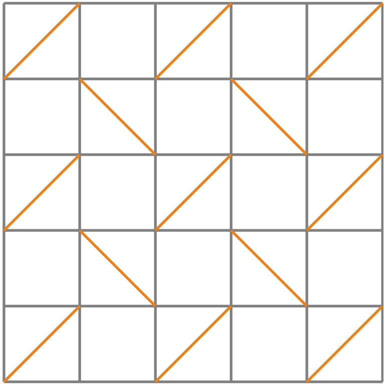
While waiting for the talk to begin, enjoy the puzzle: draw 16 diagonals that do not touch each other



Satisfiability Problem

Alexander Kulikov

October 13, 2022

JetBrains

Satisfiability (SAT)

- \$1M prize for proving that SAT can or cannot be solved efficiently

Satisfiability (SAT)

- $1M prize for proving that SAT can or cannot be solved efficiently
- Many practical applications: scheduling, planning, verification, etc. We will implement together a few simple programs using SAT-solvers

Satisfiability (SAT)

- $1M prize for proving that SAT can or cannot be solved efficiently
- Many practical applications: scheduling, planning, verification, etc. We will implement together a few simple programs using SAT-solvers
- Many theoretical connections: proof complexity, formal verification, fine-grained reductions

Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

Is it possible to select a representative from every clause such that x and $-x$ are not selected simultaneously?

Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

Is it possible to select a representative from every clause such that x and $\neg x$ are not selected simultaneously?

Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

Example

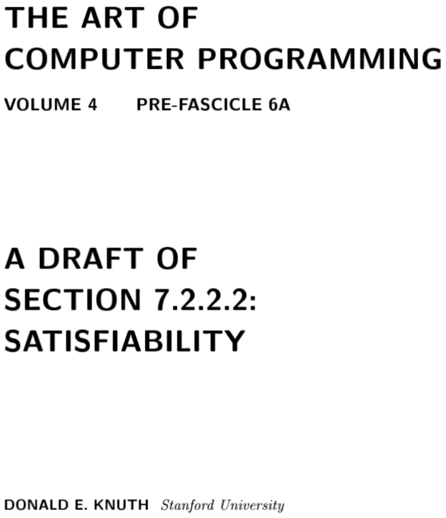
```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

Is it possible to select a representative from every clause such that x and $-x$ are not selected simultaneously?

Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

The Art of Computer Programming


THE ART OF
COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 6A

A DRAFT OF
SECTION 7.2.2.2:
SATISFIABILITY

DONALD E. KNUTH *Stanford University*

The Art of Computer Programming

Wow! — Section 7.2.2.2 has turned out to be the longest section, by far, in *The Art of Computer Programming*. The SAT problem is evidently a "killer app," because it is key to the solution of so many problems. Consequently I can only hope that my lengthy treatment does not also kill off my faithful readers!

Donald Knuth

Handbook of Satisfiability

Frontiers in Artificial Intelligence and Applications

HANDBOOK of satisfiability

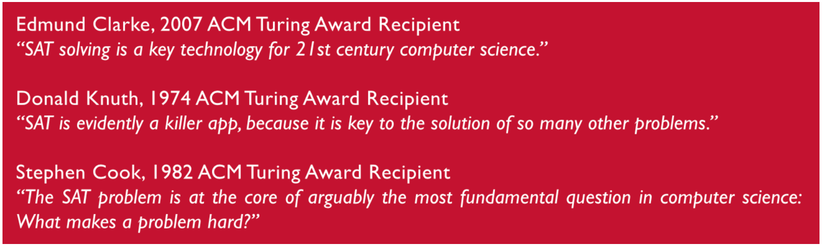
SECOND EDITION

Editors:
Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh

IOS Press

1400+ pages, 34 chapters

Handbook of Satisfiability



Edmund Clarke, 2007 ACM Turing Award Recipient
"SAT solving is a key technology for 21st century computer science."

Donald Knuth, 1974 ACM Turing Award Recipient
"SAT is evidently a killer app, because it is key to the solution of so many other problems."

Stephen Cook, 1982 ACM Turing Award Recipient
"The SAT problem is at the core of arguably the most fundamental question in computer science: What makes a problem hard?"

More Resources

- Annual conference, since 1996:
  `http://satisfiability.org`

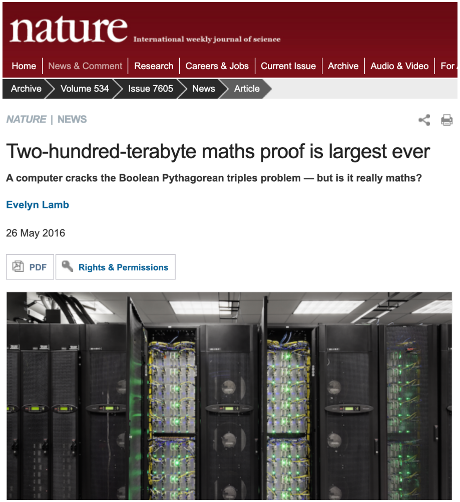
More Resources

- Annual conference, since 1996: `http://satisfiability.org`
- Annual competitions, since 2002: `http://www.satcompetition.org/`

More Resources

- Annual conference, since 1996:
  `http://satisfiability.org`
- Annual competitions, since 2002:
  `http://www.satcompetition.org/`
- Journal on Satisfiability, Boolean Modeling and Computation:
  `http://jsatjournal.org/`

Mathematical Proofs and SAT



NATURE | NEWS

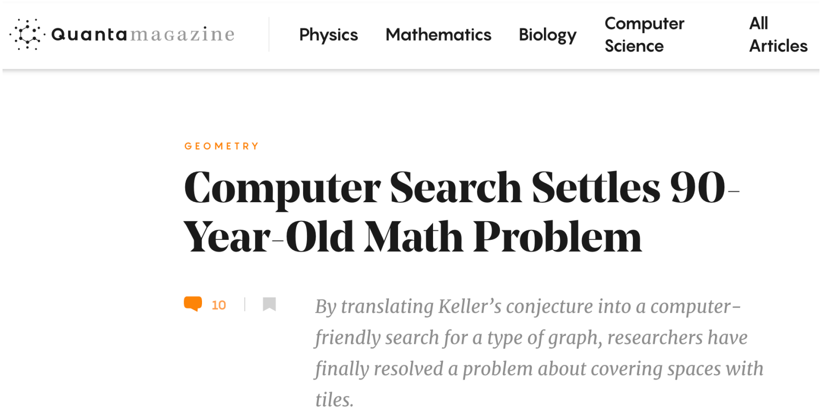
Two-hundred-terabyte maths proof is largest ever

A computer cracks the Boolean Pythagorean triples problem — but is it really maths?

Evelyn Lamb

26 May 2016

Mathematical Proofs and SAT



GEOMETRY

Computer Search Settles 90-Year-Old Math Problem

By translating Keller's conjecture into a computer-friendly search for a type of graph, researchers have finally resolved a problem about covering spaces with tiles.

This Talk

- Practice
  - With your own hands: will implement together simple programs for Sudoku, Queens, and Diagonals puzzles
  - Under the hood: algorithms used in SAT solvers
- Theory
  - Reductions: all hard problems reduce to SAT
  - Formal verification: proving unsatisfiability

Solving Puzzles Using SAT Solvers

Plan

- We’ll solve Sudoku, Queens, and Diagonals puzzles using SAT solvers
- Declarative programming: explain the rules of the game to a SAT solver — the SAT solver then finds a solution in the blink of an eye!

SAT Solvers: Input Format

```python
from pycosat import solve

clauses = [
    [-1, -2, -3],
    [1, -2],
    [2, -3],
    [3, -1],
    [1, 2, 3]
]

print(solve(clauses))
print(solve(clauses[1:]))
```

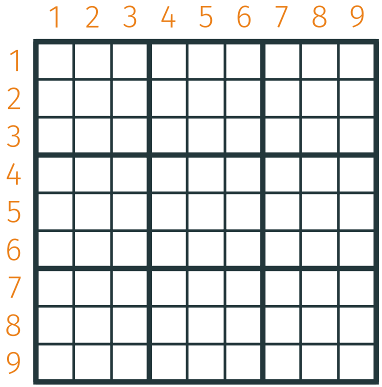
```
UNSAT
[1, 2, 3]
```

Sudoku: Puzzle Statement

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

Each row, each column, and each of the nine 3×3 blocks should contain different digits

Sudoku: Encoding



for each row r, column c, and digit d, $x_{rcd} = 1$ iff $T[r][c] = d$

Triple to an Integer

```python
def varnum(row, column, digit):
    assert row in range(1, 10)
    assert column in range(1, 10)
    assert digit in range(1, 10)
    return 100 * row + 10 * column + digit
```

At-Most-One Constraint

```
[1, 2, 3], [-1, -2], [-1, -3], [-2, -3]
```

```python
def exactly_one_of(literals):
    clauses = [[l for l in literals]]
    for pair in combinations(literals, 2):
        clauses.append([-l for l in pair])
    return clauses
```

```python
def one_digit_in_every_cell():
    clauses = []
    for row, column in product(range(1, 10), repeat=2):
        clauses += exactly_one_of([varnum(row, column, digit)
            for digit in range(1, 10)])
    return clauses

def one_digit_in_every_row():
    clauses = []
    for row, digit in product(range(1, 10), repeat=2):
        clauses += exactly_one_of([varnum(row, column, digit)
            for column in range(1, 10)])
    return clauses

def one_digit_in_every_column():
    clauses = []
    for column, digit in product(range(1, 10), repeat=2):
        clauses += exactly_one_of([varnum(row, column, digit)
    for row in range(1, 10)])
    return clauses
```

```python
def one_digit_in_every_block():
    clauses = []
    for row, column in product([1, 4, 7], repeat=2):
        for digit in range(1, 10):
            clauses += exactly_one_of(
                [varnum(row + a, column + b, digit)
                for (a, b) in product(range(3), repeat=2)]
            )
return clauses
```

Input Puzzle

```python
def solve_puzzle(puzzle):
    assert len(puzzle) == 9
    assert all(len(row) == 9 for row in puzzle)

    clauses = []
    clauses += one_digit_in_every_cell()
    clauses += one_digit_in_every_row()
    clauses += one_digit_in_every_column()
    clauses += one_digit_in_every_block()

    for row, column in product(range(1, 10), repeat=2):
        if puzzle[row - 1][column - 1] != "*":
            digit = int(puzzle[row - 1][column - 1])
            assert digit in range(1, 10)
            clauses += [[varnum(row, column, digit)]]

    solution = pycosat.solve(clauses)
```

Satisfying Assignment into Solution

```python
if isinstance(solution, str):
    print("No solution")
    exit()

assert isinstance(solution, list)

for row in range(1, 10):
    for column in range(1, 10):
        for digit in range(1, 10):
            if varnum(row, column, digit) in solution:
                print(digit, end="")
    print()
```

Let’s Run It!

- We are going to run the code now
- We’ll also show a more compact code (using `pysat` module) solving Queens and Diagonals puzzles

Under the Hood: Algorithms Used in SAT Solvers

Two Main Approaches

- Backtracking

Two Main Approaches

- Backtracking

  Assign literals one by one; if a conflict is found, learn it and backtrack

Two Main Approaches

- Backtracking

  Assign literals one by one; if a conflict is found, learn it and backtrack
- Local search

Two Main Approaches

- Backtracking

  Assign literals one by one; if a conflict is found, learn it and backtrack

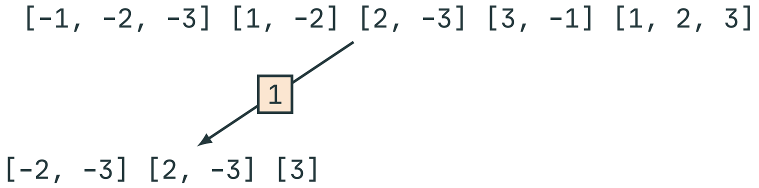
- Local search

  Starting from an assignment to variables, make local modifications to turn it into a satisfying assignment

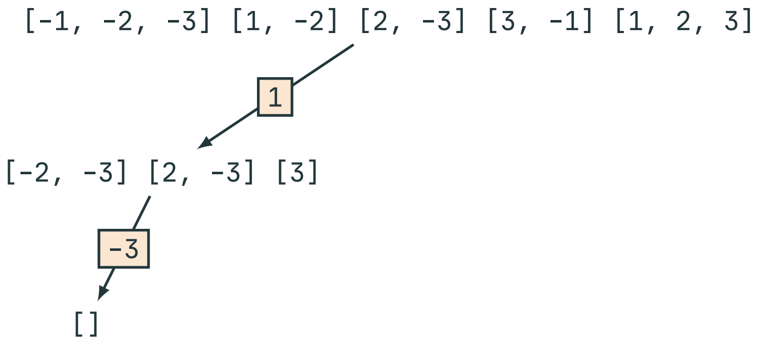
Backtracking

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

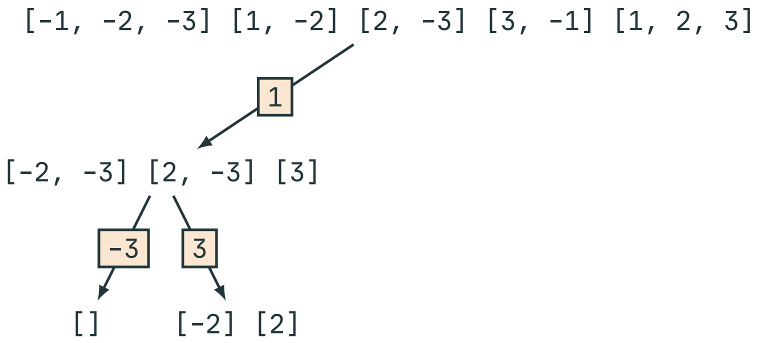
Backtracking



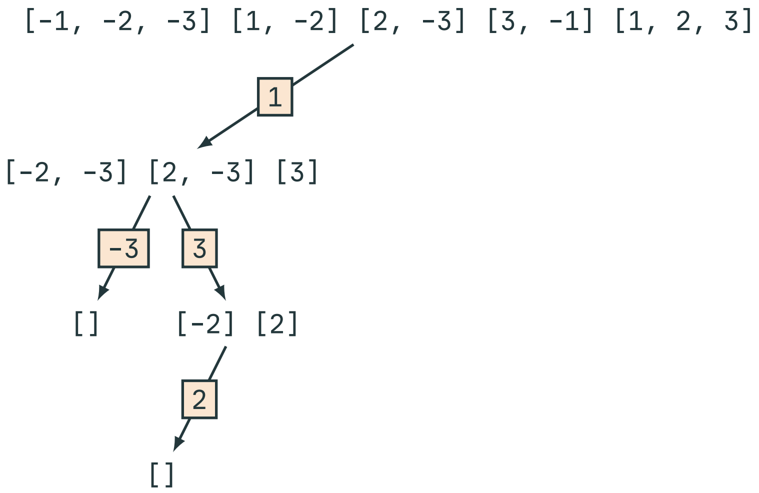
Backtracking



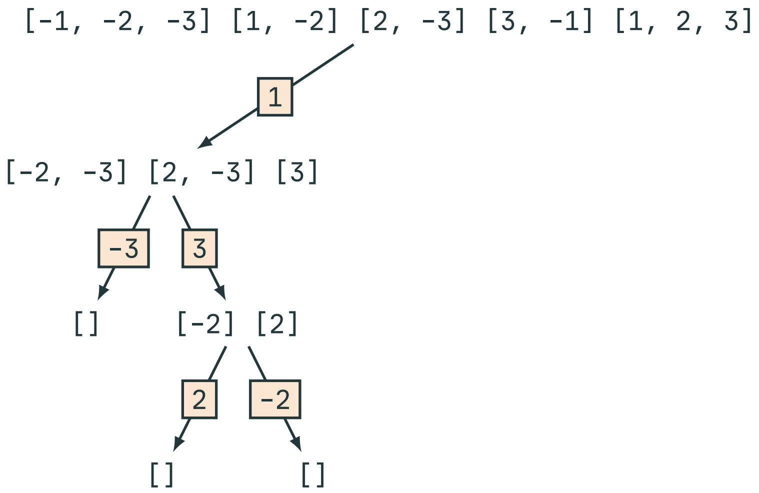
Backtracking



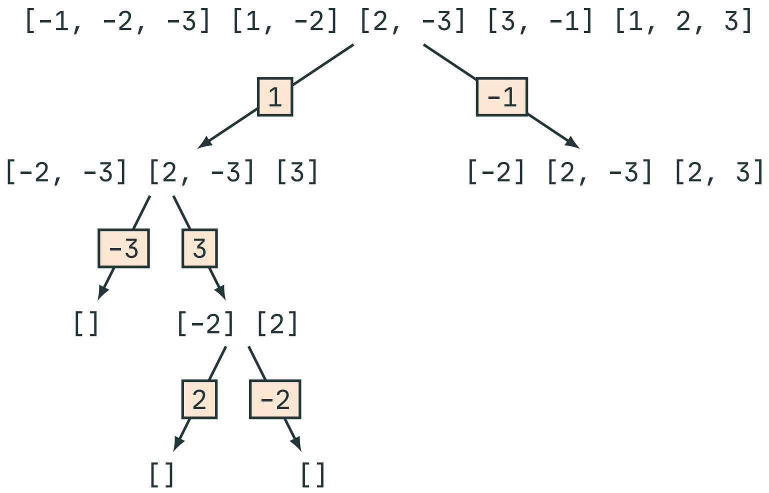
Backtracking



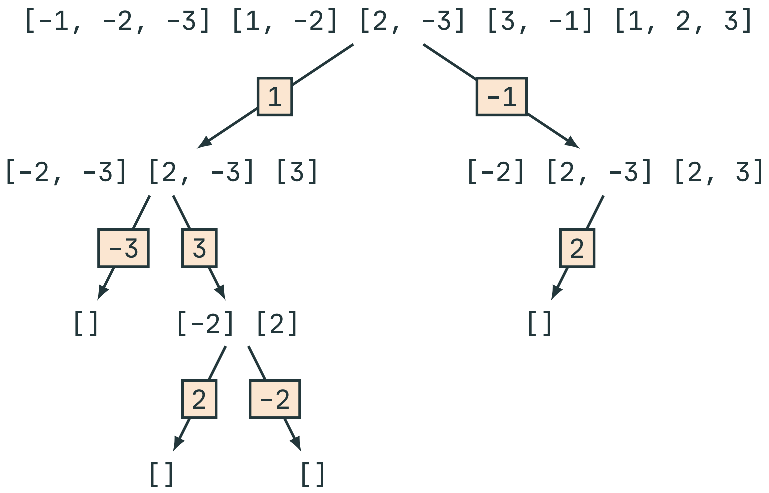
Backtracking



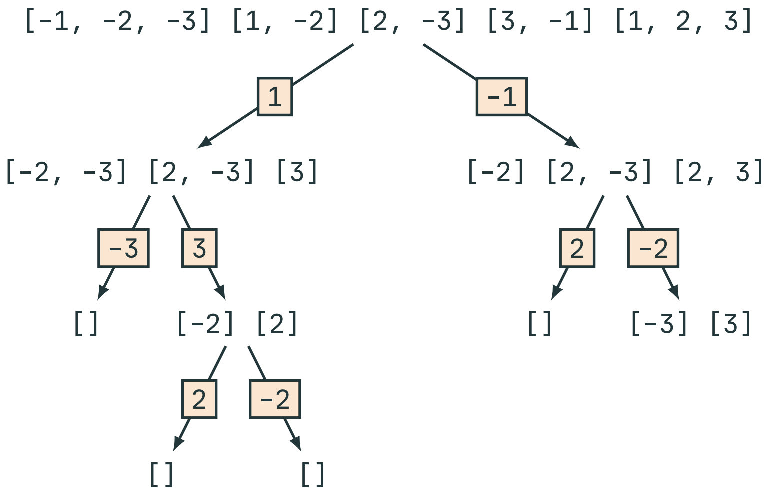
Backtracking



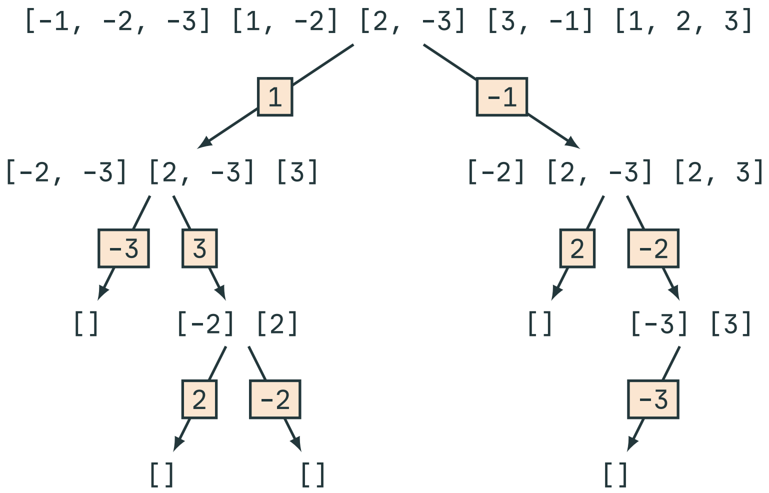
Backtracking



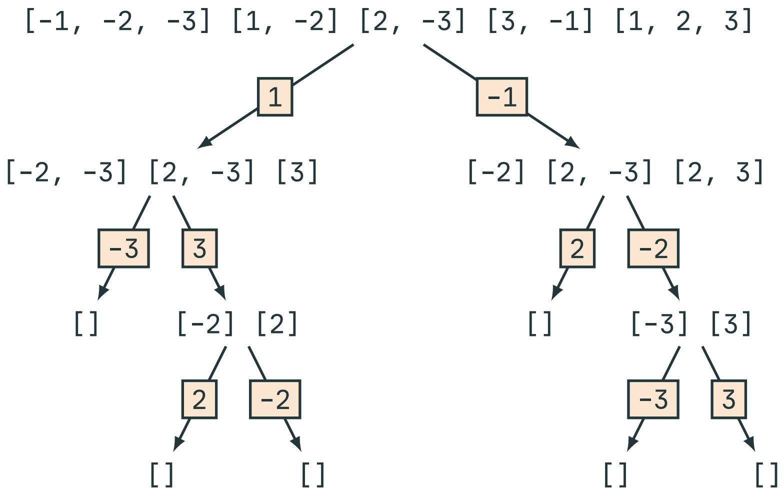
Backtracking



Backtracking



Backtracking



What Else?

- Reduction (simplification) rules: unit clauses, pure literals, etc

What Else?

- Reduction (simplification) rules: unit clauses, pure literals, etc
- Heuristics for selecting the next variable

What Else?

- Reduction (simplification) rules: unit clauses, pure literals, etc
- Heuristics for selecting the next variable
- Clause learning

What Else?

- Reduction (simplification) rules: unit clauses, pure literals, etc
- Heuristics for selecting the next variable
- Clause learning
- Efficient data structures (indices)

What Else?

- Reduction (simplification) rules: unit clauses, pure literals, etc
- Heuristics for selecting the next variable
- Clause learning
- Efficient data structures (indices)
- Optimization at every level

What’s next?

- In practice, SAT solvers are extremely efficient

What’s next?

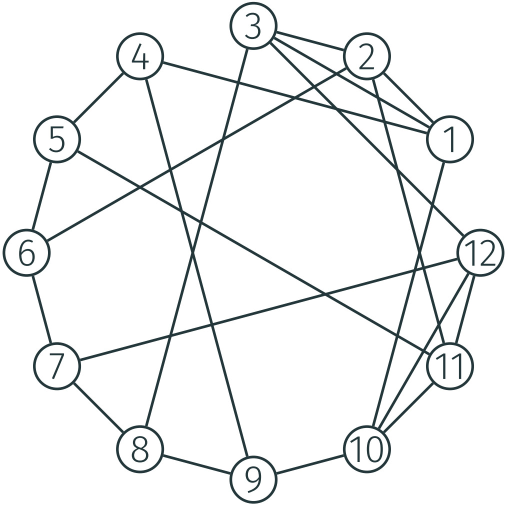
- In practice, SAT solvers are extremely efficient
- In theory, nobody can prove that there exists an algorithm that solves SAT on every formula faster than in 2^n (brute-force time)!

What’s next?

- In practice, SAT solvers are extremely efficient
- In theory, nobody can prove that there exists an algorithm that solves SAT on every formula faster than in 2^n (brute-force time)!
- Next part: theoretical aspects

Reductions: Every Hard Problem is SAT

Independent Set Problem



How many nodes can you select such that no two of them are adjacent?

Knapsack Problem

```
2991751 4213432 3513558 7994865
2730772 8316558 2035139 7170565
7850538 1221948 5759470 5335790
5855311 3424294 1408320 2446928
2938684 1932450 5026799 5493700
7208349 8988402 1432131 8291879
8589667 9134725 2525218 6675668
```

Can you select some numbers whose sum is 31960448? What about 31960449?

Common Property

SAT, Independent Set, and Knapsack problems share the following property:

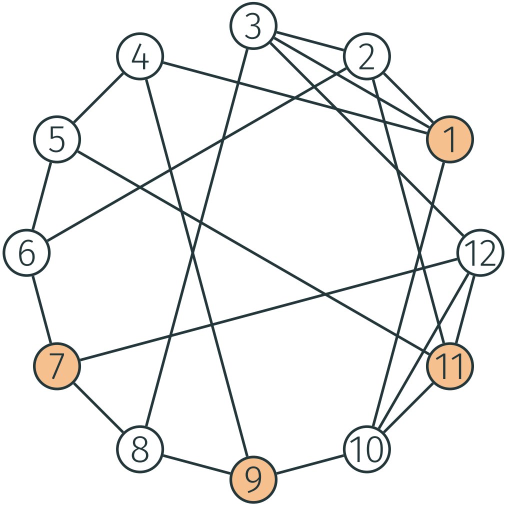
given a candidate solution, it is easy to check whether it is indeed a solution

Knapsack: Solution

2991751	4213432	**3513558**	**7994865**
2730772	8316558	2035139	**7170565**
7850538	1221948	5759470	5335790
5855311	3424294	1408320	2446928
2938684	**1932450**	5026799	**5493700**
7208349	8988402	1432131	8291879
8589667	9134725	2525218	6675668

The sum is 31960449

Independent Set: Solution



Class NP

- NP: class of efficiently verifiable problems

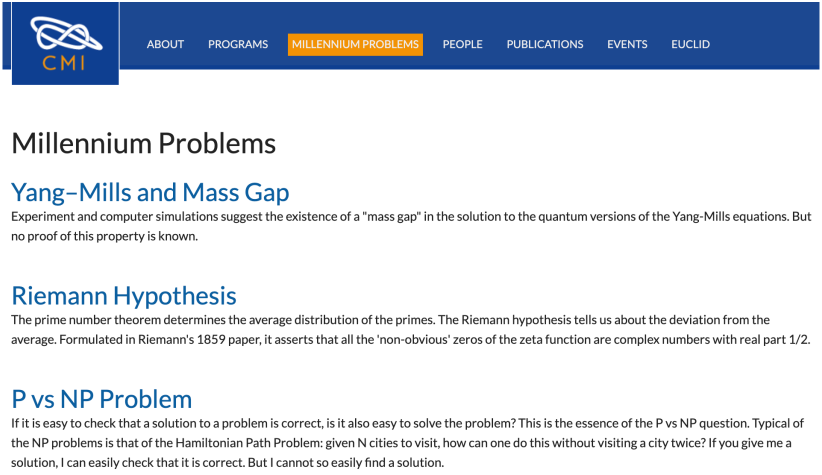
Class NP

- NP: class of efficiently verifiable problems
- P: class of efficiently solvable problems

Class NP

- NP: class of efficiently verifiable problems
- P: class of efficiently solvable problems
- Millennium Problem: Is P $=$ NP? In other words, do there exist an efficiently verifiable problem that cannot be solved efficiently?

Millennium Problems



Millennium Problems

Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

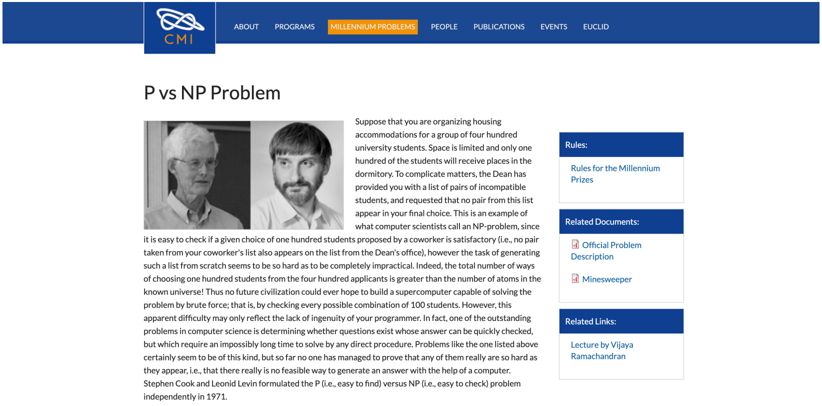
The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

`https://claymath.org/millennium-problems`

Millennium Problems



P vs NP Problem

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Rules:

Rules for the Millennium Prizes

Related Documents:

Official Problem Description

Minesweeper

Related Links:

Lecture by Vijaya Ramachandran

Reductions

- We’ve used a SAT solver in a black box fashion: state Sudoku as SAT and invoke a SAT solver

Reductions

- We’ve used a SAT solver in a black box fashion: state Sudoku as SAT and invoke a SAT solver
- *A* reduces to *B*, if an efficient program for *B* can be used to solve *A* efficiently

SAT is the Hardest

- Cook–Levin Theorem: every problem from NP reduces to SAT

SAT is the Hardest

- Cook–Levin Theorem: every problem from NP reduces to SAT
- Consequences:

SAT is the Hardest

- Cook–Levin Theorem: every problem from NP reduces to SAT
- Consequences:
    - If SAT can be solved efficiently, then P $=$ NP and there are efficient algorithms for all problems from NP

SAT is the Hardest

- Cook–Levin Theorem: every problem from NP reduces to SAT
- Consequences:
    - If SAT can be solved efficiently, then $P = NP$ and there are efficient algorithms for all problems from NP
    - If SAT cannot be solved efficiently, then $P \neq NP$ and many other important problems cannot be solved efficiently

SAT is the Hardest

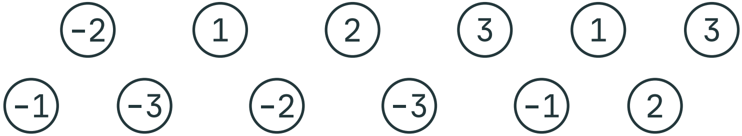
- Cook–Levin Theorem: every problem from NP reduces to SAT
- Consequences:
  - If SAT can be solved efficiently, then P $=$ NP and there are efficient algorithms for all problems from NP
  - If SAT cannot be solved efficiently, then P $\neq$ NP and many other important problems cannot be solved efficiently
- The same holds for Independent Set and Knapsack (as well as many other) problems

Reducing SAT to Independent Set

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

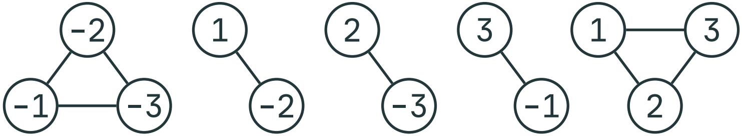
Reducing SAT to Independent Set

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```



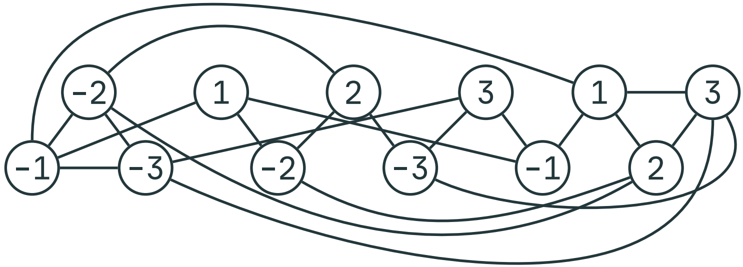
Reducing SAT to Independent Set

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```



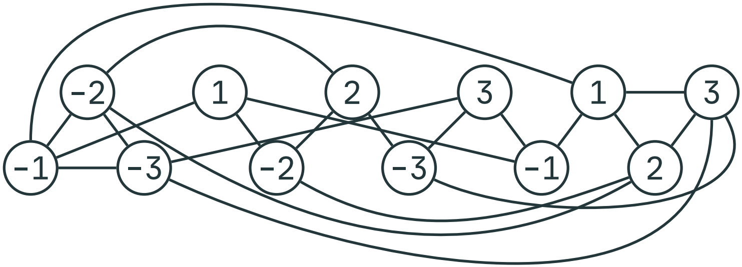
Reducing SAT to Independent Set

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```



Reducing SAT to Independent Set

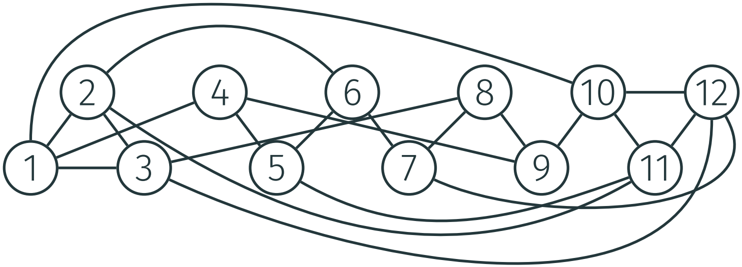
```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```



5-independent set ⇔ SAT

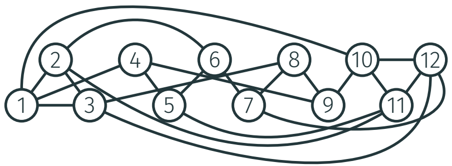
Reducing SAT to Independent Set

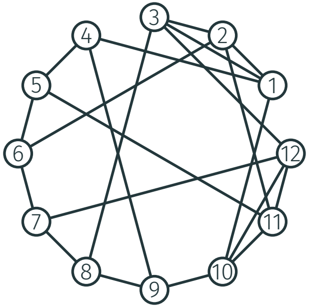
```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```



(we've seen this graph before!)

Reducing SAT to Independent Set





(we’ve seen this graph before!)

Fine-Grained Reductions

- SAT can also be reduced to various problems from P

Fine-Grained Reductions

- SAT can also be reduced to various problems from P
- If Edit Distance can be solved in time $n^{1.99}$, then SAT can be solved in time 1.9999^n

What’s Next?

- Many computational problems are hard for the same reason: either all of them have an efficient algorithm or none of them have

What’s Next?

- Many computational problems are hard for the same reason: either all of them have an efficient algorithm or none of them have
- Given a formula, one can construct a graph that have a large enough independent set iff the formula is satisfiable

What’s Next?

- Many computational problems are hard for the same reason: either all of them have an efficient algorithm or none of them have
- Given a formula, one can construct a graph that have a large enough independent set iff the formula is satisfiable
- Next part: how to prove that a formula is unsatisfiable?

Formal Verification: Proving Unsatisfiability

Typical Verification Setting

- Translate the statement “the system run into a forbidden state” to SAT and run a SAT solver

Typical Verification Setting

- Translate the statement “the system run into a forbidden state” to SAT and run a SAT solver
- If the formula is satisfiable, running into forbidden state is possible

Typical Verification Setting

- Translate the statement “the system run into a forbidden state” to SAT and run a SAT solver
- If the formula is satisfiable, running into forbidden state is possible
- If the formula is unsatisfiable, this is impossible

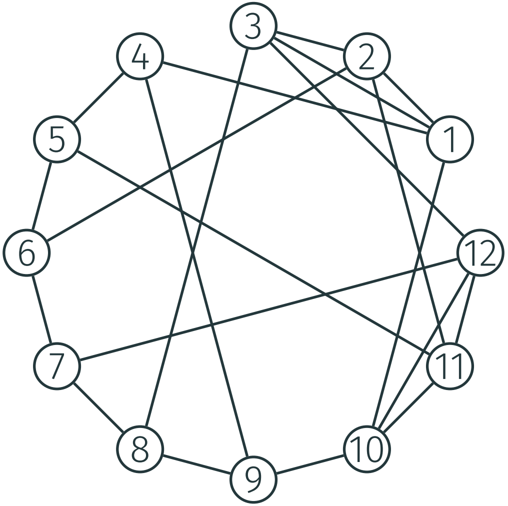
Satisfiability and Unsatisfiability

- Satisfiable formulas have short certificates: given a satisfying assignment, it is easy to verify that it indeed satisfies all clauses

Satisfiability and Unsatisfiability

- Satisfiable formulas have short certificates: given a satisfying assignment, it is easy to verify that it indeed satisfies all clauses
- How to convince yourself that a given formula is unsatisfiable? In other words, do there exist short certificates of unsatisfiability?

Large Independent Set



What is the reason there is no independent set of size five?

Knapsack Problem

```
2991751 4213432 3513558 7994865
2730772 8316558 2035139 7170565
7850538 1221948 5759470 5335790
5855311 3424294 1408320 2446928
2938684 1932450 5026799 5493700
7208349 8988402 1432131 8291879
8589667 9134725 2525218 6675668
```

Why there is no subset of items that sum up to 31960448?

Resolution: Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

Resolution: Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

- `[-1, -2, -3] [1, -2]` ⇒ `[-2, -3]`

Resolution: Example

`[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]`

- `[-1, -2, -3] [1, -2]` $\Rightarrow$ `[-2, -3]`
- `[-2, -3] [2, -3]` $\Rightarrow$ `[-3]`

Resolution: Example

`[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]`

- `[-1, -2, -3] [1, -2]` $\Rightarrow$ `[-2, -3]`
- `[-2, -3] [2, -3]` $\Rightarrow$ `[-3]`
- `[3, -1] [-3]` $\Rightarrow$ `[-1]`

Resolution: Example

`[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]`

- `[-1, -2, -3] [1, -2]` $\Rightarrow$ `[-2, -3]`
- `[-2, -3] [2, -3]` $\Rightarrow$ `[-3]`
- `[3, -1] [-3]` $\Rightarrow$ `[-1]`
- `[-3] [1, 2, 3]` $\Rightarrow$ `[1, 2]`

Resolution: Example

```
[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]
```

- `[-1, -2, -3] [1, -2]` ⇒ `[-2, -3]`
- `[-2, -3] [2, -3]` ⇒ `[-3]`
- `[3, -1] [-3]` ⇒ `[-1]`
- `[-3] [1, 2, 3]` ⇒ `[1, 2]`
- `[1, 2] [1, -2]` ⇒ `[1]`

Resolution: Example

`[-1, -2, -3] [1, -2] [2, -3] [3, -1] [1, 2, 3]`

- `[-1, -2, -3] [1, -2]` $\Rightarrow$ `[-2, -3]`
- `[-2, -3] [2, -3]` $\Rightarrow$ `[-3]`
- `[3, -1] [-3]` $\Rightarrow$ `[-1]`
- `[-3] [1, 2, 3]` $\Rightarrow$ `[1, 2]`
- `[1, 2] [1, -2]` $\Rightarrow$ `[1]`
- `[-1] [1]` $\Rightarrow$ `[]`

Resolution

- This is a toy example of a proof in the resolution proof system

Resolution

- This is a toy example of a proof in the resolution proof system
- Correctness: if the empty clause can be derived from a formula using the resolution rule, then the formula is unsatisfiable

Resolution

- This is a toy example of a proof in the resolution proof system
- Correctness: if the empty clause can be derived from a formula using the resolution rule, then the formula is unsatisfiable
- Completeness: for any unsatisfiable formula, one can derive the empty clause

Resolution

- This is a toy example of a proof in the resolution proof system
- Correctness: if the empty clause can be derived from a formula using the resolution rule, then the formula is unsatisfiable
- Completeness: for any unsatisfiable formula, one can derive the empty clause
- Backtracking-based solver construct (implicitly) a resolution proof

Proof Size

- The size of a proof can be enormous

Proof Size

- The size of a proof can be enormous
- In theory, there are unsatisfiable formulas that have no polynomial size resolution proof

Proof Size

- The size of a proof can be enormous
- In theory, there are unsatisfiable formulas that have no polynomial size resolution proof
- In practice, a proof may have size 200Tb

Pigeonhole Principle

If $n+1$ pigeons occupy n holes, then there exists a hole occupied by at least two pigeons

Works for Ages

```
n = 20
pigeons, holes = range(n + 1), range(n)

pool, formula = IDPool(), CNF()

for p in pigeons:
    formula.extend(CardEnc.atleast(
        lits=[pool.id((p, h)) for h in holes], bound=1, vpool=pool))

for h in holes:
    formula.extend(CardEnc.atmost(
        lits=[pool.id((p, h)) for p in pigeons], bound=1, vpool=pool)

solver = Solver(bootstrap_with=formula)
print(solver.solve())
```

Compact Code

```python
from pysat.solvers import Solver
from pysat.examples.genhard import PHP

solver = Solver(bootstrap_with=PHP(nof_holes=20))
solver.solve()
```

Stronger Proof Systems

- Is there a proof system having a short proof for every unsatisfiable formula?

Stronger Proof Systems

- Is there a proof system having a short proof for every unsatisfiable formula?
- We don’t know!

Stronger Proof Systems

- Is there a proof system having a short proof for every unsatisfiable formula?
- We don't know!
- This is the central question of proof complexity

Conclusion

Edmund Clarke, 2007 ACM Turing Award Recipient
"SAT solving is a key technology for 21st century computer science."

Donald Knuth, 1974 ACM Turing Award Recipient
"SAT is evidently a killer app, because it is key to the solution of so many other problems."

Stephen Cook, 1982 ACM Turing Award Recipient
"The SAT problem is at the core of arguably the most fundamental question in computer science: What makes a problem hard?"

Slides and source code:

`https://bit.ly/jb-sat`