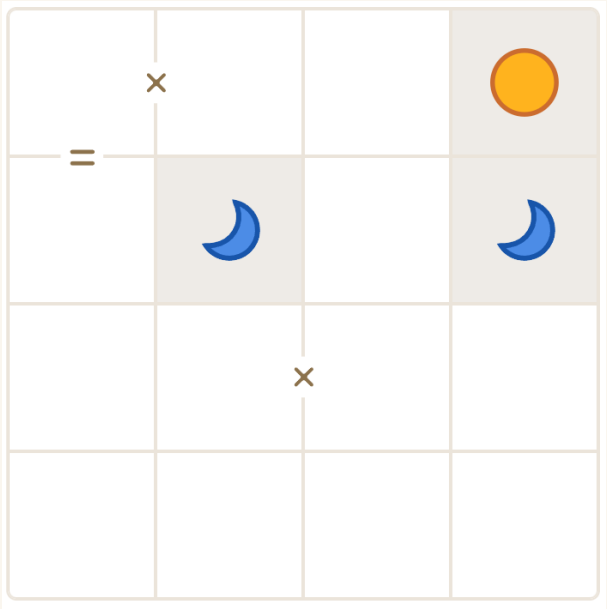
Vamshi Jandhyala

Building a LinkedIn Tango Solver with Z3

✠

November 2024

Tango is an engaging logic game played on a square grid that LinkedIn ships as one of its daily puzzles. In what follows we set out the rules and build a solver in Python using the Z3 theorem prover, with a matplotlib visualisation of the solutions.



Game rules

The puzzle is played on an $n \times n$ grid. Each cell must contain either a sun or a moon, subject to the following:

1. **Adjacent symbols.** No more than two suns or two moons may be adjacent to each other, horizontally or vertically.
2. **Balance.** Each row and column must contain an equal number of suns and moons.
3. **Equal signs.** Cells connected by an = sign must contain the same symbol.
4. **Opposite signs.** Cells connected by an × sign must contain opposite symbols.
5. **Unique solution.** Each puzzle has exactly one solution that can be found through logical deduction.

Implementation with Z3

We model each cell as a Boolean variable, where `True` represents a sun and `False` a moon.

Setting up the grid

```
def solve_tango(size, equals_constraints, opposite_constraints,
              initial_suns=None, initial_moons=None):
    solver = Solver()
    cells = [[Bool(f"cell_{i}_{j}") for j in range(size)]
            for i in range(size)]
```

Rule 1: adjacent symbols

To enforce "no more than two adjacent symbols," we check every sequence of three cells horizontally and vertically. We use Z3's `Sum` and `If` to count suns:

```
def count_suns(cell_list):
    return Sum([If(cell, 1, 0) for cell in cell_list])

for i in range(size):
    for j in range(size - 2):
        triple = [cells[i][j], cells[i][j + 1], cells[i][j + 2]]
        solver.add(count_suns(triple) <= 2) # at most 2 suns
        solver.add(count_suns(triple) >= 1) # at most 2 moons
```

A symmetric block enforces the rule on vertical triples.

Rule 2: balance

```
half = size // 2
for i in range(size):
    solver.add(count_suns(cells[i]) == half)
    solver.add(count_suns([cells[j][i] for j in range(size)]) == half)
```

Rules 3 and 4: equal and opposite

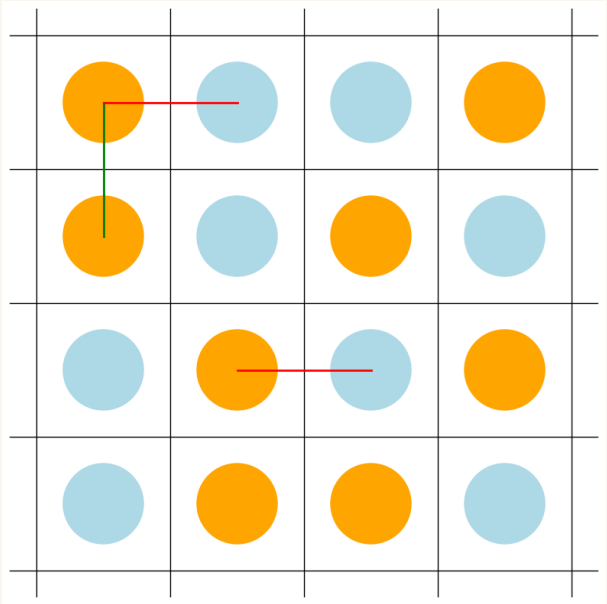
```
for (i1, j1), (i2, j2) in equals_constraints:
    solver.add(cells[i1][j1] == cells[i2][j2])

for (i1, j1), (i2, j2) in opposite_constraints:
    solver.add(cells[i1][j1] != cells[i2][j2])
```

Initial placements

```
if initial_suns:
    for i, j in initial_suns:
        solver.add(cells[i][j] == True)
if initial_moons:
    for i, j in initial_moons:
        solver.add(cells[i][j] == False)
```

Visualisation

We use matplotlib: orange circles for suns, light-blue circles for moons, green lines for equal constraints, red lines for opposite constraints, with reduced opacity for cells filled by the solver versus the initial placements.



Complete implementation

```
from z3 import *
import matplotlib.pyplot as plt

def solve_tango(size, equals_constraints, opposite_constraints,
                initial_suns=None, initial_moons=None):
    solver = Solver()
    cells = [[Bool(f"cell_{i}_{j}") for j in range(size)]
             for i in range(size)]

    def count_suns(cell_list):
        return Sum([If(cell, 1, 0) for cell in cell_list])

    for i in range(size):
        for j in range(size - 2):
            h = [cells[i][j], cells[i][j+1], cells[i][j+2]]
            v = [cells[j][i], cells[j+1][i], cells[j+2][i]]
            solver.add(count_suns(h) <= 2)
            solver.add(count_suns(h) >= 1)
            solver.add(count_suns(v) <= 2)
            solver.add(count_suns(v) >= 1)

    half = size // 2
    for i in range(size):
        solver.add(count_suns(cells[i]) == half)
        solver.add(count_suns([cells[j][i] for j in range(size)]) == half)

    for (i1, j1), (i2, j2) in equals_constraints:
```

```
            solver.add(cells[i1][j1] == cells[i2][j2])
        for (i1, j1), (i2, j2) in opposite_constraints:
            solver.add(cells[i1][j1] != cells[i2][j2])

        if initial_suns:
            for i, j in initial_suns:
                solver.add(cells[i][j] == True)
        if initial_moons:
            for i, j in initial_moons:
                solver.add(cells[i][j] == False)

        if solver.check() == sat:
            m = solver.model()
            return [[m.evaluate(cells[i][j]) for j in range(size)]
                    for i in range(size)]
        return None
```

Conclusion

Tango is a clean example of how constraint-satisfaction problems yield to a few lines of Z3. Logical constraints and visual representation together make it both a good programming exercise and an engaging puzzle. The solver above handles puzzles of any size, including initial placements, and is therefore a versatile tool for both solving existing puzzles and creating new ones.