1 SAT Approach in Schur Number Problem

In this week's report, we explored the SAT-based methods used to address the Schur number problem. Building on our previous findings, where we stated, *"We would go back to understand the SOTA paper on how they developed Schur number 5"*, we have now analyzed the SAT-solving approach in depth. Below, we summarize our findings.

1.1 How SAT Solvers Work

SAT solvers are tools designed to determine the satisfiability of a propositional logic formula in Conjunctive Normal Form (CNF). For the Schur number problem, the goal is to encode the coloring constraints into a propositional formula and verify if a valid coloring exists for a given number of colors and integers.

In the implementation, Z3 library was used, a high-performance theorem prover and SAT solver. Z3 provides a Python API, which allowed us to efficiently encode constraints, solve problems, and analyze results. The entire implementation, including the code discussed in this report, is available in our GitHub repository. This implementation facilitated the generation of CNF formulas, applying constraints, and optimizing the search space.

1.2 Constraints Used in the SAT Encoding

To ensure the encoding correctly represents the problem, several key constraints are applied:

1. **Each number must have exactly one color:** Each integer j $(1 \leq j \leq n)$ is assigned one and only one color among k colors. This constraint is encoded as:

$$\bigvee_{i=1}^{k} v_{ij}$$

$$\neg(v_{i1} \wedge v_{i2} \wedge \ldots \wedge v_{ik})$$

   The equivalent Python code is:

```
    # At least one color
    s.add(Or(colors[i]))
    # At most one color
    s.add(Not(And(colors[i])))
```

2. **No monochromatic solution to $a + b = c$:** For any triple (a, b, c) satisfying $a + b = c$, the numbers a, b, and c cannot all have the same color. This is expressed as:

$$\neg(v_{ia} \wedge v_{ib} \wedge v_{ic}), \forall i \in [1, k]$$

   The equivalent Python code is:

```
    bluefor a bluein bluerange(1, n+1):
        bluefor b bluein bluerange(a, n+1):
            c = a + b
            blueif c <= n:
                bluefor i bluein bluerange(k):
                    s.add(Not(And(colors[a-1][i], colors
                        [b-1][i], colors[c-1][i])))
```

3. **Symmetry breaking:** To reduce redundant searches caused by color permutation, additional constraints are imposed, such as assigning specific colors to initial numbers. For example:

$$v_{11} = 1, \quad v_{22} = 1$$

   The equivalent Python code is:

```
    s.add(colors[1-1][0]) # Assign color 0 to number 1
    s.add(colors[2-1][1]) # Assign color 1 to number 2
```

4. **Additional heuristics:** Custom heuristics prevent invalid patterns, such as two consecutive numbers having the same color:

$$\neg(v_{i1} \wedge v_{i2}), \quad \forall i \in [1, k]$$

   Similarly, for preventing overlap in specific patterns:

$$\neg(v_{i1} \wedge v_{i3}), \quad \forall i \in [1, k]$$

   The equivalent Python code is:

```
    bluefor i bluein bluerange(n-1):
        s.add(Not(And(colors[i][0], colors[i+1][0])))
    # Prevent overlap in specific patterns
    bluefor i bluein bluerange(n-2):
        s.add(Not(And(colors[i][1], colors[i+2][1])))
```

The constraints work together as follows:

- **Organizing the search space:** Constraints (1) ensure each number has one and only one color.
- **Avoiding conflicts:** Constraint (2) prevents any monochromatic solution to $a + b = c$, ensuring the validity of the coloring.
- **Speeding up the computation:** Constraint (3) reduces the search space by breaking symmetries, and Constraint (4) introduces additional optimizations for efficient solving.

1.3 Cube-and-Conquer Method

The Cube-and-Conquer (CC) method enhances SAT solvers by splitting the problem into smaller subproblems (”cubes”) that can be solved independently. This method consists of two phases:

1. **Cube Phase:** The problem is divided into millions or billions of smaller subproblems using decision heuristics. Each subproblem represents a partial assignment of variables.
2. **Conquer Phase:** Each subproblem (cube) is solved using a Conflict-Driven Clause Learning (CDCL) solver. Solutions from these subproblems are combined to form the final result.

This hybrid approach combines the strengths of look-ahead solvers for splitting and CDCL solvers for solving, enabling efficient parallel processing.

1.4 Proofs Used in the Paper

Three main types of proofs to ensure correctness and completeness were developed and verified:

1. **Re-encoding Proof:** This proof validated the correctness of the symmetry-breaking techniques used to reduce the search space. It ensured that the modified formulas preserved the logical equivalence of the original problem.
2. **Implication Proof:** This proof demonstrated that $S(5) > 160$ is unsatisfiable by partitioning the problem into smaller cubes and proving unsatisfiability for each subproblem. The total size of this proof exceeded two petabytes.
3. **Tautology Proof:** This proof confirmed that the cubes covered the entire search space, ensuring that no valid solution was overlooked.

These proofs were validated using the ACL2 theorem prover, providing high confidence in the correctness of the results.

1.5 Results of the SAT Approach and Cube-and-Conquer

The SAT-based Cube-and-Conquer method produced the following results:

- Confirmed that $S(5) = 160$.
- Partitioned the search space into over 10 million subproblems, solved efficiently in parallel.
- Generated a proof of unsatisfiability for $S(5) > 160$, with a total proof size exceeding two petabytes.

- Achieved a runtime of just three days on a high-performance computing cluster (equivalent to 14 CPU years).

These findings highlight the power and scalability of the SAT-based Cube-and-Conquer method in addressing complex combinatorial problems like the Schur number problem.

References

Heule, M. J. H. (2017). *Schur Number Five.* arXiv. https://arxiv.org/abs/1711.08076