Z3 PYTHON API

Advanced Verification Techniques - fall semester

Authors
Bálint Takáts - PUWI1T
Gergő Mészáros - OKU4CN

Contents

1 Introduction

This document was created for our Advanced Verification Techniques course. The goal was to create a User Propagator. Thus, in this document, we dive into the Z3 framework, a powerful tool designed for theorem proving and solving complex constraint satisfaction problems. Our main objective was to get a hands-on understanding of how Z3 operates and apply it across various algorithmic scenarios.

We began our exploration with experiments involving different algorithms within the Z3 framework. This approach was instrumental in showcasing the versatility and robustness of Z3 in different contexts. This part was also helpful for us to understand the framework and to gain some experience before creating the User Propagator.

After these initial experiments, we moved on to the real focus of this document: exploring the User Propagator functionality in Z3. This part of Z3 allows for a more advanced level of problem-solving by enabling custom propagation strategies. It represented a significant step up from Z3 and opened up new possibilities for tackling even more complex problems such as the N-Queens challenge and Sudoku puzzles.

To make the creation of User Propagators easier, we also published our Python code on GitHub.

2 Exploration using graphs

2.1 Peterson 3-Coloring Graph

This algorithm challenged us to colour the vertices of the Peterson graph with three colours such that no two adjacent vertices had the same colour. It's a well-known problem in graph theory and provides a great way to test Z3's graph colouring and constraint satisfaction capabilities.

Let $G = (V, E)$ be the Peterson graph, where V is the set of vertices and E is the set of edges. Let $C = \{1, 2, 3\}$ be the set of colors. Define a function $f : V \to C$ which assigns a colour to each vertex.

The Peterson 3-colouring problem can be formulated as:

$$\forall v \in V, f(v) \in C,$$
$$\forall (u, v) \in E, f(u) \neq f(v).$$

Where:

- $\forall v \in V, f(v) \in C$ states that every vertex v in the graph G must be assigned a color from the set C.
- $\forall (u, v) \in E, f(u) \neq f(v)$ ensures that for every edge (u, v) in G, the vertices u and v must have different colors.

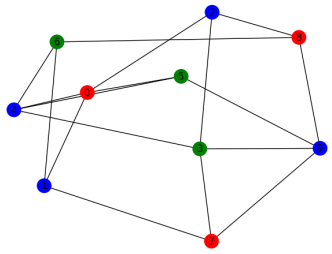

Figure 1: Peterson 3-Coloring Graph Solution

2.2 Hamiltonian Cycle for Dodecahedral Graph

This task involved finding a cycle that visits each vertex of a Dodecahedral graph exactly once. This is another more complex task that the Z3 can perform in terms of cycle recognition of graphs.

Let $G = (V, E)$ be a graph, where V is the set of vertices and E is the set of edges. The Hamiltonian cycle problem for the graph G can be formulated as finding a sequence of vertices $(v_1, v_2, \ldots, v_n)$ such that:

1. $v_1, v_2, \ldots, v_n$ are distinct vertices of V.
2. $(v_i, v_{i+1}) \in E$ for all $i \in \{1, \ldots, n-1\}$.
3. $(v_n, v_1) \in E$.

This sequence represents a cycle that visits each vertex exactly once and returns to the starting vertex.

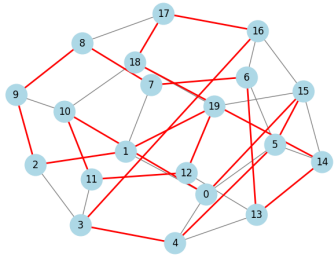

Figure 2: Finding Hamilton cycle in graphs.

3 N-Queen using User Propagators

After concluding our exploration of the basic Z3 API, we began experimenting with User Propagators. This endeavour was incredibly challenging due to our unfamiliarity with Z3. Furthermore, a prominent red banner on the documentation page cautioned us with the warning, *"Program user propagators at your own risk."* [1].

We first started by examining the provided example code. Although the problem presented is highly complex and not yet fully comprehensible to us, it does offer a comprehensive demonstration of the API. Subsequently, we embarked on a search for additional examples to gain further insights. We discovered that there is scarcely any code available on the internet with regards to how User Propagators work in Z3. There are a few examples in C++ that we used for inspiration [2]. We then decided to implement the N-Queens problem as we fully understood it. Based on Bjørner et al.'s work, we believed it would offer a performance boost over the naive implementation [3]. Furthermore, this problem is already included in the Z3 tutorial [4].

The following logic formula describes how one can solve the N-Queens problem without User Propagation:

Let $Q = \{Q_1, Q_2, \ldots, Q_N\}$ be a set of integer variables representing the queens.

Subject to:

$\forall i \in \{1, \ldots, N\}, 1 \leq Q_i \leq N$

$\text{Distinct}(Q_1, Q_2, \ldots, Q_N)$

$\forall i, j \in \{1, \ldots, N\}, i \neq j \implies (Q_i - Q_j \neq i - j) \wedge (Q_i - Q_j \neq j - i)$

Solve for Q.

Implementing the User Propagator posed numerous challenges, mainly due to the lack of documentation. After extensive trial and error, we developed our solution. Firstly, we opted to use *BitVec* instead of the *Int* sort, as the *fixed* function callback does not trigger when an *Int* value is fixed. Moreover, it was crucial to register the variables with the Propagator, as it does not monitor changes to a variable unless registered. Another vital step involved using *BitVecVal* for constants. In the *check_conflict* function, we needed to extract the column and row positions of the current queen and compare them with all other queens in fixed positions. To accomplish this, we had to convert the values fixed to the queens into Python integers, which is not feasible with *BitVec*. After completing the variable declarations, we added constraints to ensure that the queen variables are distinct. Although this could have been integrated into the Propagator logic, we chose not to do so.

We modified the Propagator in a few places compared to the tutorial: We registered the queens so that they are being tracked, rewrote the *fixed* function and implemented the checking logic in *check_conflict*.

The check conflict logic is described in the following pseudo code:

Algorithm 1 Check Conflict Function

```
1:  function CheckConflict(current_queen, queen_row)
2:      queen_column ← queen_to_col[current_queen]
3:      queen_row ← queen_row
4:      for all fixed_queen in fixed_le do
5:          other_queen ← fixed_queen[0]
6:          other_queen_column ← queen_to_col[other_queen]
7:          other_queen_row ← fixed_queen[1])
8:          diff_row ← abs(other_queen_row − queen_row)
9:          diff_col ← abs(other_queen_column − queen_column)
10:         if diff_col = diff_row then
11:             conflict([current_queen, other_queen])
12:             return True
13:         end if
14:     end for
15:     return False
16: end function
```

Finally, Figure 3 illustrates our solution to the 8-Queen puzzle. Furthermore, we also compared how it fares against the previously described solution. The results can be seen on Figure 4.

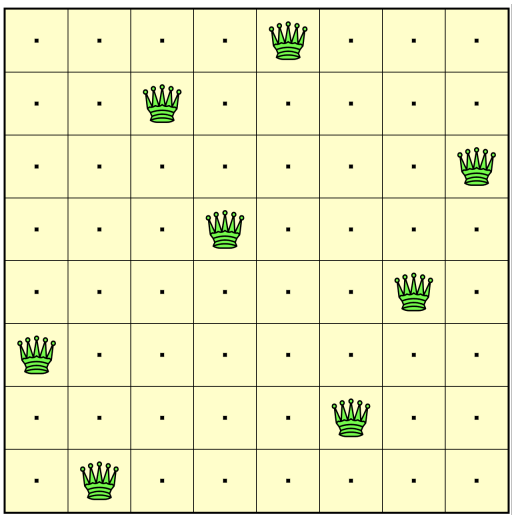

Figure 3: Our solution to the 8 Queen puzzle.

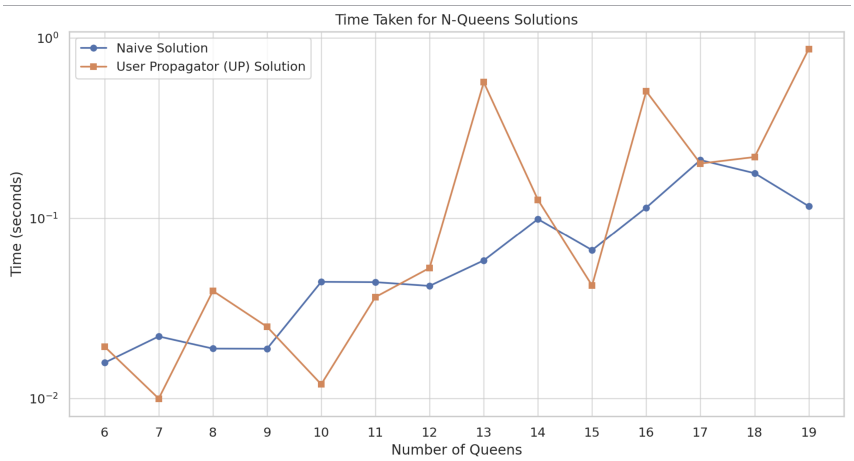

Figure 4: Comparison of the naive solution with User Propagation.

4 Sudoku Solving Using User Propagators

Following our exploration of the Z3 API and successful implementation of the N-Queens problem using User Propagators, we shifted our focus to applying similar techniques to solve Sudoku puzzles. The complexity of User Propagators, combined with limited documentation, presented a challenging yet intriguing problem. We drew inspiration from a few C++ examples [2] and our understanding of the N-Queens problem [3] to embark on this endeavor.

The Sudoku problem is fundamentally different from N-Queens but shares similarities in constraint satisfaction. A standard Sudoku puzzle consists of a 9x9 grid, divided into 3x3 subgrids. The objective is to fill the grid with digits from 1 to 9, such that each row, column, and subgrid contains distinct values.

We utilized *BitVec* instead of *Int* for cell values, as the *fixed* callback function in User Propagators does not trigger with *Int* values. Registering variables with the Propagator was essential for monitoring their changes. The *BitVecVal* was used for constants, and a *check_conflict* function was implemented to compare the value of the current cell with all other cells in fixed positions, ensuring uniqueness in rows, columns, and subgrids.

Here is the logical formula to solve Sudoku without User Propagation:

Let $S = \{S_{1,1}, S_{1,2}, \ldots, S_{9,9}\}$ be a set of integer variables representing the cells of the Sudoku grid.

Subject to:

$\forall i, j \in \{1, \ldots, 9\}, 1 \leq S_{i,j} \leq 9$

$\forall i \in \{1, \ldots, 9\}, \text{Distinct}(S_{i,1}, S_{i,2}, \ldots, S_{i,9})$

$\forall j \in \{1, \ldots, 9\}, \text{Distinct}(S_{1,j}, S_{2,j}, \ldots, S_{9,j})$

$\forall p, q \in \{0, 3, 6\}, \text{Distinct}(\{S_{p+r,q+s} \mid r, s \in \{0, 1, 2\}\})$

Solve for S.

The implementation of the User Propagator for Sudoku was as follows:

- Define a *BitVec* for each cell in the Sudoku grid.
- Implement the *check_conflict* function to detect conflicts in rows, columns, and subgrids.
- Register each cell with the Propagator to track its value.
- Introduce constraints for distinct values in rows, columns, and subgrids.

4.1 Check Conflict Logic

The *check_conflict* function plays a crucial role in identifying violations of Sudoku rules. Upon fixing a cell's value, this function checks for any conflicts with previously fixed cells. If a conflict is detected, the *conflict* method of the User Propagator is invoked, indicating the incompatibility of the fixed values.

4.2 Visualization

To aid in debugging and understanding the solver's progression, a *print_grid* function was implemented. This function visualizes the current state of the Sudoku grid, showing the fixed values and empty cells.

```
---------------------
5 3 0 | 0 7 0 | 0 0 0
6 0 0 | 1 9 5 | 0 0 0
0 9 8 | 0 0 0 | 0 6 0
---------------------
8 0 0 | 0 6 0 | 0 0 3
4 0 0 | 8 0 3 | 0 0 1
7 0 0 | 0 2 0 | 0 0 6
---------------------
0 6 0 | 0 0 0 | 2 8 0
0 0 0 | 4 1 9 | 0 0 5
0 0 0 | 0 8 0 | 0 7 9
---------------------
```

Figure 5: Sudoku Puzzle

```
---------------------
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
---------------------
```

Figure 6: Sudoku Solution

Figure 5 and Figure 6 provide an illustration of our solution to a Sudoku puzzle.

In conclusion, the application of User Propagators to the Sudoku problem not only deepened our understanding of Z3's capabilities but also demonstrated a powerful approach to solving complex constraint satisfaction problems.

References

[1] "User propagators," https://microsoft.github.io/z3guide/programming/Example%20Programs/User%20Propagator, 2023, online Z3 Guide.

[2] dwightguth, "Optimizing an smt query about partial orders," https://github.com/Z3Prover/z3/discussions/6173, 2022, discussion 6173 on Z3Prover GitHub repository.

[3] C. Eisenhofer, "Z3: User Propagator," https://github.com/Z3Prover/z3/blob/master/examples/userPropagator/example.pdf, 2023, experience report, in joint work with Nikolaj Bjørner.

[4] "Z3: z3py namespace reference," https://z3prover.github.io/api/html/namespacez3py.html, 2023.