitions between states
- Goals $\equiv$

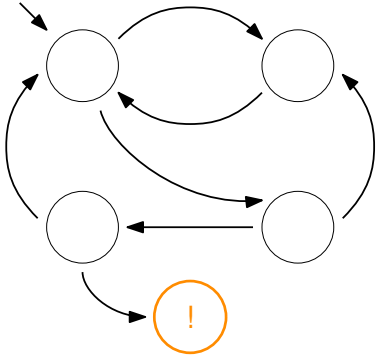


A snack machine or an electronic component or a C program or ...

From Planning to Verification

Let's use our SAT-based planning to check the correctness of a system!

- World state features $\equiv$ State features of our system
- Actions $\equiv$ Valid transitions between states
- Goals $\equiv$ incorrect state, violating some constraint
- Plan $\equiv$

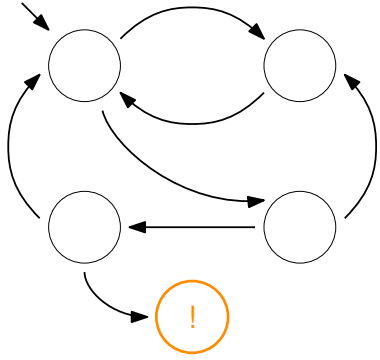


A snack machine or an electronic component or a C program or ...

From Planning to Verification

Let's use our SAT-based planning to check the correctness of a system!

- World state features $\equiv$ State features of our system
- Actions $\equiv$ Valid transitions between states
- Goals $\equiv$ incorrect state, violating some constraint
- Plan $\equiv$ reachable incorrectness
- Unsatisfiability $\equiv$ system is always correct?

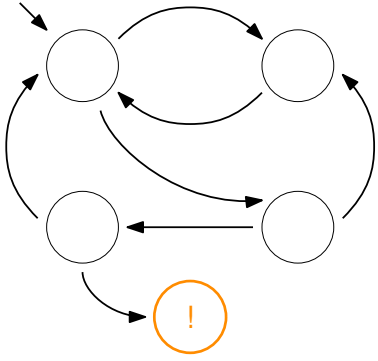


A snack machine or an electronic component or a C program or ...

From Planning to Verification

Let's use our SAT-based planning to check the correctness of a system!

- World state features $\equiv$ State features of our system
- Actions $\equiv$ Valid transitions between states
- Goals $\equiv$ incorrect state, violating some constraint
- Plan $\equiv$ reachable incorrectness
- Unsatisfiability at k steps $\equiv$ system is correct within k steps



A snack machine or an electronic component or a C program or ...

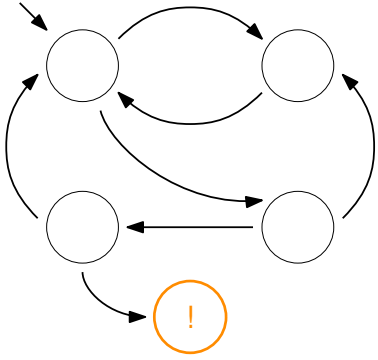
From Planning to Verification

Let's use our SAT-based planning to check the correctness of a system!

- World state features $\equiv$ State features of our system
- Actions $\equiv$ Valid transitions between states
- Goals $\equiv$ incorrect state, violating some constraint
- Plan $\equiv$ reachable incorrectness
- Unsatisfiability at k steps $\equiv$ system is correct within k steps

Bounded Model Checking

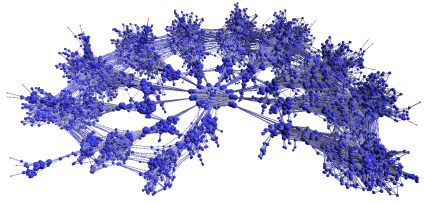
- Encode and check transition system for $k = 1, 2, \ldots$
- Satisfying assignment $\equiv$ counter example!
- Crucial tool for hardware and software verification [1]



A snack machine or an electronic component or a C program or ...

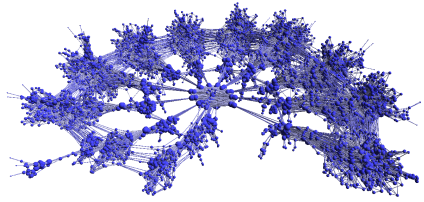
Bounded Model Checking

- Invented by Clarke & Biere in ∼2000 [2], mostly replacing BDD-based model checking
- State transition system based on temporal logic (LTL, CTL, . . .); solving via SAT or SMT
- Applications: Computer-aided design (CAD), software verification, invariant checking, bug detection, . . . [1]



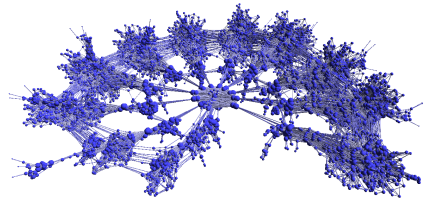
Bounded Model Checking

- Invented by Clarke & Biere in ∼2000 [2], mostly replacing BDD-based model checking
- State transition system based on temporal logic (LTL, CTL, . . .); solving via SAT or SMT
- Applications: Computer-aided design (CAD), software verification, invariant checking, bug detection, . . . [1]
- One of the most essential real-world applications of SAT
  - Pushed industrial interest in SAT solvers in 2000s
  - Actively influenced solver design and algorithms
  - Some of the largest, structurally most distinct benchmarks



Bounded Model Checking

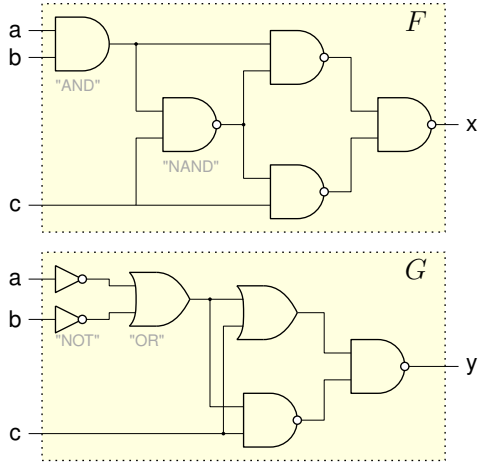
- Invented by Clarke & Biere in ∼2000 [2], mostly replacing BDD-based model checking
- State transition system based on temporal logic (LTL, CTL, . . .); solving via SAT or SMT
- Applications: Computer-aided design (CAD), software verification, invariant checking, bug detection, . . . [1]
- One of the most essential real-world applications of SAT
  - Pushed industrial interest in SAT solvers in 2000s
  - Actively influenced solver design and algorithms
  - Some of the largest, structurally most distinct benchmarks
- Examples for BMC @ KIT:
  - Low-Level Bounded Model Checker (LLBMC) [3] (C program verification)
  - Verification of Java contracts [4] (see right)
  - Cryptography [5]



```
/*@ requires 0 <= x1;
  @ ensures \result == x1 * x2;
  @ assignable \nothing;
  @*/
public int mult(int x1, int x2) {
  int res = 0;
  /*@ loop_invariant 0 <= i && i <= x1 && res == i * x2;
    @ decreases x1 - i;
    @ assignable \nothing;
    @*/
  for (int i = 0; i < x1; ++i) res += x2;
  return res;
}
```

Combinational Equivalence Checking

- Given: two combinational circuits
  stateless "input→output" circuit, no feedback
- Question: are the circuits logically equivalent?
- Right example: Is $F(a, b, c) \equiv G(a, b, c)$?
- How to solve with SAT?

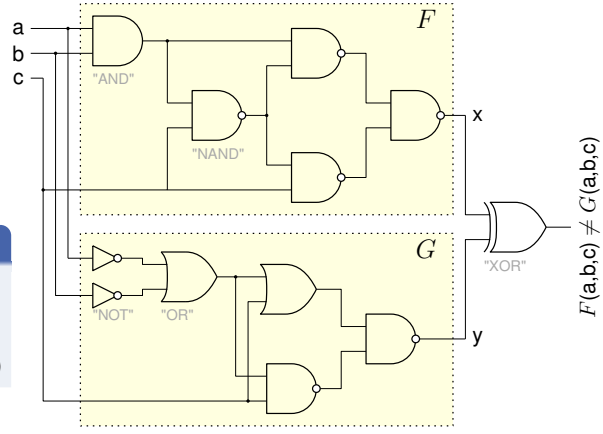


Combinational Equivalence Checking

- Given: two combinational circuits
  stateless "input→output" circuit, no feedback
- Question: are the circuits logically equivalent?
- Right example: Is $F(a, b, c) \equiv G(a, b, c)$?
- How to solve with SAT?

Miter Formula

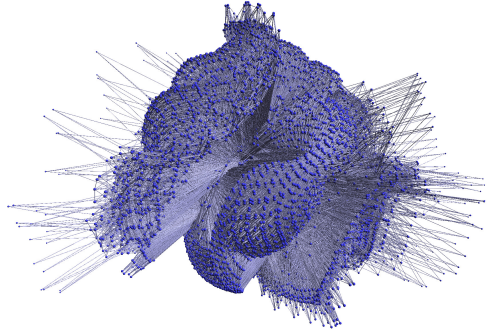
- Encode F, G relative to shared input bits
- Assert $x \neq y$ (multi-bit output: $\bigvee_{x_i, y_i} x_i \neq y_i$)
- Satisfiable $\Leftrightarrow$ ($F \not\equiv G$) (why this way?)



CEC: How Hard Can It Be?

Two Miter examples from SAT Competition 2023

- Instance A: 260k variables, 850k clauses
  - Circuits are not equivalent
  - Solved in 1.33 s by KISSAT_MAB_PROP-NO_SYM [6]

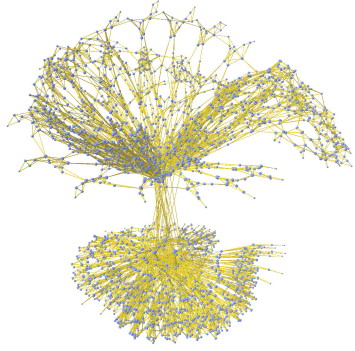


Nodes = variables;
Edges = common clause(s);
variables contracted by factor ≈ 16

CEC: How Hard Can It Be?

Two Miter examples from SAT Competition 2023

- Instance A: 260k variables, 850k clauses
  - Circuits are not equivalent
  - Solved in 1.33 s by KISSAT_MAB_PROP-NO_SYM [6]
- Instance B: 4k variables, 13k clauses
  - Circuits are equivalent
  - Unsolved by sequential solvers within 5000 s
  - Solved by some parallel solvers :)



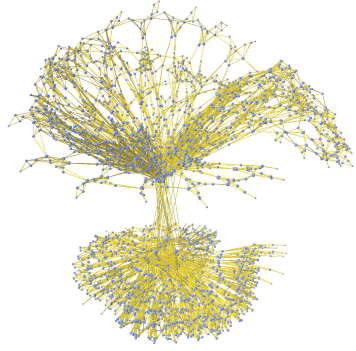
Nodes = variables;
Edges = common clause(s)

CEC: How Hard Can It Be?

Two Miter examples from SAT Competition 2023

- Instance A: 260k variables, 850k clauses
  - Circuits are not equivalent
  - Solved in 1.33 s by KISSAT_MAB_PROP-NO_SYM [6]
- Instance B: 4k variables, 13k clauses
  - Circuits are equivalent
  - Unsolved by sequential solvers within 5000 s
  - Solved by some parallel solvers :)

Generally: co-NP-complete, can require very large proofs

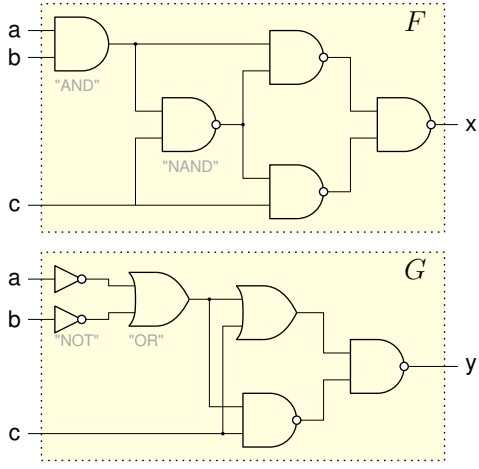


Nodes = variables;
Edges = common clause(s)

CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

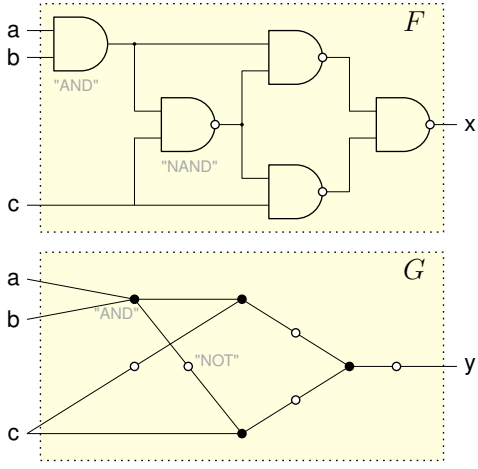
- Randomly test different inputs, collecting pairs of potentially equivalent nodes



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

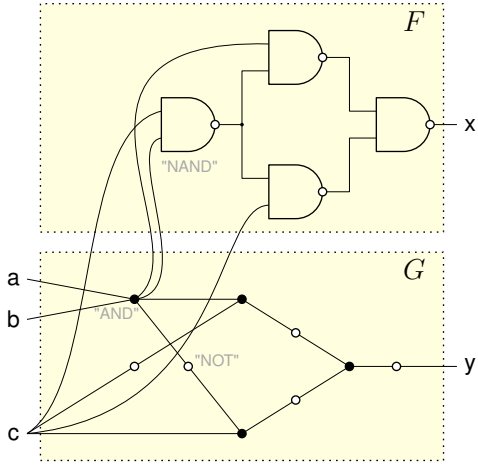
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

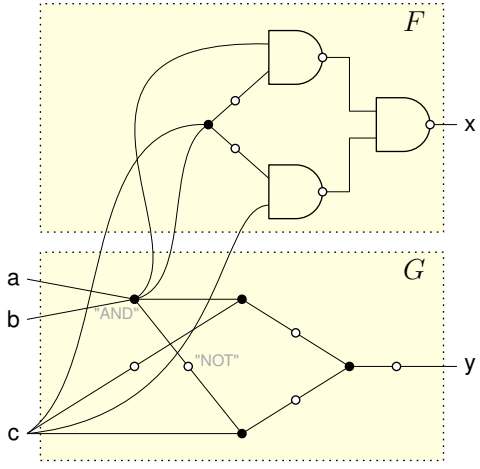
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

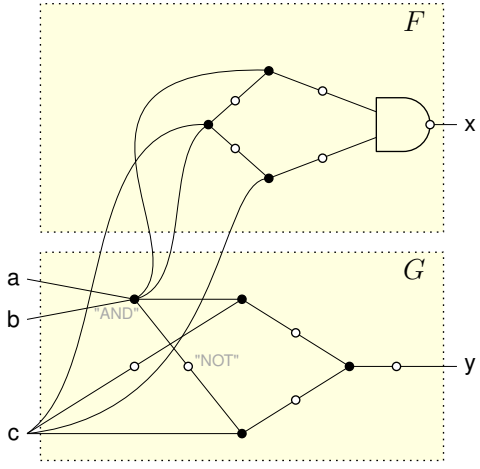
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

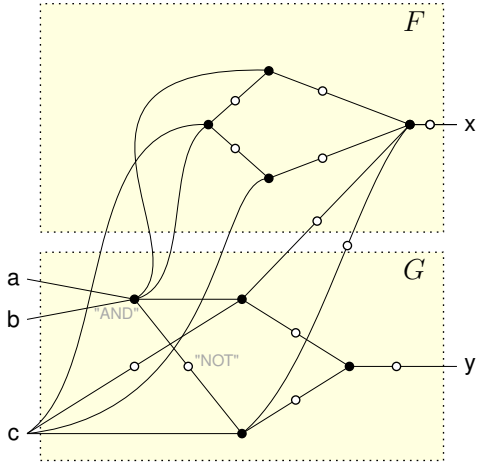
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

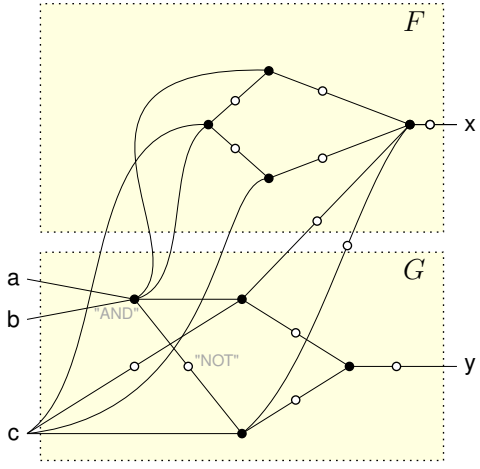
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

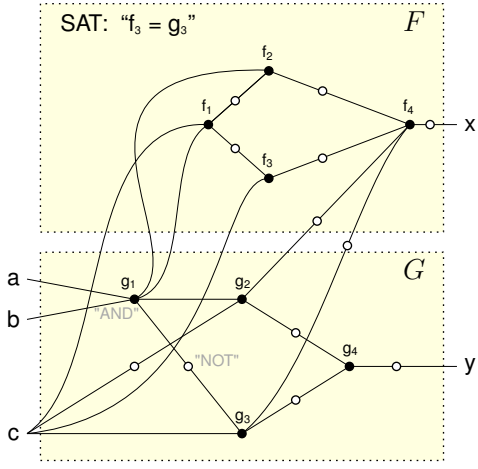
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging
- Structural hashing: Ensure that each functionally distinct sub-circuit is encoded only once



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

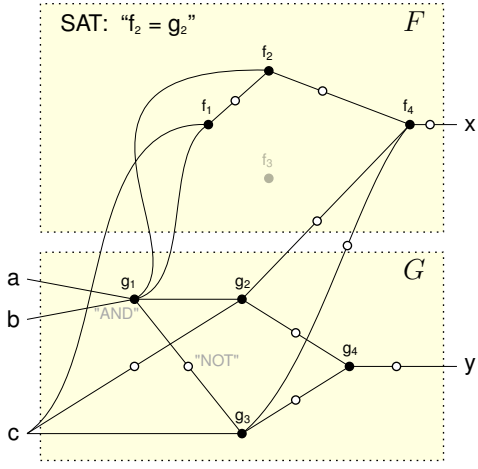
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging
- Structural hashing: Ensure that each functionally distinct sub-circuit is encoded only once
- SAT sweeping: Use SAT sub-program to test whether potentially equivalent nodes are actually equivalent



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

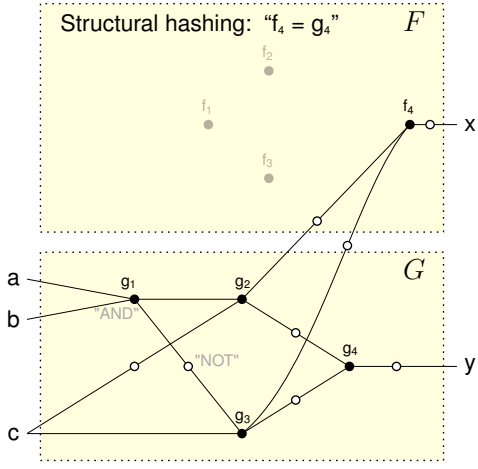
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging
- Structural hashing: Ensure that each functionally distinct sub-circuit is encoded only once
- SAT sweeping: Use SAT sub-program to test whether potentially equivalent nodes are actually equivalent



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

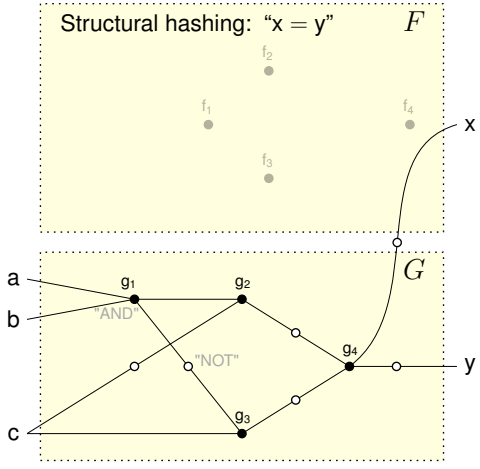
- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging
- Structural hashing: Ensure that each functionally distinct sub-circuit is encoded only once
- SAT sweeping: Use SAT sub-program to test whether potentially equivalent nodes are actually equivalent



CEC Techniques [7]

Improving CEC performance: Try to merge equivalent sub-circuits

- Randomly test different inputs, collecting pairs of potentially equivalent nodes
- Use And/Inverter-Graph (AIG) for simple circuit manipulation and merging
- Structural hashing: Ensure that each functionally distinct sub-circuit is encoded only once
- SAT sweeping: Use SAT sub-program to test whether potentially equivalent nodes are actually equivalent



CEC: Remarks

- Important cornerstone of **Electronic Design Automation** [8]
  – can be used to validate implementation based on specification
  – other EDA techniques: model checking, Automated Test Pattern Generation (ATPG)
- Crucial "intrinsically Boolean" benchmark problem throughout history of SAT solving
  – Every SAT competition features miters
- SAT sweeping originally proposed for bounded model checking [9]
- Gate recognition and merging now a form of general inprocessing for any formula, connected to variable elimination [10]

Analyzing Cryptographic Building Blocks

Cryptanalysis = analyze, attempt to “break” cryptographic building blocks to test, advance them

- Building blocks: Stream ciphers ((msg,key) $\mapsto$ encrypted msg) [11], hash functions [12], ...
- Algebraic cryptanalysis: try to build equations relating output to input [11]
  - SAT solver should support XOR clauses
  - SAT solver should use Gaussian Elimination as a sub-program

Analyzing Cryptographic Building Blocks

Cryptanalysis = analyze, attempt to “break” cryptographic building blocks to test, advance them

- Building blocks: Stream ciphers ((msg,key) $\mapsto$ encrypted msg) [11], hash functions [12], . . .
- Algebraic cryptanalysis: try to build equations relating output to input [11]
  - SAT solver should support XOR clauses
  - SAT solver should use Gaussian Elimination as a sub-program
- Established SAT-based approaches:
  - Prove mathematical properties of internal states [12]
  - Find weak keys and preimages [13]
  - Find collisions of hash functions [14]

Analyzing Cryptographic Building Blocks

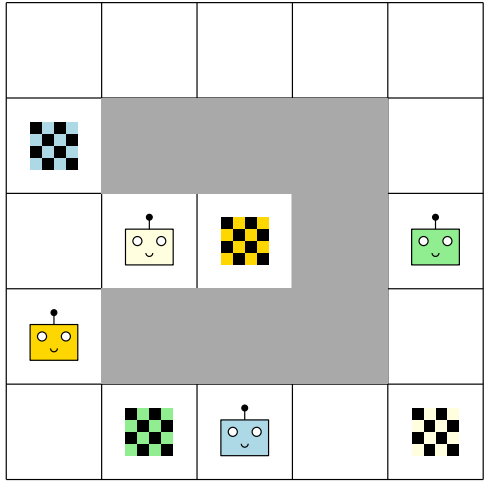
Cryptanalysis = analyze, attempt to "break" cryptographic building blocks to test, advance them

- Building blocks: Stream ciphers ((msg,key) $\mapsto$ encrypted msg) [11], hash functions [12], ...
- Algebraic cryptanalysis: try to build equations relating output to input [11]
  - SAT solver should support XOR clauses
  - SAT solver should use Gaussian Elimination as a sub-program
- Established SAT-based approaches:
  - Prove mathematical properties of internal states [12]
  - Find weak keys and preimages [13]
  - Find collisions of hash functions [14]
- Cross-application use of SAT techniques:
  - Cryptanalysis via SMT solving [15]
  - Cryptanalysis via bounded model checking [16]
  - Hash function analysis also used in algorithm design [17]

"*it turns out that the highly combinatorial nature of the problem is not well suited for linear solvers, and that SAT solvers are a better fit for this type of problem*" —Dobraunig et al. after trying MILP for ASCON [12]

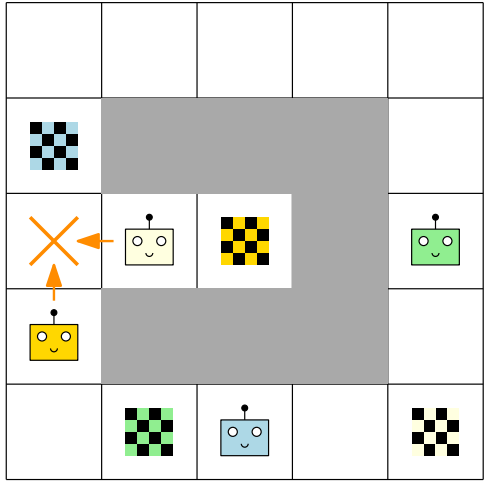
Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position



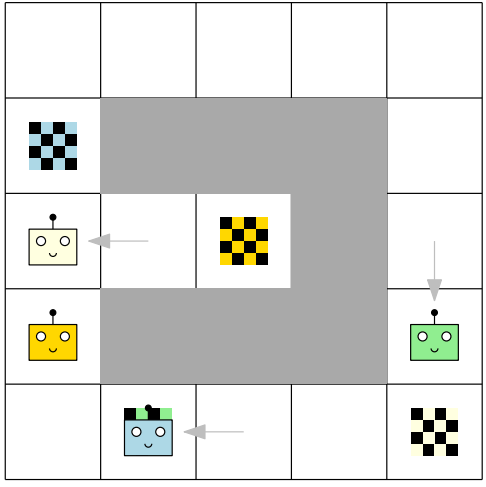
Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position
- Collisions disallowed
- Optimize makespan (= steps until all goals reached) or Sum of Costs (= total number of actions performed)



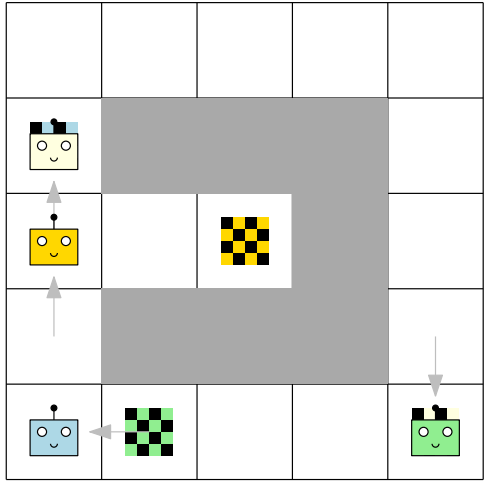
Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position
- Collisions disallowed
- Optimize makespan (= steps until all goals reached) or Sum of Costs (= total number of actions performed)



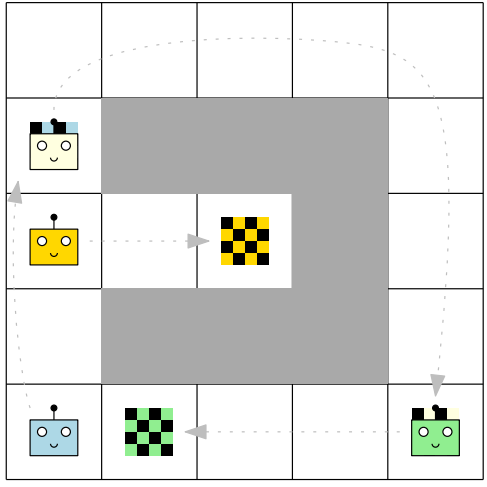
Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position
- Collisions disallowed
- Optimize makespan (= steps until all goals reached) or Sum of Costs (= total number of actions performed)



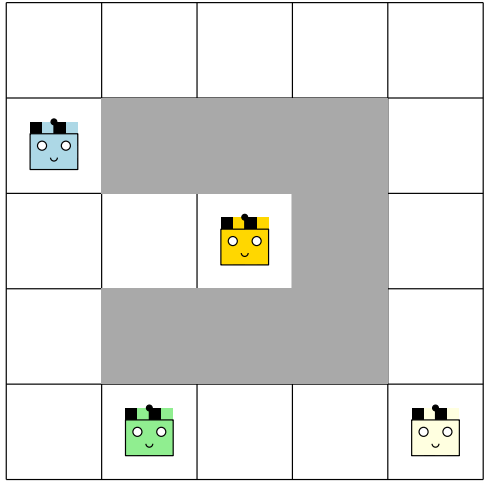
Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position
- Collisions disallowed
- Optimize makespan (= steps until all goals reached) or Sum of Costs (= total number of actions performed)



Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position
- Collisions disallowed
- Optimize makespan (= steps until all goals reached) or Sum of Costs (= total number of actions performed)

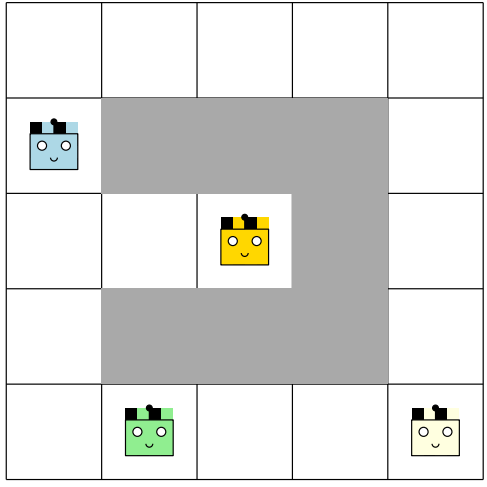


Multi Agent Path Finding [18]

- Discretized 2D grid of positions, n cooperative agents
- Discretized time steps: move 0-1 cells per time step
- Per agent: Initial position and goal position
- Collisions disallowed
- Optimize makespan (= steps until all goals reached) or Sum of Costs (= total number of actions performed)

Optimal Approaches to MAPF

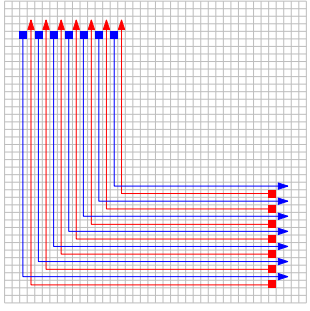
- M* algorithm: adjusted A^* with collision handling and backtracking
- Conflict Based Search (CBS): route individually; at collision, add constraint to a colliding agent and re-route
- Reduction-based approaches: SAT, MaxSAT, ASP, CSP



Is SAT-based MAPF worthwhile?

Observation on MAPF [18] (and planning, and scheduling, and probably many other problems . . .):

Direct search-based approaches
perform especially well on large, lightly constrained instances.



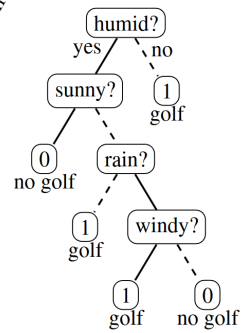
SAT-based approaches
perform especially well on small-sized, highly constrained instances.

13	7	2	4
3	8	15	5
9	12	1	6
14	11	10	

Explainable AI: Learning Decision Trees

- Given: n d-dimensional sample vectors (d features) mapped to a (binary) class
- Task: Learn decision tree classifying all samples
- Explainable classifier (the more shallow the better)

sample	sunny	rain	overcast	temp:mild	temp:hot	temp:cool	humid	windy	play golf
e_1	0	1	0	0	0	1	0	1	1
e_2	0	1	0	1	0	0	0	1	1
e_3	1	0	0	0	1	0	1	1	0
e_4	0	0	1	0	0	1	0	0	1
e_5	1	0	0	1	0	0	0	0	1
e_6	0	1	0	1	0	0	1	0	0
e_7	0	1	0	1	0	0	1	1	1
e_8	0	0	1	0	1	0	1	1	1
e_9	1	0	0	1	0	0	1	1	0
e_{10}	1	0	0	0	0	1	0	1	1
e_{11}	0	0	1	1	0	0	1	0	1
e_{12}	1	0	0	0	1	0	1	0	0



taken from [19]

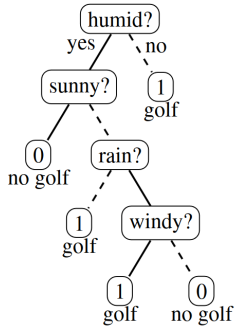
Explainable AI: Learning Decision Trees

- Given: n d-dimensional sample vectors (d features) mapped to a (binary) class
- Task: Learn decision tree classifying all samples
- Explainable classifier (the more shallow the better)

SAT-based approach [20]

- Encode complete binary tree of depth k
- Encode recursively for each node which samples are excluded along its path
- Constrain that 0-leaves exclude all 1-labeled samples and vice versa
- Solver picks a sub-tree and each node's feature

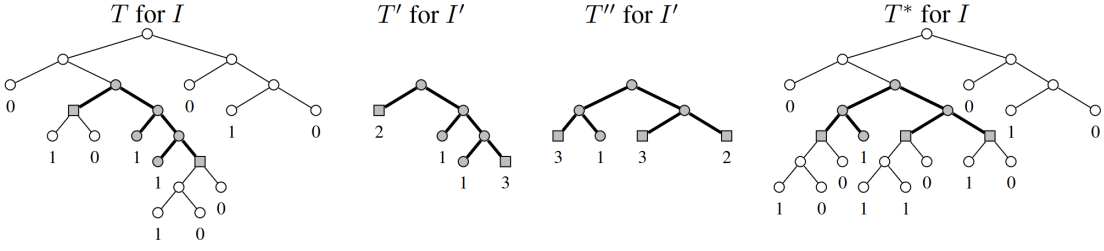
sample	sunny	rain	overcast	temp:mild	temp:hot	temp:cool	humid	windy	play golf
e_1	0	1	0	0	0	1	0	1	1
e_2	0	1	0	1	0	0	0	1	1
e_3	1	0	0	0	1	0	1	1	0
e_4	0	0	1	0	0	1	0	0	1
e_5	1	0	0	1	0	0	0	0	1
e_6	0	1	0	1	0	0	1	0	0
e_7	0	1	0	1	0	0	1	1	1
e_8	0	0	1	0	1	0	1	1	1
e_9	1	0	0	1	0	0	1	1	0
e_{10}	1	0	0	0	0	1	0	1	1
e_{11}	0	0	1	1	0	0	1	0	1
e_{12}	1	0	0	0	1	0	1	0	0



taken from [19]

SAT-based Improvement of Decision Trees [19]

- SAT-based approach slow/infeasible for large data sets
- Better: Construct initial decision tree heuristically, then locally improve sub-trees via SAT
  - **Hybrid approach**, also beneficial in other contexts, e.g., CEC, planning [21]
- Enables to scale up merits of SAT to arbitrarily large data sets



taken from [19]

More on SAT Solving × Machine Learning

- Algorithm selection
  Xu, Lin, et al. “SATzilla: portfolio-based algorithm selection for SAT.” JAIR 2008.
  Eggensperger, Katharina, Marius Lindauer, and Frank Hutter. “Neural networks for predicting algorithm runtime distributions.” IJCAI 2018.
- SAT Solving featuring ML techniques
  Liang, Jia Hui, et al. “Learning rate based branching heuristic for SAT solvers.” SAT 2016.
  Guo, Wenxuan, et al. “Machine learning methods in solving the boolean satisfiability problem.” Machine Intelligence Research (2023).
- Verify Neural Networks via SAT/SMT solving
  Huang, Xiaowei, et al. “Safety verification of deep neural networks.” CAV 2017.
  Ehlers, Ruediger. “Formal verification of piece-wise linear feed-forward neural networks.” ATVA 2017.
- Analyze and understand behavior of SAT solvers and instances
  Soos, Mate, Raghav Kulkarni, and Kuldeep S. Meel. “CrystalBall: gazing in the black box of SAT solving.” SAT 2019.
  Fuchs, Tobias, Jakob Bach, and Markus Iser. “Active Learning for SAT Solver Benchmarking.” TACAS 2023.

Train Scheduling with Disruptions via MaxSAT [22]

Can SAT Solving fix the Deutsche Bahn?

Train Scheduling with Disruptions via MaxSAT [22]

Can SAT Solving fix the Deutsche Bahn?
No.

Train Scheduling with Disruptions via MaxSAT [22]

Can SAT Solving fix the Deutsche Bahn?

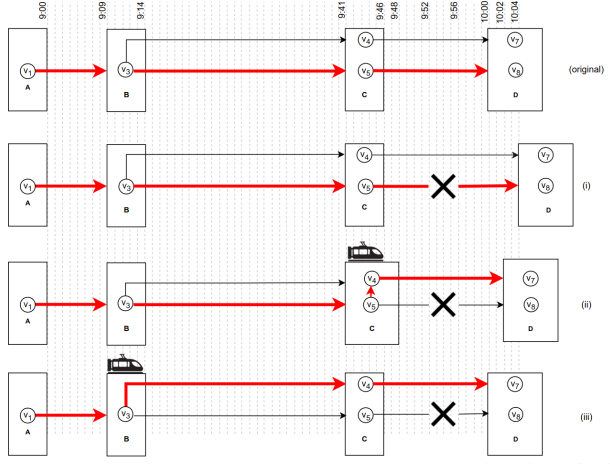
No.

But it might make the Swiss trains run even better!

Train Scheduling with Disruptions via MaxSAT [22]

Train Scheduling Optimization Problem (TSOP)

- Route trains through predefined stations
- Schedule train timetable subject to time and resource constraints



taken from [22]

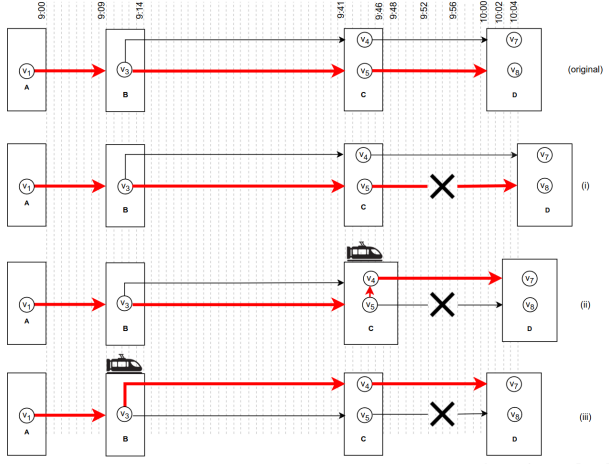
Train Scheduling with Disruptions via MaxSAT [22]

Train Scheduling Optimization Problem (TSOP)

- Route trains through predefined stations
- Schedule train timetable subject to time and resource constraints

TSOP Under Disruption (TSOPUD)

- Numerous disruptions: slowdown, train blocked, track blocked, staffing / rolling stock
- Reroute, reschedule to minimize delays



taken from [22]

Train Scheduling with Disruptions via MaxSAT [22]

Train Scheduling Optimization Problem (TSOP)

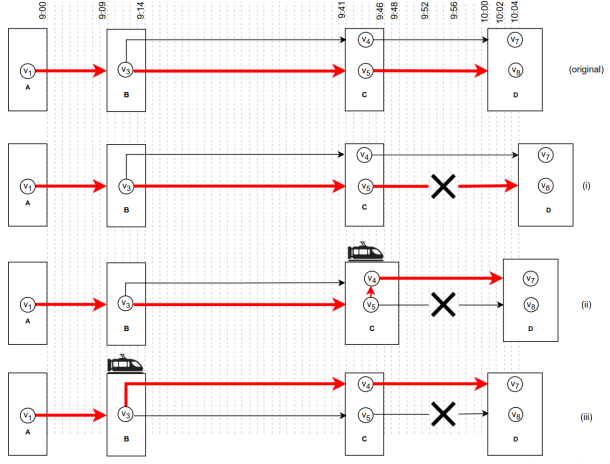
- Route trains through predefined stations
- Schedule train timetable subject to time and resource constraints

TSOP Under Disruption (TSOPUD)

- Numerous disruptions: slowdown, train blocked, track blocked, staffing / rolling stock
- Reroute, reschedule to minimize delays

MaxSAT-based approach

- Encode time requirements as hard clauses, route/train cost as soft clauses
- Relax timings incrementally until feasible
- Add disruption(s), relax timings as needed



taken from [22]

Application Highlights: Takeaways

- SAT is an (actually!) **essential and well-established tool** for
  - software & hardware verification
  - electronic design automation
  - specific cryptanalysis tasks.
- Indicators for a problem to best use SAT for?
  - large portion of “intrinsically Boolean” constraints
  - combinatorial search space / set of decisions, NP-hard (or harder)
  - problem description not too large
- Cross-application techniques and insights:
  - Often promising to hybridize SAT with direct (search) methods: use SAT to resolve difficult cores
  - Positive feedback loop between solver techniques and applications
  - Cross-fertilization between different applications (e.g., BMC, SMT)
- **Many more**, sometimes important, **applications** of SAT we didn’t talk about
  - Automated test pattern generation, graph theory, theorem proving, social choice theory, puzzle solving, knowledge compilation, combinatorial design theory, . . .

References I

[1] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. “Boolean Satisfiability Solvers and Their Applications in Model Checking”. In: *Proc. IEEE*. Vol. 103. 11. 2015, pp. 2021–2035. DOI: 10.1109/JPROC.2015.2455034.

[2] Edmund Clarke et al. “Bounded model checking using satisfiability solving”. In: *Formal methods in system design* 19 (2001), pp. 7–34.

[3] Stephan Falke, Florian Merz, and Carsten Sinz. “The bounded model checker LLBMC”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 706–709.

[4] Bernhard Beckert et al. “Modular verification of JML contracts using bounded model checking”. In: *Int. Symposium on Leveraging Applications of Formal Methods (ISoLA)*. Springer. 2020, pp. 60–80.

[5] Alexander Koch, Michael Schrempp, and Michael Kirsten. “Card-based cryptography meets formal verification”. In: *New Generation Computing* 39.1 (2021), pp. 115–158.

[6] Yu Gao. “Kissat_MAB_prop in SAT Competition 2023”. In: *SAT Competition*. 2023, p. 16.

[7] Sean Weaver. *Equivalence Checking*. https://www21.in.tum.de/~lammich/2015_SS_Seminar_SAT/resources/Equivalence_Checking_11_30_08.pdf. 2015.

[8] João P. Marques-Silva and Karem A. Sakallah. “Boolean satisfiability in electronic design automation”. In: *Proc. DAC*. 2000, pp. 675–680. DOI: 10.1145/337292.337611.

[9] Andreas Kuehlmann. “Dynamic transition relation simplification for bounded property checking”. In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004*. IEEE. 2004, pp. 50–57.

[10] Armin Biere and Mathias Fleury. “Gimsatul, IsaSAT, Kissat Entering the SAT Competition 2022”. In: *SAT Competition*. 2022, pp. 10–11.

References II

[11] Mate Soos, Karsten Nohl, and Claude Castelluccia. “Extending SAT Solvers to Cryptographic Problems”. In: *Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[12] Christoph Dobraunig et al. “Cryptanalysis of ascon”. In: *Topics in Cryptology—CT-RSA 2015: The Cryptographer’s Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*. Springer. 2015, pp. 371–387.

[13] Frédéric Lafitte, Jorge Nakahara Jr, and Dirk Van Heule. “Applications of SAT solvers in cryptanalysis: finding weak keys and preimages”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 9.1 (2014), pp. 1–25.

[14] Ilya Mironov and Lintao Zhang. “Applications of SAT solvers to cryptanalysis of hash functions”. In: *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*. Springer. 2006, pp. 102–115.

[15] Wenqian Xin et al. “Improved cryptanalysis on SipHash”. In: *International Conference on Cryptology and Network Security*. Springer. 2019, pp. 61–79.

[16] Norbert Manthey. “Testing the ASCON Hash Function”. In: *SAT Competition*. 2023, p. 63.

[17] Sean Weaver and Marijn J. H. Heule. “Constructing minimal perfect hash functions using SAT technology”. In: *AAAI Conference on Artificial Intelligence*. Vol. 34. 02. 2020, pp. 1668–1675. DOI: 10.1609/aaai.v34i02.5529.

[18] Pavel Surynek et al. “Migrating Techniques from Search-Based Multi-Agent Path Finding Solvers to SAT-Based Approach”. In: *JAIR* 73 (2022), pp. 553–618. DOI: 10.1613/jair.1.13318.

[19] André Schidler and Stefan Szeider. “SAT-based decision tree learning for large data sets”. In: *AAAI conference on artificial intelligence*. Vol. 35. 5. 2021, pp. 3904–3912.

References III

[20] Nina Narodytska et al. “Learning optimal decision trees with SAT”. In: *International Joint Conference on Artificial Intelligence 2018*. Association for the Advancement of Artificial Intelligence (AAAI). 2018, pp. 1362–1368.

[21] Nils Froleyks, Tomáš Balyo, and Dominik Schreiber. “PASAR—Planning as Satisfiability with Abstraction Refinement”. In: *Proc. SoCS*. Vol. 10. 1. 2019, pp. 70–78. URL: https://ojs.aaai.org/index.php/SOCS/article/download/18504/18295.

[22] Alexandre Lemos et al. “Iterative Train Scheduling under Disruption with Maximum Satisfiability”. In: *Journal of Artificial Intelligence Research* 79 (2024), pp. 1047–1090.