COSC 76: PA5 Report

Wesley Tan

November 2024

1 Introduction

The implementation of the SAT (Boolean Satisfiability) solver leverages propositional logic to solve Sudoku puzzles by converting them into `.cnf`. Two algorithms were implemented to solve the SAT problem: GSAT and WalkSAT.

1.1 Problem Representation

The Sudoku problem was encoded as follows:

- Variables are represented as 3-digit numbers rcv, where:
  - r: Row (1-9)
  - c: Column (1-9)
  - v: Value (1-9)
- Each variable represents whether a specific value is placed in a specific cell.
- The CNF formula includes clauses for:
  - **Cell Constraints:** Each cell must have exactly one value.
  - **Row, Column, and Box Uniqueness:** Each number appears exactly once in each row, column, and 3x3 box.
  - **Initial Values:** Known values are encoded as unit clauses.

1.2 Implementation

The SAT solver processes input CNF files and employs optimizations to improve performance. The implementation is structured into the following components:

1.2.1 Input Processing

- **CNF Parsing:** The `load_cnf` function reads CNF files and builds:
  - **Variables:** Stored as integers to represent literals.
  - **Clauses:** A list of lists, where each clause is a disjunction of literals.
- **Variable-to-Clause Indexing:** Each variable is mapped to the clauses it appears in, reducing redundant evaluations during flips.

1.2.2 Clause Evaluation

- Each clause is evaluated as satisfied if at least one literal in it is true.
- Functions are provided to count satisfied clauses and identify unsatisfied clauses efficiently.

1.2.3 WalkSAT

The WalkSAT algorithm balances random exploration and greedy optimization:

1. Start with a random truth assignment.
2. Repeatedly select an unsatisfied clause and:
   - Flip a random variable in the clause with probability p.
   - Otherwise, flip the variable that maximizes the number of satisfied clauses (using make-break scores).
3. Track the best solution across iterations.

1.2.4 GSAT

The GSAT algorithm focuses on global optimization:

1. Start with a random truth assignment.
2. Evaluate all variables and flip the one that maximizes the number of satisfied clauses.
3. Occasionally perform random flips to escape local optima.

1.2.5 Validation and Output

- **Solution Validation:** Ensures that all clauses are satisfied in the final assignment.
- **Solution Output:** Writes the assignment to a solution file, listing all satisfied variables.

1.3 Optimizations

Key optimizations implemented include:

- Speeds up clause evaluation by maintaining direct references.
- **Make-Break Calculation:** Reduces redundant evaluations during variable flips.
- **Best Solution Tracking:** Tracks and updates the best assignment across all iterations.

2 Evaluation

2.1 WalkSAT Algorithm

The WalkSAT algorithm balances random exploration and greedy optimization:

1. Start with a random truth assignment.
2. Repeatedly select an unsatisfied clause and:
    - Flip a random variable in the clause with probability p.
    - Otherwise, flip the variable that maximizes the number of satisfied clauses (using make-break scores).
3. Track the best solution across iterations.

```
def walksat(self, max_tries=100, max_flips=100000, p=0.3) -> bool:
    for try_num in range(max_tries):
        assignment = {var: random.choice([True, False]) for var in self.variables}
        current_satisfied = self.count_satisfied_clauses(assignment)
        for flip in range(max_flips):
            if current_satisfied == len(self.clauses):
                self.assignment = assignment
                return True
            unsatisfied = self.get_unsatisfied_clauses(assignment)
            clause = self.clauses[random.choice(unsatisfied)]
            if random.random() < p:
                var = abs(random.choice(clause))
            else:
                best_score = float('-inf')
                candidates = []
                for lit in clause:
                    var = abs(lit)
                    make, break_count = self.calculate_make_break(var, assignment)
                    score = make - break_count
                    if score > best_score:
                        best_score = score
                        candidates = [var]
                    elif score == best_score:
                        candidates.append(var)
                var = random.choice(candidates)
            assignment[var] = not assignment[var]
            current_satisfied = self.count_satisfied_clauses(assignment)
    return False
```

2.2 GSAT Algorithm

The GSAT algorithm focuses on global optimization:

1. Start with a random truth assignment.
2. Evaluate all variables and flip the one that maximizes the number of satisfied clauses.
3. Occasionally perform random flips to escape local optima.

```
def gsat(self, max_tries=100, max_flips=1000, h=0.3) -> bool:
    for try_num in range(max_tries):
        assignment = {var: random.choice([True, False]) for var in self.variables}
        current_satisfied = self.count_satisfied_clauses(assignment)
        for flip in range(max_flips):
            if current_satisfied == len(self.clauses):
                self.assignment = assignment
                return True
            if random.random() < h:
                var = random.choice(list(self.variables))
            else:
                best_score = -1
                best_vars = []
                for var in self.variables:
                    test_assignment = assignment.copy()
                    test_assignment[var] = not test_assignment[var]
                    score = self.count_satisfied_clauses(test_assignment)
                    if score > best_score:
                        best_score = score
                        best_vars = [var]
                    elif score == best_score:
                        best_vars.append(var)
                var = random.choice(best_vars)
            assignment[var] = not assignment[var]
            current_satisfied = self.count_satisfied_clauses(assignment)
    return False
```

2.3 Effectiveness

The implemented algorithms successfully solved various SAT problems, including Sudoku puzzles. WalkSAT and GSAT demonstrated distinct strengths in different scenarios.

2.3.1 Results

- Basic Sudoku puzzles were solved quickly with both algorithms.
- Harder puzzles required up to 20,000 flips but were solvable with WalkSAT.

2.3.2 Performance Metrics

- **WalkSAT:** Faster convergence, more reliable solutions for complex puzzles.
- **GSAT:** Higher computational cost due to complete neighborhood evaluation.

2.3.3 Limitations

- Solution time scales with problem size and complexity.
- Random initialization introduces variability in convergence speed.
- GSAT's exhaustive evaluation becomes inefficient for large problems (could not run rows_and_cols.cnf).

2.4 Example Outputs

2.4.1 Basic Puzzle (puzzle2)

```
Solution found in 0.003 seconds
Nodes explored: 25
```

2.4.2 Bonus Puzzle (puzzle_bonus)

```
Solution found in 20.5 seconds
Nodes explored: 25,000
```

In summary:

- **Basic Puzzle:** Solved in the first attempt using both algorithms.
- **Bonus Puzzle:** Solved after approximately 20,000 flips using WalkSAT.
- **Performance Comparison:** WalkSAT consistently outperformed GSAT in solving larger problems. Often, GSAT took a really long time to solve the large problems.

3 Bonus

Included implementations for `nqueens` and `map_coloring`.