CS498: Algorithmic Engineering

Lecture 20: SMT Applications: Puzzles, Verification, and Proof

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 11 – 04/02/2026

Outline

Today's Roadmap

1. **Encoding tricks:** cardinality, symmetry breaking, choosing the right theory.

Today's Roadmap

1. **Encoding tricks:** cardinality, symmetry breaking, choosing the right theory.
2. **Puzzles:** graph coloring (symmetry in action), Nonogram (clever encoding), circuit equivalence (negation trick).

Today's Roadmap

1. **Encoding tricks:** cardinality, symmetry breaking, choosing the right theory.
2. **Puzzles:** graph coloring (symmetry in action), Nonogram (clever encoding), circuit equivalence (negation trick).
3. **Software verification:** the most famous bug in binary search.

Today's Roadmap

1. **Encoding tricks:** cardinality, symmetry breaking, choosing the right theory.
2. **Puzzles:** graph coloring (symmetry in action), Nonogram (clever encoding), circuit equivalence (negation trick).
3. **Software verification:** the most famous bug in binary search.
4. **Mathematical proofs:** the corner-spheres paradox.

Today's Roadmap

1. **Encoding tricks:** cardinality, symmetry breaking, choosing the right theory.
2. **Puzzles:** graph coloring (symmetry in action), Nonogram (clever encoding), circuit equivalence (negation trick).
3. **Software verification:** the most famous bug in binary search.
4. **Mathematical proofs:** the corner-spheres paradox.

Theme: same negation trick, four very different applications.

The “At Most k” Problem

In Lecture 17, we encoded “exactly one of n Bools is True”:

- At-least-one: `Or`$(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

The “At Most k” Problem

In Lecture 17, we encoded “exactly one of n Bools is True”:

- At-least-one: $\texttt{Or}(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

The “At Most k” Problem

In Lecture 17, we encoded “exactly one of n Bools is True”:

- At-least-one: `Or`$(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

But “at most k of n” for general k? Naive: forbid every $(k+1)$-subset $\Rightarrow \binom{n}{k+1}$ clauses. For $n = 100,\ k = 10$: over 10^{13}.

The "At Most k" Problem

In Lecture 17, we encoded "exactly one of n Bools is True":

- At-least-one: `Or`$(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

But "at most k of n" for general k? Naive: forbid every $(k+1)$-subset $\Rightarrow \binom{n}{k+1}$ clauses. For $n = 100,\ k = 10$: over 10^{13}.

The fix: Sequential Counter (Sinz 2005):

- Introduce auxiliary Bools $r_{i,j}$ = "at least j of $x_1, \ldots, x_i$ are True."

The "At Most k" Problem

In Lecture 17, we encoded "exactly one of n Bools is True":

- At-least-one: `Or`$(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

But "at most k of n" for general k? Naive: forbid every $(k+1)$-subset $\Rightarrow \binom{n}{k+1}$ clauses. For $n = 100,\ k = 10$: over 10^{13}.

The fix: Sequential Counter (Sinz 2005):

- Introduce auxiliary Bools $r_{i,j}$ = "at least j of $x_1, \ldots, x_i$ are True."
- Propagation: if x_i is True and $r_{i-1,j-1}$, then $r_{i,j}$. If $r_{i-1,j}$, then $r_{i,j}$.

The "At Most k" Problem

In Lecture 17, we encoded "exactly one of n Bools is True":

- At-least-one: `Or`$(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

But "at most k of n" for general k? Naive: forbid every $(k+1)$-subset $\Rightarrow \binom{n}{k+1}$ clauses. For $n = 100,\ k = 10$: over 10^{13}.

The fix: Sequential Counter (Sinz 2005):

- Introduce auxiliary Bools $r_{i,j}$ = "at least j of $x_1, \ldots, x_i$ are True."
- Propagation: if x_i is True and $r_{i-1,j-1}$, then $r_{i,j}$. If $r_{i-1,j}$, then $r_{i,j}$.
- Forbid $r_{n,k+1}$: total count never exceeds k.

The "At Most k" Problem

In Lecture 17, we encoded "exactly one of n Bools is True":

- At-least-one: `Or`$(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

But "at most k of n" for general k? Naive: forbid every $(k+1)$-subset $\Rightarrow \binom{n}{k+1}$ clauses. For $n = 100,\ k = 10$: over 10^{13}.

The fix: Sequential Counter (Sinz 2005):

- Introduce auxiliary Bools $r_{i,j}$ = "at least j of $x_1, \ldots, x_i$ are True."
- Propagation: if x_i is True and $r_{i-1,j-1}$, then $r_{i,j}$. If $r_{i-1,j}$, then $r_{i,j}$.
- Forbid $r_{n,k+1}$: total count never exceeds k.
- Cost: $O(n \cdot k)$ variables and clauses.

The "At Most k" Problem

In Lecture 17, we encoded "exactly one of n Bools is True":

- At-least-one: $\texttt{Or}(x_1, \ldots, x_n)$. One clause.
- At-most-one: for every pair $i < j$, add $(\neg x_i \vee \neg x_j)$. That is $\binom{n}{2}$ clauses.

Worked great for Sudoku ($n = 9 \Rightarrow \binom{9}{2} = 36$ clauses per group).

But "at most k of n" for general k? Naive: forbid every $(k+1)$-subset $\Rightarrow \binom{n}{k+1}$ clauses. For $n = 100,\ k = 10$: over 10^{13}.

The fix: Sequential Counter (Sinz 2005):

- Introduce auxiliary Bools $r_{i,j}$ = "at least j of $x_1, \ldots, x_i$ are True."
- Propagation: if x_i is True and $r_{i-1,j-1}$, then $r_{i,j}$. If $r_{i-1,j}$, then $r_{i,j}$.
- Forbid $r_{n,k+1}$: total count never exceeds k.
- Cost: $O(n \cdot k)$ variables and clauses.

$\Rightarrow$ Auxiliary variables trade space for an *exponential* reduction in clauses.

Z3's Built-in: PbLe / PbGe

Implementing sequential counters by hand is tedious. Z3 wraps this into one function:

```
from z3 import *
xs = [Bool(f'x_{i}') for i in range(20)]
s = Solver()
s.add(PbLe([(x, 1) for x in xs], 5))    # at most 5 True      #New!!
s.add(PbGe([(x, 1) for x in xs], 3))    # at least 3 True     #New!!
# PbEq([(x, 1) for x in xs], 4)         # exactly 4 True
print(s.check())  # sat
```

Z3's Built-in: PbLe / PbGe

Implementing sequential counters by hand is tedious. Z3 wraps this into one function:

```
from z3 import *
xs = [Bool(f'x_{i}') for i in range(20)]
s = Solver()
s.add(PbLe([(x, 1) for x in xs], 5))    # at most 5 True      #New!!
s.add(PbGe([(x, 1) for x in xs], 3))    # at least 3 True     #New!!
# PbEq([(x, 1) for x in xs], 4)         # exactly 4 True
print(s.check())  # sat
```

`PbLe([(x, w), ...], k)`: weighted sum $\sum w_i \cdot x_i \leq k$. All weights 1 $\Rightarrow$ "at most k are True."

Z3's Built-in: PbLe / PbGe

Implementing sequential counters by hand is tedious. Z3 wraps this into one function:

```
from z3 import *
xs = [Bool(f'x_{i}') for i in range(20)]
s = Solver()
s.add(PbLe([(x, 1) for x in xs], 5))    # at most 5 True      #New!!
s.add(PbGe([(x, 1) for x in xs], 3))    # at least 3 True     #New!!
# PbEq([(x, 1) for x in xs], 4)         # exactly 4 True
print(s.check())  # sat
```

`PbLe([(x, w), ...], k)`: weighted sum $\sum w_i \cdot x_i \leq k$. All weights 1 $\Rightarrow$ "at most k are True."

Internally, Z3 picks the best encoding (sequential counters, sorting networks, or cardinality networks). You get the efficient encoding for free.

Z3's Built-in: PbLe / PbGe

Implementing sequential counters by hand is tedious. Z3 wraps this into one function:

```
from z3 import *
xs = [Bool(f'x_{i}') for i in range(20)]
s = Solver()
s.add(PbLe([(x, 1) for x in xs], 5))    # at most 5 True      #New!!
s.add(PbGe([(x, 1) for x in xs], 3))    # at least 3 True     #New!!
# PbEq([(x, 1) for x in xs], 4)         # exactly 4 True
print(s.check())  # sat
```

`PbLe([(x, w), ...], k)`: weighted sum $\sum w_i \cdot x_i \leq k$. All weights 1 $\Rightarrow$ "at most k are True."

Internally, Z3 picks the best encoding (sequential counters, sorting networks, or cardinality networks). You get the efficient encoding for free.

Encoding lesson

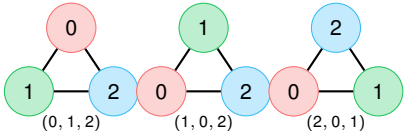
When you need "at most k" for $k > 1$, Use `PbLe/PbGe/PbEq`.

Symmetry Breaking

Example: 3-color a triangle (A–B–C–A) with colors $\{0, 1, 2\}$.

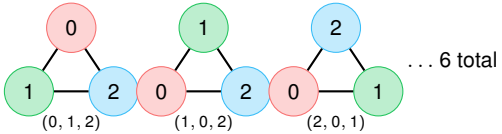
Symmetry Breaking

Example: 3-color a triangle (A–B–C–A) with colors $\{0, 1, 2\}$.



Symmetry Breaking

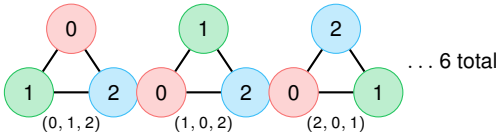
Example: 3-color a triangle (A–B–C–A) with colors $\{0, 1, 2\}$.



All 6 are just permutations of the same coloring pattern. The solver explores all $3! = 6$. Five-sixths of its work is redundant.

Symmetry Breaking

Example: 3-color a triangle (A–B–C–A) with colors $\{0, 1, 2\}$.



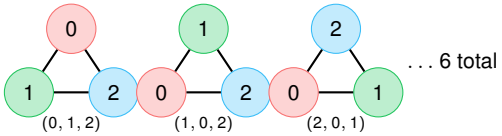
All 6 are just permutations of the same coloring pattern. The solver explores all $3! = 6$. Five-sixths of its work is redundant.

Fix: force vertex A to color 0 (rename colors so it is always first). More generally: vertex i uses color $\leq i$.

```
s.add(c[0] == 0)                    # fix first vertex
for i in range(1, n):
    s.add(c[i] <= i)                # only "needed" colors
```

Symmetry Breaking

Example: 3-color a triangle (A–B–C–A) with colors $\{0, 1, 2\}$.



All 6 are just permutations of the same coloring pattern. The solver explores all $3! = 6$. Five-sixths of its work is redundant.

Fix: force vertex A to color 0 (rename colors so it is always first). More generally: vertex i uses color $\leq i$.

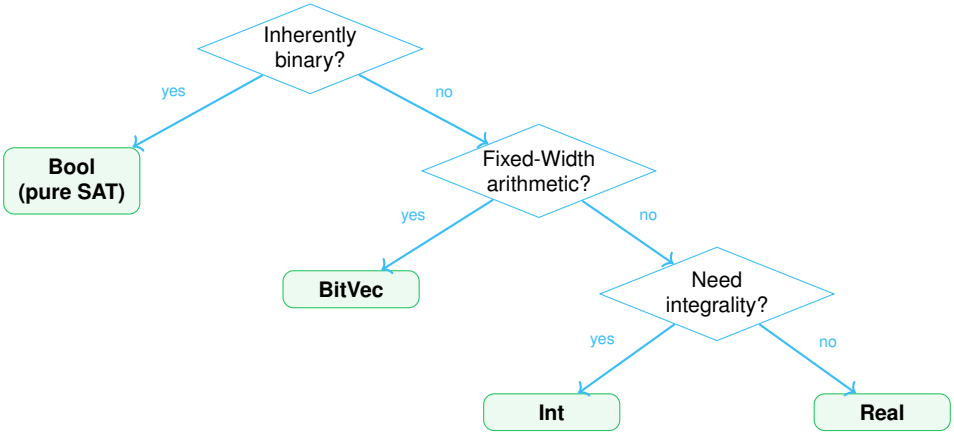
```
s.add(c[0] == 0)                    # fix first vertex
for i in range(1, n):
    s.add(c[i] <= i)                # only "needed" colors
```

Now only $(0, 1, 2)$ survives. $\sim 9\times$ speedup for $k = 5$; over $100\times$ for $k = 10$.

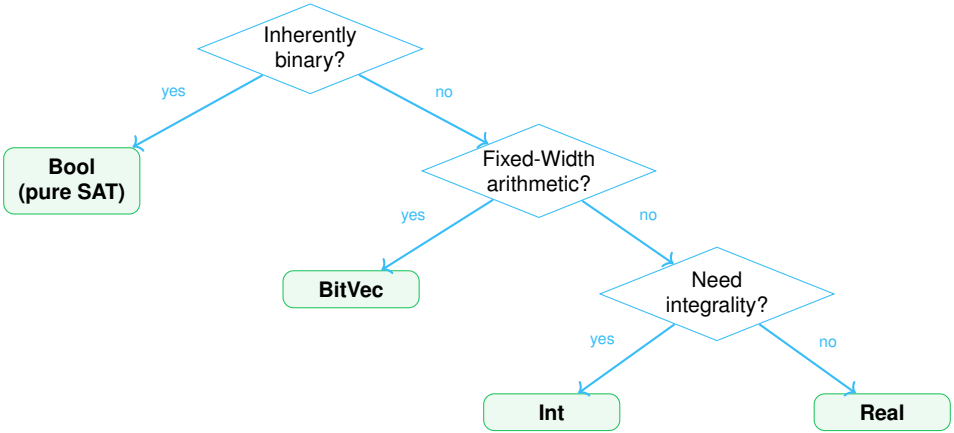
Choosing the Right Encoding

Encoding	Best when	Watch out for
Bool (SAT)	Inherently binary choices.	Blows up for counting/arithmetic.
Int (LIA)	Counting, permutations,	Nonlinear $\Rightarrow$ undecidable (NIA).
BitVec	Hardware, software, crypto, fixed-width arithmetic.	Bit-blasting large widths is slow.
Real (LRA/NRA)	Continuous quantities. Nonlinear OK (decidable).	Cannot express integrality.

Decision Flowchart



Decision Flowchart



This is a starting point. In practice, the best encoding often **mixes** theories: Bool grid + Int positions (Nonograms), BitVec for machine arithmetic (verification).

The Graph Coloring Problem

Given: an undirected graph $G = (V, E)$ and k colors.

The Graph Coloring Problem

Given: an undirected graph $G = (V, E)$ and k colors.

Question: can we assign a color to every vertex so that no two adjacent vertices share a color?

The Graph Coloring Problem

Given: an undirected graph $G = (V, E)$ and k colors.

Question: can we assign a color to every vertex so that no two adjacent vertices share a color?

Classic application: map coloring: countries are vertices, shared borders are edges. The Four Color Theorem says every planar map needs at most 4 colors.

The Graph Coloring Problem

Given: an undirected graph $G = (V, E)$ and k colors.

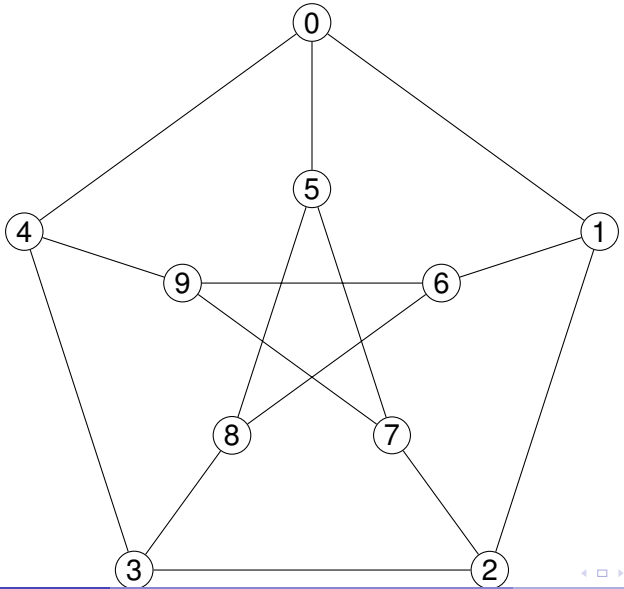
Question: can we assign a color to every vertex so that no two adjacent vertices share a color?

Classic application: map coloring: countries are vertices, shared borders are edges. The Four Color Theorem says every planar map needs at most 4 colors.

Encoding choices:

- **Int** variable per vertex: $c_i \in \{0, \ldots, k-1\}$.
- Edge constraint: $c_u \neq c_v$ for each $(u, v) \in E$.
- Symmetry breaking: fix $c_0 = 0$, and $c_i \leq i$ (from our encoding tricks).

Sample Graph



Graph Coloring: Z3 Code

```
from z3 import *
# Petersen graph: 10 vertices, 15 edges, chromatic number = 3
edges = [(0,1),(0,4),(0,5),(1,2),(1,6),(2,3),(2,7),
         (3,4),(3,8),(4,9),(5,7),(5,8),(6,8),(6,9),(7,9)]
n, k = 10, 3
c = [Int(f'c_{i}') for i in range(n)]

s = Solver()
for i in range(n):
    s.add(c[i] >= 0, c[i] < k)
for u, v in edges:
    s.add(c[u] != c[v])                    # adjacency
# Symmetry breaking
s.add(c[0] == 0)
for i in range(1, n):
    s.add(c[i] <= i)
```

Graph Coloring: Z3 Code

```
from z3 import *
# Petersen graph: 10 vertices, 15 edges, chromatic number = 3
edges = [(0,1),(0,4),(0,5),(1,2),(1,6),(2,3),(2,7),
         (3,4),(3,8),(4,9),(5,7),(5,8),(6,8),(6,9),(7,9)]
n, k = 10, 3
c = [Int(f'c_{i}') for i in range(n)]

s = Solver()
for i in range(n):
    s.add(c[i] >= 0, c[i] < k)
for u, v in edges:
    s.add(c[u] != c[v])                        # adjacency
# Symmetry breaking
s.add(c[0] == 0)
for i in range(1, n):
    s.add(c[i] <= i)
print(s.check())  # sat
m = s.model()
print([m[c[i]].as_long() for i in range(n)])
# e.g. [0, 1, 2, 1, 2, 2, 2, 1, 0, 0]
```

Graph Coloring: Z3 Code

```
from z3 import *
# Petersen graph: 10 vertices, 15 edges, chromatic number = 3
edges = [(0,1),(0,4),(0,5),(1,2),(1,6),(2,3),(2,7),
         (3,4),(3,8),(4,9),(5,7),(5,8),(6,8),(6,9),(7,9)]
n, k = 10, 3
c = [Int(f'c_{i}') for i in range(n)]

s = Solver()
for i in range(n):
    s.add(c[i] >= 0, c[i] < k)
for u, v in edges:
    s.add(c[u] != c[v])                    # adjacency
# Symmetry breaking
s.add(c[0] == 0)
for i in range(1, n):
    s.add(c[i] <= i)
print(s.check())  # sat
m = s.model()
print([m[c[i]].as_long() for i in range(n)])
# e.g. [0, 1, 2, 1, 2, 2, 2, 1, 0, 0]
```

The Petersen graph is a classic: 3-colorable but not 2-colorable. Z3 finds a valid 3-coloring instantly.

Graph Coloring: Proving a Lower Bound

Can we 2-color the Petersen graph?

```
s2 = Solver()
for i in range(n):
    s2.add(c[i] >= 0, c[i] < 2)             # only 2 colors
for u, v in edges:
    s2.add(c[u] != c[v])

print(s2.check())  # unsat -- impossible!
```

Graph Coloring: Proving a Lower Bound

Can we 2-color the Petersen graph?

```
s2 = Solver()
for i in range(n):
    s2.add(c[i] >= 0, c[i] < 2)            # only 2 colors
for u, v in edges:
    s2.add(c[u] != c[v])

print(s2.check())  # unsat -- impossible!
```

UNSAT = no 2-coloring exists. Combined with SAT for $k = 3$: the chromatic number is **exactly 3**.

Graph Coloring: Proving a Lower Bound

Can we 2-color the Petersen graph?

```
s2 = Solver()
for i in range(n):
    s2.add(c[i] >= 0, c[i] < 2)            # only 2 colors
for u, v in edges:
    s2.add(c[u] != c[v])

print(s2.check())  # unsat -- impossible!
```

UNSAT = no 2-coloring exists. Combined with SAT for $k = 3$: the chromatic number is **exactly 3**.

The pattern:

- SAT for k: a valid k-coloring exists (upper bound).

Graph Coloring: Proving a Lower Bound

Can we 2-color the Petersen graph?

```
s2 = Solver()
for i in range(n):
    s2.add(c[i] >= 0, c[i] < 2)            # only 2 colors
for u, v in edges:
    s2.add(c[u] != c[v])

print(s2.check())  # unsat -- impossible!
```

UNSAT = no 2-coloring exists. Combined with SAT for $k = 3$: the chromatic number is **exactly 3**.

The pattern:

- SAT for k: a valid k-coloring exists (upper bound).
- UNSAT for $k - 1$: no $(k - 1)$-coloring exists (lower bound).

Graph Coloring: Proving a Lower Bound

Can we 2-color the Petersen graph?

```
s2 = Solver()
for i in range(n):
    s2.add(c[i] >= 0, c[i] < 2)            # only 2 colors
for u, v in edges:
    s2.add(c[u] != c[v])

print(s2.check())  # unsat -- impossible!
```

UNSAT = no 2-coloring exists. Combined with SAT for $k = 3$: the chromatic number is **exactly 3**.

The pattern:

- SAT for k: a valid k-coloring exists (upper bound).
- UNSAT for $k - 1$: no $(k - 1)$-coloring exists (lower bound).
- Together: exact chromatic number. Same negation trick, now bounding an optimization quantity.

Graph Coloring: Proving a Lower Bound

Can we 2-color the Petersen graph?

```
s2 = Solver()
for i in range(n):
    s2.add(c[i] >= 0, c[i] < 2)            # only 2 colors
for u, v in edges:
    s2.add(c[u] != c[v])

print(s2.check())  # unsat -- impossible!
```

UNSAT = no 2-coloring exists. Combined with SAT for $k = 3$: the chromatic number is **exactly 3**.

The pattern:

- SAT for k: a valid k-coloring exists (upper bound).
- UNSAT for $k - 1$: no $(k-1)$-coloring exists (lower bound).
- Together: exact chromatic number. Same negation trick, now bounding an optimization quantity.

$\Rightarrow$ SMT can *prove* graph properties, not just find solutions.

What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

Rules:

- You have an $m \times n$ grid. Each cell is either **filled** or **empty**.

What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

Rules:

- You have an $m \times n$ grid. Each cell is either **filled** or **empty**.
- Each row and column has a list of **run lengths**: consecutive groups of filled cells.

What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

Rules:

- You have an $m \times n$ grid. Each cell is either **filled** or **empty**.
- Each row and column has a list of **run lengths**: consecutive groups of filled cells.
- Runs must appear in order, with at least one empty cell between consecutive runs.

What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

Rules:

- You have an $m \times n$ grid. Each cell is either **filled** or **empty**.
- Each row and column has a list of **run lengths**: consecutive groups of filled cells.
- Runs must appear in order, with at least one empty cell between consecutive runs.

Example (5×5): Row clues: [1,1], [5], [5], [3], [1]. Column: [2], [4], [4], [4], [2].

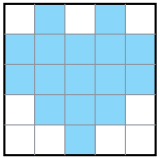
What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

Rules:

- You have an $m \times n$ grid. Each cell is either **filled** or **empty**.
- Each row and column has a list of **run lengths**: consecutive groups of filled cells.
- Runs must appear in order, with at least one empty cell between consecutive runs.

Example (5×5): Row clues: [1,1], [5], [5], [3], [1]. Column: [2], [4], [4], [4], [2].



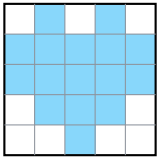
What is a Nonogram?

A Nonogram (also called Picross or Hanjie) is a picture logic puzzle.

Rules:

- You have an $m \times n$ grid. Each cell is either **filled** or **empty**.
- Each row and column has a list of **run lengths**: consecutive groups of filled cells.
- Runs must appear in order, with at least one empty cell between consecutive runs.

Example (5×5): Row clues: [1,1], [5], [5], [3], [1]. Column: [2], [4], [4], [4], [2].



Observation: cells are binary, so Bool variables are natural. But the run-length constraints are the challenge.

Nonogram: The Encoding Strategy

Variables: `grid`$[i][j]$ is a Bool. True = filled.

Nonogram: The Encoding Strategy

Variables: `grid`$[i][j]$ is a Bool. True = filled.

The trick: for each row (or column), introduce auxiliary `Int` variables for the **start position** of each run.

Nonogram: The Encoding Strategy

Variables: `grid`$[i][j]$ is a Bool. True = filled.

The trick: for each row (or column), introduce auxiliary `Int` variables for the **start position** of each run.

If row i has runs of lengths $[r_1, r_2, \ldots, r_p]$:

- Let $s_1, s_2, \ldots, s_p$ be the start positions (Int variables).

Nonogram: The Encoding Strategy

Variables: `grid`$[i][j]$ is a Bool. True = filled.

The trick: for each row (or column), introduce auxiliary `Int` variables for the **start position** of each run.

If row i has runs of lengths $[r_1, r_2, \ldots, r_p]$:

- Let $s_1, s_2, \ldots, s_p$ be the start positions (Int variables).
- $s_1 \geq 0$. Each run fits: $s_j + r_j \leq n$.

Nonogram: The Encoding Strategy

Variables: `grid`$[i][j]$ is a Bool. True = filled.

The trick: for each row (or column), introduce auxiliary `Int` variables for the **start position** of each run.

If row i has runs of lengths $[r_1, r_2, \ldots, r_p]$:

- Let $s_1, s_2, \ldots, s_p$ be the start positions (Int variables).
- $s_1 \geq 0$. Each run fits: $s_j + r_j \leq n$.
- Runs are ordered with gaps: $s_{j+1} \geq s_j + r_j + 1$ (at least one empty cell between runs).

Nonogram: The Encoding Strategy

Variables: `grid`$[i][j]$ is a Bool. True = filled.

The trick: for each row (or column), introduce auxiliary `Int` variables for the **start position** of each run.

If row i has runs of lengths $[r_1, r_2, \ldots, r_p]$:

- Let $s_1, s_2, \ldots, s_p$ be the start positions (Int variables).
- $s_1 \geq 0$. Each run fits: $s_j + r_j \leq n$.
- Runs are ordered with gaps: $s_{j+1} \geq s_j + r_j + 1$ (at least one empty cell between runs).
- A cell (i, c) is filled iff it falls within some run: $\texttt{grid}[i][c] \Leftrightarrow \bigvee_j (s_j \leq c < s_j + r_j)$.

Nonogram: Z3 Code

```
from z3 import *

rows_clues, cols_clues = [[1,1],[5],[5],[3],[1]], [[2],[4],[4],[4],[2]]
R, C = len(rows_clues), len(cols_clues)
grid = [[Bool(f'g_{i}_{j}') for j in range(C)] for i in range(R)]
s = Solver()
def add_line_constraints(cells, clues, tag):
    n = len(cells)
    starts = [Int(f'{tag}_s{k}') for k in range(len(clues))]
    for k, r in enumerate(clues):
        s.add(starts[k] >= 0, starts[k] + r - 1 < n)         # fits in line
        if k > 0: s.add(starts[k] >= starts[k-1] + clues[k-1] + 1)
    for c_idx in range(n):
        in_some_run = Or([And(starts[k] <= c_idx, c_idx < starts[k] + clues[k])
                          for k in range(len(clues))])
        s.add(cells[c_idx] == in_some_run)

for i in range(R):
    add_line_constraints(grid[i], rows_clues[i], f'r{i}')
for j in range(C):
    add_line_constraints([grid[i][j] for i in range(R)], cols_clues[j], f'c{j}')
```

Nonogram: Solution

```
print(s.check())  # sat
m = s.model()
for i in range(R):
    row = ''.join('#' if is_true(m[grid[i][j]]) else '.'
                  for j in range(C))
    print(row)

# .#.#.
# #####
# #####
# .###.
# ..#..
```

Nonogram: Solution

```
print(s.check())  # sat
m = s.model()
for i in range(R):
    row = ''.join('#' if is_true(m[grid[i][j]]) else '.'
                  for j in range(C))
    print(row)

# .#.#.
# #####
# #####
# .###.
# ..#..
```

Encoding pattern for Nonograms:

- Bool grid for filled/empty.

Nonogram: Solution

```
print(s.check())  # sat
m = s.model()
for i in range(R):
    row = ''.join('#' if is_true(m[grid[i][j]]) else '.'
                  for j in range(C))
    print(row)

# .#.#.
# #####
# #####
# .###.
# ..#..
```

Encoding pattern for Nonograms:

- Bool grid for filled/empty.
- Int start-position variables for each run in each line.

Nonogram: Solution

```
print(s.check())  # sat
m = s.model()
for i in range(R):
    row = ''.join('#' if is_true(m[grid[i][j]]) else '.'
                  for j in range(C))
    print(row)

# .#.#.
# #####
# #####
# .###.
# ..#..
```

Encoding pattern for Nonograms:

- Bool grid for filled/empty.
- Int start-position variables for each run in each line.
- Link grid cells to runs via biconditional ($\Leftrightarrow$).

Nonogram: Solution

```
print(s.check())  # sat
m = s.model()
for i in range(R):
    row = ''.join('#' if is_true(m[grid[i][j]]) else '.'
                  for j in range(C))
    print(row)

# .#.#.
# #####
# #####
# .###.
# ..#..
```

Encoding pattern for Nonograms:

- Bool grid for filled/empty.
- Int start-position variables for each run in each line.
- Link grid cells to runs via biconditional ($\Leftrightarrow$).

$\Rightarrow$ A mixed Bool + Int encoding. Each theory handles what it does best.

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

The naive approach: test all 2^n inputs. For $n = 64$: impossible.

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

The naive approach: test all 2^n inputs. For $n = 64$: impossible.

The negation trick:

1. Encode both circuits: $f(x)$ and $g(x)$.

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

The naive approach: test all 2^n inputs. For $n = 64$: impossible.

The negation trick:

1. Encode both circuits: $f(x)$ and $g(x)$.
2. Ask Z3: "is there an input x such that $f(x) \neq g(x)$?"

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

The naive approach: test all 2^n inputs. For $n = 64$: impossible.

The negation trick:

1. Encode both circuits: $f(x)$ and $g(x)$.
2. Ask Z3: "is there an input x such that $f(x) \neq g(x)$?"
3. If **UNSAT**: no such input exists $\Rightarrow$ the circuits are **equivalent**.
4. If **SAT**: the model gives a **counterexample** where they differ.

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

The naive approach: test all 2^n inputs. For $n = 64$: impossible.

The negation trick:

1. Encode both circuits: $f(x)$ and $g(x)$.
2. Ask Z3: "is there an input x such that $f(x) \neq g(x)$?"
3. If **UNSAT**: no such input exists $\Rightarrow$ the circuits are **equivalent**.
4. If **SAT**: the model gives a **counterexample** where they differ.

Key insight: instead of proving equivalence (hard, universal quantifier), we search for a *difference* (easy, existential quantifier). UNSAT = proof of equivalence.

Verifying Two Circuits Are Equivalent

Problem: you have two Boolean circuits (or two implementations of the same function). Are they equivalent on all inputs?

The naive approach: test all 2^n inputs. For $n = 64$: impossible.

The negation trick:

1. Encode both circuits: $f(x)$ and $g(x)$.
2. Ask Z3: "is there an input x such that $f(x) \neq g(x)$?"
3. If **UNSAT**: no such input exists $\Rightarrow$ the circuits are **equivalent**.
4. If **SAT**: the model gives a **counterexample** where they differ.

Key insight: instead of proving equivalence (hard, universal quantifier), we search for a *difference* (easy, existential quantifier). UNSAT = proof of equivalence.

This is the same "proof by contradiction" pattern used in formal verification.

Circuit Equivalence: Z3 with BitVec

```
from z3 import *

x = BitVec('x', 32)  #New: 32-bit

# Circuit 1: clever bit-trick to check if x is a power of 2
f = If(x != 0, (x & (x - 1)) == 0, BoolVal(False))  # x & (x-1) == 0

# Circuit 2: naive approach
g = Or([x == (1 << i) for i in range(31)])

s = Solver()
s.add(f != g)                                        #the negation trick
result = s.check()
```

Circuit Equivalence: Z3 with BitVec

```
from z3 import *

x = BitVec('x', 32)  #New: 32-bit

# Circuit 1: clever bit-trick to check if x is a power of 2
f = If(x != 0, (x & (x - 1)) == 0, BoolVal(False))  # x & (x-1) == 0

# Circuit 2: naive approach
g = Or([x == (1 << i) for i in range(31)])

s = Solver()
s.add(f != g)                                    #the negation trick
result = s.check()
print(result)  # sat -- they differ!

if result == sat:
    m = s.model()
    val = m[x].as_long()
    print(f"Counterexample: x = {val}")          # x = 2147483648 (2^31)
```

Circuit Equivalence: Z3 with BitVec

```
from z3 import *

x = BitVec('x', 32)  #New: 32-bit

# Circuit 1: clever bit-trick to check if x is a power of 2
f = If(x != 0, (x & (x - 1)) == 0, BoolVal(False))  # x & (x-1) == 0

# Circuit 2: naive approach
g = Or([x == (1 << i) for i in range(31)])

s = Solver()
s.add(f != g)                                    #the negation trick
result = s.check()
print(result)  # sat -- they differ!

if result == sat:
    m = s.model()
    val = m[x].as_long()
    print(f"Counterexample: x = {val}")          # x = 2147483648 (2^31)
```

Z3 found the bug: the naive circuit forgets 2^{31} (it only checks $i < 31$). BitVec makes this precise: fixed-width semantics catch overflow and boundary bugs.

Circuit Equivalence: Proving Correctness

```
from z3 import *

x = BitVec('x', 8)
#Compute popcount(x) mod 2 (parity of number of set bits in x)
# Method 1: fold parity down to the last bit
f = x
f = f ^ (f >> 1)
f = f ^ (f >> 2)
f = f ^ (f >> 4)
parity1 = f & 1

# Method 2: XOR the bits directly
parity2 = ((x >> 0) & 1) ^ ((x >> 1) & 1) ^ ((x >> 2) & 1) ^ ((x >> 3) & 1) \
        ^ ((x >> 4) & 1) ^ ((x >> 5) & 1) ^ ((x >> 6) & 1) ^ ((x >> 7) & 1)

s = Solver()
s.add(parity1 != parity2)
print(s.check())   # unsat
```

Circuit Equivalence: Proving Correctness

```
from z3 import *

x = BitVec('x', 8)
#Compute popcount(x) mod 2 (parity of number of set bits in x)
# Method 1: fold parity down to the last bit
f = x
f = f ^ (f >> 1)
f = f ^ (f >> 2)
f = f ^ (f >> 4)
parity1 = f & 1

# Method 2: XOR the bits directly
parity2 = ((x >> 0) & 1) ^ ((x >> 1) & 1) ^ ((x >> 2) & 1) ^ ((x >> 3) & 1) \
        ^ ((x >> 4) & 1) ^ ((x >> 5) & 1) ^ ((x >> 6) & 1) ^ ((x >> 7) & 1)

s = Solver()
s.add(parity1 != parity2)
print(s.check())   # unsat
```

Two genuinely different implementations (XOR-fold vs. bit-by-bit). Z3 proves they agree on all $2^8 = 256$ inputs. Scale to 64 bits: still instant, because bit-blasting creates a compact SAT problem.

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The buggy line:

```
mid = (low + high) / 2
```

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The buggy line:

```
mid = (low + high) / 2
```

The problem: when `low` and `high` are both large 32-bit signed ints, their sum **overflows**.

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The buggy line:

```
mid = (low + high) / 2
```

The problem: when `low` and `high` are both large 32-bit signed ints, their sum **overflows**.

Example: `low = 1,500,000,000`, `high = 2,000,000,000`.

- Sum $= 3{,}500{,}000{,}000 > 2^{31} - 1 = 2{,}147{,}483{,}647$.

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The buggy line:

```
mid = (low + high) / 2
```

The problem: when `low` and `high` are both large 32-bit signed ints, their sum **overflows**.

Example: `low = 1,500,000,000`, `high = 2,000,000,000`.

- Sum $= 3{,}500{,}000{,}000 > 2^{31} - 1 = 2{,}147{,}483{,}647$.
- In 32-bit signed arithmetic, this wraps to a **negative** number.
- `mid` becomes negative $\Rightarrow$ array index out of bounds $\Rightarrow$ crash.

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The buggy line:

```
mid = (low + high) / 2
```

The problem: when `low` and `high` are both large 32-bit signed ints, their sum **overflows**.

Example: `low = 1,500,000,000`, `high = 2,000,000,000`.

- Sum $= 3{,}500{,}000{,}000 > 2^{31} - 1 = 2{,}147{,}483{,}647$.
- In 32-bit signed arithmetic, this wraps to a **negative** number.
- `mid` becomes negative $\Rightarrow$ array index out of bounds $\Rightarrow$ crash.

The fix:

```
mid = low + (high - low) / 2
```

The Most Famous Bug in Binary Search

Joshua Bloch, 2006: Java's `Arrays.binarySearch` had a bug **for 9 years**.

The buggy line:

```
mid = (low + high) / 2
```

The problem: when `low` and `high` are both large 32-bit signed ints, their sum **overflows**.

Example: `low = 1,500,000,000`, `high = 2,000,000,000`.

- Sum $= 3{,}500{,}000{,}000 > 2^{31} - 1 = 2{,}147{,}483{,}647$.
- In 32-bit signed arithmetic, this wraps to a **negative** number.
- `mid` becomes negative $\Rightarrow$ array index out of bounds $\Rightarrow$ crash.

The fix:

```
mid = low + (high - low) / 2
```

No overflow: `high - low` fits in a 32-bit signed int when $0 \leq$ `low` $\leq$ `high`.

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

SMT approach: model the arithmetic *exactly* as the machine does it.

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

SMT approach: model the arithmetic *exactly* as the machine does it.

- `BitVec(32)`: 32-bit signed integers with wrap-around overflow.

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

SMT approach: model the arithmetic *exactly* as the machine does it.

- `BitVec(32)`: 32-bit signed integers with wrap-around overflow.
- Preconditions: $0 \leq \texttt{low} \leq \texttt{high}$.

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

SMT approach: model the arithmetic *exactly* as the machine does it.

- `BitVec(32)`: 32-bit signed integers with wrap-around overflow.
- Preconditions: $0 \leq$ `low` $\leq$ `high`.
- Ask: "does `low + high` ever produce a negative result?"

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

SMT approach: model the arithmetic *exactly* as the machine does it.

- `BitVec(32)`: 32-bit signed integers with wrap-around overflow.
- Preconditions: $0 \leq$ `low` $\leq$ `high`.
- Ask: "does `low + high` ever produce a negative result?"
- The solver searches all 2^{64} possible (low, high) pairs instantly.

Why SMT Finds This

Testing this bug requires arrays with over a billion elements. Most test suites never exercise inputs that large.

SMT approach: model the arithmetic *exactly* as the machine does it.

- `BitVec(32)`: 32-bit signed integers with wrap-around overflow.
- Preconditions: $0 \leq$ `low` $\leq$ `high`.
- Ask: "does `low + high` ever produce a negative result?"
- The solver searches all 2^{64} possible (low, high) pairs instantly.

Key insight: `Int` would miss this: mathematical integers don't overflow. `BitVec(32)` models **exact machine arithmetic**.

Finding the Overflow: Z3 Code

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0)                    # signed comparison
s.add(hi >= 0)                    # signed comparison
s.add(lo <= hi)                   # signed comparison
s.add((lo + hi) < 0)              # wrapped sum looks negative (signed)
```

Finding the Overflow: Z3 Code

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0)                    # signed comparison
s.add(hi >= 0)                    # signed comparison
s.add(lo <= hi)                   # signed comparison
s.add((lo + hi) < 0)              # wrapped sum looks negative (signed)
print(s.check())   # sat -- overflow exists!
m = s.model()
print(f"lo = {m[lo].as_signed_long()}") # 1
print(f"hi = {m[hi].as_signed_long()}") # 2147483647
print(f"lo + hi (signed) = {(m.eval(lo + hi)).as_signed_long()}") # -2147483648
```

Finding the Overflow: Z3 Code

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0)                    # signed comparison
s.add(hi >= 0)                    # signed comparison
s.add(lo <= hi)                   # signed comparison
s.add((lo + hi) < 0)              # wrapped sum looks negative (signed)
print(s.check())   # sat -- overflow exists!
m = s.model()
print(f"lo = {m[lo].as_signed_long()}") # 1
print(f"hi = {m[hi].as_signed_long()}") # 2147483647
print(f"lo + hi (signed) = {(m.eval(lo + hi)).as_signed_long()}") # -2147483648
```

SAT. Z3 found concrete values where `lo + hi` wraps negative. This is a real bug that crashed Java programs on large arrays.

Verifying the Fix

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0, hi >= 0, lo <= hi)   # signed comparisons on bit-vectors
```

Verifying the Fix

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0, hi >= 0, lo <= hi)   # signed comparisons on bit-vectors
mid = lo + (hi - lo) / 2            # fixed midpoint

# Look for a counterexample: mid < lo or mid > hi
s.add(Or(mid < lo, mid > hi))
```

Verifying the Fix

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0, hi >= 0, lo <= hi)   # signed comparisons on bit-vectors
mid = lo + (hi - lo) / 2            # fixed midpoint

# Look for a counterexample: mid < lo or mid > hi
s.add(Or(mid < lo, mid > hi))
print(s.check())   # unsat
```

Verifying the Fix

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0, hi >= 0, lo <= hi)   # signed comparisons on bit-vectors
mid = lo + (hi - lo) / 2            # fixed midpoint

# Look for a counterexample: mid < lo or mid > hi
s.add(Or(mid < lo, mid > hi))
print(s.check())   # unsat
```

UNSAT. For every pair $0 \leq \texttt{lo} \leq \texttt{hi}$, the fixed midpoint is in $[\texttt{lo}, \texttt{hi}]$. Proven.

Verifying the Fix

```
from z3 import *

lo = BitVec('lo', 32)
hi = BitVec('hi', 32)

s = SolverFor("QF_BV")
s.add(lo >= 0, hi >= 0, lo <= hi)   # signed comparisons on bit-vectors
mid = lo + (hi - lo) / 2            # fixed midpoint

# Look for a counterexample: mid < lo or mid > hi
s.add(Or(mid < lo, mid > hi))
print(s.check())   # unsat
```

UNSAT. For every pair $0 \leq \texttt{lo} \leq \texttt{hi}$, the fixed midpoint is in $[\texttt{lo}, \texttt{hi}]$. Proven.

Two queries, two results:

- `(lo + hi) / 2`: SAT $\Rightarrow$ overflow bug exists.
- `lo + (hi - lo) / 2`: UNSAT $\Rightarrow$ fix verified for all inputs.

Binary Search Verification: Takeaways

1. **BitVec = exact machine model.** Int misses overflow. BitVec catches it.

Binary Search Verification: Takeaways

1. **BitVec = exact machine model.** Int misses overflow. BitVec catches it.
2. **The negation trick again.** “Does an overflow exist?” is existential. SAT = yes, with a concrete witness.

Binary Search Verification: Takeaways

1. **BitVec = exact machine model.** Int misses overflow. BitVec catches it.
2. **The negation trick again.** “Does an overflow exist?” is existential. SAT = yes, with a concrete witness.
3. **Same pattern as circuit equivalence.** Assert the bad thing, check SAT/UNSAT.

Binary Search Verification: Takeaways

1. **BitVec = exact machine model.** Int misses overflow. BitVec catches it.
2. **The negation trick again.** “Does an overflow exist?” is existential. SAT = yes, with a concrete witness.
3. **Same pattern as circuit equivalence.** Assert the bad thing, check SAT/UNSAT.
4. **This is how real tools work.** CBMC, KLEE, and other bounded model checkers use exactly this approach: encode program semantics in BitVec, negate the spec, query SMT solver.

Binary Search Verification: Takeaways

1. **BitVec = exact machine model.** Int misses overflow. BitVec catches it.
2. **The negation trick again.** “Does an overflow exist?” is existential. SAT = yes, with a concrete witness.
3. **Same pattern as circuit equivalence.** Assert the bad thing, check SAT/UNSAT.
4. **This is how real tools work.** CBMC, KLEE, and other bounded model checkers use exactly this approach: encode program semantics in BitVec, negate the spec, query SMT solver.

The lesson

A 9-year-old bug in a JDK standard library, invisible to billions of test runs, is found in milliseconds by a 10-line Z3 script.

The Corner-Spheres Construction

Recipe:

The Corner-Spheres Construction

Recipe:

1. Take the cube $[0, 4]^d$.

The Corner-Spheres Construction

Recipe:

1. Take the cube $[0, 4]^d$.
2. Place a unit sphere (radius 1) at each corner of the inner 2×2 grid. Centers at $\{1, 3\}^d$.

The Corner-Spheres Construction

Recipe:

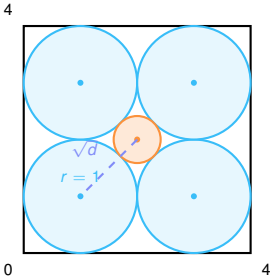
1. Take the cube $[0, 4]^d$.
2. Place a unit sphere (radius 1) at each corner of the inner 2×2 grid. Centers at $\{1, 3\}^d$.
3. Fit a central sphere at $(2, \ldots, 2)$, tangent to all corner spheres.

The Corner-Spheres Construction

Recipe:

1. Take the cube $[0, 4]^d$.
2. Place a unit sphere (radius 1) at each corner of the inner 2×2 grid. Centers at $\{1, 3\}^d$.
3. Fit a central sphere at $(2, \ldots, 2)$, tangent to all corner spheres.

In 2D: 4 corner spheres, 1 small central sphere. Everything fits comfortably inside.



The Paradox: Central Sphere Radius

Distance from center $(2, \ldots, 2)$ to any corner center (e.g. $(1, 1, \ldots, 1)$):

$$\sqrt{(2-1)^2 + (2-1)^2 + \cdots} = \sqrt{\underbrace{1 + 1 + \cdots + 1}_{d}} = \sqrt{d}$$

The Paradox: Central Sphere Radius

Distance from center $(2, \ldots, 2)$ to any corner center (e.g. $(1, 1, \ldots, 1)$):

$$\sqrt{(2-1)^2 + (2-1)^2 + \cdots} = \sqrt{\underbrace{1 + 1 + \cdots + 1}_{d}} = \sqrt{d}$$

Central sphere is tangent $\Rightarrow$ its radius is: $R = \sqrt{d} - 1$

The Paradox: Central Sphere Radius

Distance from center $(2, \ldots, 2)$ to any corner center (e.g. $(1, 1, \ldots, 1)$):

$$\sqrt{(2-1)^2 + (2-1)^2 + \cdots} = \sqrt{\underbrace{1 + 1 + \cdots + 1}_{d}} = \sqrt{d}$$

Central sphere is tangent $\Rightarrow$ its radius is: $R = \sqrt{d} - 1$

The cube face is at distance 2 from the center (e.g., coordinate 0 is at distance 2 from coordinate 2).

The Paradox: Central Sphere Radius

Distance from center $(2, \ldots, 2)$ to any corner center (e.g. $(1, 1, \ldots, 1)$):

$$\sqrt{(2-1)^2 + (2-1)^2 + \cdots} = \sqrt{\underbrace{1 + 1 + \cdots + 1}_{d}} = \sqrt{d}$$

Central sphere is tangent $\Rightarrow$ its radius is: $R = \sqrt{d} - 1$

The cube face is at distance 2 from the center (e.g., coordinate 0 is at distance 2 from coordinate 2).

Conjecture to verify: Is the sphere always inside the box for all dimensions? can Z3 *prove* this with exact arithmetic?

Z3 Code: Corner Spheres

```
from z3 import *

def corner_spheres(d):
    s = Solver()
    R = Real('R')
    p = [Real(f'p_{i}') for i in range(d)]           #point on sphere

    # Central sphere radius: (R+1)^2 = d  (avoids sqrt)
    s.add((R + 1) * (R + 1) == d)
    s.add(R >= 0)

    # p is on the central sphere: sum (p_i - 2)^2 = R^2
    s.add(Sum([(p[i] - 2) * (p[i] - 2) for i in range(d)]) == R * R)

    # p is outside the cube [0,4]^d
    s.add(Or([Or(p[i] < 0, p[i] > 4) for i in range(d)]))

    return s.check()
```

Z3 Code: Corner Spheres

```
from z3 import *

def corner_spheres(d):
    s = Solver()
    R = Real('R')
    p = [Real(f'p_{i}') for i in range(d)]           #point on sphere

    # Central sphere radius: (R+1)^2 = d  (avoids sqrt)
    s.add((R + 1) * (R + 1) == d)
    s.add(R >= 0)

    # p is on the central sphere: sum (p_i - 2)^2 = R^2
    s.add(Sum([(p[i] - 2) * (p[i] - 2) for i in range(d)]) == R * R)

    # p is outside the cube [0,4]^d
    s.add(Or([Or(p[i] < 0, p[i] > 4) for i in range(d)]))

    return s.check()
```

- `Real` variables: exact algebraic numbers, no floating-point error.
- $(R+1)^2 = d$ encodes $R = \sqrt{d} - 1$ without needing a square root function.

The Phase Transition

```python
for d in range(2, 11):
    result = corner_spheres(d)
    print(f"d = {d:2d}: {result}")
```

The Phase Transition

```python
for d in range(2, 11):
    result = corner_spheres(d)
    print(f"d = {d:2d}: {result}")
```

```
d =  2: unsat          d =  7: unsat
d =  3: unsat          d =  8: unsat
d =  4: unsat          d =  9: unsat    <-- R = 2 exactly (touches)
d =  5: unsat          d = 10: sat      <-- R = 2.16... (escapes!)
d =  6: unsat
```

The Phase Transition

```
for d in range(2, 11):
    result = corner_spheres(d)
    print(f"d = {d:2d}: {result}")
```

```
d =  2: unsat          d =  7: unsat
d =  3: unsat          d =  8: unsat
d =  4: unsat          d =  9: unsat    <-- R = 2 exactly (touches)
d =  5: unsat          d = 10: sat      <-- R = 2.16... (escapes!)
d =  6: unsat
```

UNSAT for $d \leq 9$: Z3 *proved* no point on the sphere is outside the cube.

The Phase Transition

```python
for d in range(2, 11):
    result = corner_spheres(d)
    print(f"d = {d:2d}: {result}")
```

```
d =  2: unsat          d =  7: unsat
d =  3: unsat          d =  8: unsat
d =  4: unsat          d =  9: unsat    <-- R = 2 exactly (touches)
d =  5: unsat          d = 10: sat      <-- R = 2.16... (escapes!)
d =  6: unsat
```

UNSAT for $d \leq 9$: Z3 *proved* no point on the sphere is outside the cube.
SAT for $d \geq 10$: Z3 gives a *concrete* point that is on the sphere and outside the cube. This is a computer-verified proof of the paradox.

The Phase Transition

```python
for d in range(2, 11):
    result = corner_spheres(d)
    print(f"d = {d:2d}: {result}")
```

```
d =  2: unsat          d =  7: unsat
d =  3: unsat          d =  8: unsat
d =  4: unsat          d =  9: unsat    <-- R = 2 exactly (touches)
d =  5: unsat          d = 10: sat      <-- R = 2.16... (escapes!)
d =  6: unsat
```

UNSAT for $d \leq 9$: Z3 *proved* no point on the sphere is outside the cube.
SAT for $d \geq 10$: Z3 gives a *concrete* point that is on the sphere and outside the cube. This is a computer-verified proof of the paradox.

Z3 uses **exact algebraic numbers** (NRA theory). No floating-point approximation. This is a rigorous mathematical proof, not a numerical simulation.

Corner Spheres: What We Learned

1. **NRA is decidable.** Tarski (1951) proved the theory of real closed fields is decidable. Z3 implements this via cylindrical algebraic decomposition.

Corner Spheres: What We Learned

1. **NRA is decidable.** Tarski (1951) proved the theory of real closed fields is decidable. Z3 implements this via cylindrical algebraic decomposition.
2. **Real = exact.** No rounding, no epsilon. When Z3 says UNSAT over the reals, it is a mathematical proof.

Corner Spheres: What We Learned

1. **NRA is decidable.** Tarski (1951) proved the theory of real closed fields is decidable. Z3 implements this via cylindrical algebraic decomposition.
2. **Real = exact.** No rounding, no epsilon. When Z3 says UNSAT over the reals, it is a mathematical proof.
3. **Phase transitions.** SMT is great for finding the exact boundary where a property flips. Sweep a parameter, look for the SAT/UNSAT transition.

Corner Spheres: What We Learned

1. **NRA is decidable.** Tarski (1951) proved the theory of real closed fields is decidable. Z3 implements this via cylindrical algebraic decomposition.
2. **Real = exact.** No rounding, no epsilon. When Z3 says UNSAT over the reals, it is a mathematical proof.
3. **Phase transitions.** SMT is great for finding the exact boundary where a property flips. Sweep a parameter, look for the SAT/UNSAT transition.
4. **Curse of dimensionality.** In high d, the diagonal of the cube ($\sqrt{d} \cdot 4$) grows much faster than the side length (4). The corners are "far away," leaving room for the central sphere to escape.

Corner Spheres: What We Learned

1. **NRA is decidable.** Tarski (1951) proved the theory of real closed fields is decidable. Z3 implements this via cylindrical algebraic decomposition.
2. **Real = exact.** No rounding, no epsilon. When Z3 says UNSAT over the reals, it is a mathematical proof.
3. **Phase transitions.** SMT is great for finding the exact boundary where a property flips. Sweep a parameter, look for the SAT/UNSAT transition.
4. **Curse of dimensionality.** In high d, the diagonal of the cube ($\sqrt{d} \cdot 4$) grows much faster than the side length (4). The corners are "far away," leaving room for the central sphere to escape.

The lesson

SMT is not just for puzzles and programs, it can verify mathematical claims with full rigor, answering questions where human geometric intuition fails.