UFMG

PROJECT

Conflict-Driven Kotlin Learning

January 28, 2025

Student:
Alan Cabral Trindade Prado
2020006345

Tutor:
Dr. Haniel Barbosa

Course:
Theory and Practice of SMT Solving

1 Introduction

Conflict-Driven Kotlin Learning is a project inspired by the book *Atomic Kotlin* by Bruce Eckel and Svetlana Isakova. A project implementing a Conflict-Driven Clause Learning (CDCL) solver was chosen to explore the practical application of the book's concepts. Kotlin's expressiveness makes it possible to create a highly readable and approachable implementation of a CDCL solver, allowing anyone with basic programming knowledge to explore and understand it easily.

The source code for this project can be found at: `https://github.com/alanctprado/cdkl`

2 Project Goals

This project aims to have the following characteristics:

1. **Ease of understanding and use:** The codebase should enable individuals with no prior SAT-solving experience to understand both foundational and advanced concepts. The code should be self-explanatory, minimizing the need for external documentation to understand the purpose of classes or functions.
2. **Theoretical documentation:** Provide comprehensive documentation covering the theoretical foundations, including method complexity, proofs of termination, and other relevant concepts. This should be part of the code, but kept concise to preserve readability.
3. **Encourage extensibility:** Offer a modular structure that allows developers to experiment with different heuristics in CDCL solving. Automated tests and benchmarks should assist in verifying implementation and enable performance comparisons.

2.1 Current Features

All that was required in the project specification:

- Support for the DIMACS CNF input format and output in the SAT competition format.
- Basic **CDCL solver** with:
  - Clause learning.
  - Non-chronological backjumping.
- Unit propagation using **two watched literals**.

2.2 Future Plans

- Enhance the DPLL SAT solver.
- Add comprehensive theoretical documentation for all implemented features, aligning with the project's second goal.
- Implement additional CDCL features, including:
  - First UIPs (Unique Implication Points).
  - VSIDS (Variable State Independent Decaying Sum) heuristic for literal decision-making.
- Improve the overall project structure and build system.

3 How to Run

For the best experience, it's recommended to use IntelliJ IDEA to explore and execute the project. The IDE provides an intuitive interface and excellent support for Kotlin projects.

Alternatively, you can:

3.1 Execute Using Gradle

To run this project, you can use the Gradle Wrapper (`gradlew`). The Gradle Wrapper is a script that comes with the project and ensures you use the correct Gradle version, without needing to install Gradle separately on your system.

To execute the program, use the command `./gradlew run` (or `gradlew run` on Windows). The very first time you run this command, Gradle will also need to build the project, which might take a bit longer. Alternatively, you can explicitly build the project first using `./gradlew build` (or `gradlew build` on Windows).

This program requires a DIMACS file as input. To provide the file location, pass it as an argument using the `--args` option:

```
./gradlew run --args="/path/to/your/dimacs/file.cnf"
```

Replace `/path/to/your/dimacs/file.cnf` with the actual path to your DIMACS file.

4 Project Structure

4.1 Main

The `CdklCommand` class serves as the entry point for executing the Conflict-Driven Clause Learning (CDCL) algorithm. It processes a SAT problem provided in a DIMACS-formatted file, parses it into an internal representation using a Parser object, and validates the input format. The parsed problem is then passed to a CDCL solver, which determines whether the problem is satisfiable or unsatisfiable.

The function outputs the result in a standard format. If the problem is satisfiable, it prints "s SATISFIABLE" followed by the satisfying model in DIMACS format, where positive integers represent true variables, and negative integers represent false variables. If unsatisfiable, it prints "s UNSATISFIABLE". This structure ensures compatibility with SAT-solving competition format.

4.2 CDCL

The CDCL class implements the Conflict-Driven Clause Learning (CDCL) algorithm for solving SAT problems. It operates on a given formula, managing the decision-making, conflict resolution, and clause learning required for efficient SAT solving. The primary workflow is encapsulated in the solve() method, which repeatedly performs unit propagation, checks for conflicts, makes decisions, and resolves conflicts until a solution is found or the formula is proven unsatisfiable.

Key components include an implication graph for tracking assignments and conflicts, a two-literal watching scheme for efficient clause management, and a decision-making strategy for selecting unassigned literals. Each of these components is implemented in its own file. Conflicts are resolved via conflict analysis, clause learning, and non-chronological backtracking (backjumping). If a conflict cannot be resolved at the initial decision level, the problem is deemed unsatisfiable. Otherwise, the process continues until the model is complete, at which point the problem is determined to be satisfiable. The final model can be retrieved using the `getModel` method.

4.3 Conflict

The ImplicationGraph class manages the relationships between literals, decisions, and implications during the execution of the Conflict-Driven Clause Learning (CDCL) algorithm. It is used to track the reasoning process, identify conflicts, and support conflict resolution through clause learning and backjumping. Key operations include:

- **Adding Nodes:** Decisions: The `addDecision` method adds a decision node to the graph, representing a variable assignment at a specific decision level. Implications: The `addImplication` method adds an implication node derived from unit propagation, linking it to its logical predecessors in the graph.
- **Conflict Analysis:** The `analyzeConflict` method processes a conflict clause to determine a learned clause and a backjump level. It identifies the nodes responsible for the conflict, resolves the conflict based on a configurable strategy, and derives a clause to prevent similar conflicts.
- **Backjumping:** The `backjump` method removes nodes above a specified decision level, effectively undoing assignments and implications made after that level.
- Validation: Several utility methods ensure the integrity of the graph, such as checking for valid conflicts, ensuring literals are correctly added, and verifying decision levels during backjumping.

4.4 Watcher

The Watcher class hierarchy manages clauses during the execution of the Conflict-Driven Clause Learning (CDCL) algorithm, enabling efficient identification of conflicts and unit clauses. It tracks the state of clauses and their relationships to literals, providing essential functionality for unit propagation.

1. **Abstract Watcher**: The base class defines common functionality, such as maintaining a list of watched clauses and abstract methods for adding clauses or literals, backjumping, and retrieving conflict or unit clauses.
2. **Basic Watcher**: The `BasicWatcher` implementation performs simple clause management, identifying conflicts by checking if all literals in a clause are falsified. It detects unit clauses by finding clauses where all but one literal are falsified and returns the remaining literal for propagation.
3. **Two-Literal Watcher**: The `TwoLiteralWatcher` extends the base functionality with a more sophisticated approach for efficiently managing clauses. It classifies clauses as:
   - **Satisfied Clauses:** Clauses with at least one true literal.
   - **Unsatisfied Clauses:** Clauses with unassigned or false literals. It tracks two "watched literals" per clause, dynamically updating them during propagation or backjumping. This improves performance by reducing the number of clause checks required during unit propagation.

   Key operations include:

- **Adding Clauses and Literals:** Dynamically updates clause classifications and watched literals as new clauses or literal assignments are added.
- **Backjumping:** Reverts clause states to reflect earlier decision levels during conflict resolution.
- **Conflict and Unit Detection:** Efficiently identifies conflict clauses (fully falsified) and unit clauses (all but one literal falsified).

4.5 Model

The `Model` object manages variable assignments during the execution of the CDCL algorithm. It maintains a mapping between literals and their decision levels, enabling efficient access and updates to the current state of assignments. The state of an object in Kotlin is shared between every instance (i.e. it is a singleton). Key functionalities include:

- **Adding Assignments:** The `addLiteral` method assigns a literal at a specific decision level. It ensures, in safe mode, that the literal or its opposite has not already been assigned.
- **Backjumping:** The `backjump` method removes all assignments made at levels higher than a specified decision level, effectively undoing recent decisions.
- **Querying:** Provides utility methods to:
  - Check if a literal or its variable has been assigned (`hasLiteral`, `hasVariable`).
  - Retrieve the decision level of an assignment (`levelOf`).
  - Get the list of assigned literals (`assignment`).

4.6 Decider

The `Decider` abstract class is responsible for selecting unassigned literals during the CDCL algorithm's decision phase. It defines the abstract method decide() for determining the next literal to assign. The `BasicDecider` implementation selects an unassigned literal from the formula's clauses. It checks the set of literals in the formula and returns the first unassigned literal. If all variables are assigned, it throws an exception. This approach ensures that a new decision can be made unless all variables have been assigned.

4.7 Boolean

Provides several data structures to simplify and improve the readability of the solver.

1. Enums:
   - `SolverResult`: Represents the result of the SAT solver (Satisfiable, Unsatisfiable).
   - `Polarity`: Specifies the polarity of a literal (Positive, Negative).
   - `Value`: Denotes the possible values of a literal (True, False, Unknown).
2. Classes:
   - `Variable`: Represents a variable by its index.
   - `Literal`: Combines a Variable and Polarity to represent a literal. Supports polarity checks and the opposite() method to return the opposite literal.
   - `Clause`: A collection of literals. It has utility methods for checking if it's empty or unit, removing literals, and accessing its variables.
   - `Formula`: A list of Clause objects representing the SAT problem. It provides methods for clause removal, adding new clauses, and extracting literals and variables.

4.8 Parser

The `Parser` class is designed to read and parse DIMACS formatted files that represent SAT problems. Upon initialization, the class checks that the specified file exists. The method `validateDimacsFormat` is responsible for verifying the correctness of the file's format. It ensures that the file contains exactly one problem line and that the clauses and variables are structured correctly according to the DIMACS standard.

The main parsing functionality is provided by the `parseProblem` method, which processes the file, extracting and converting the clauses into a `Formula` instance. Each clause is represented by a `Clause` containing `Literal` instances, which correspond to the variables and their respective polarities in the SAT problem.

5 Experiments

A comprehensive evaluation using the pj1-tests suite found no evidence of soundness or completeness issues. A one-minute time limit was imposed on the execution of each instance. The results are presented in the tables below. The complete results can be found in the repository root.

File Name	Variables	Clauses	Solved	Time
block0	3	3	Yes	0m0.597s
cnfgen-php-10-10	100	460	No	1m0.325s
elimredundant	5	8	Yes	0m1.440s
prime121	329	952	Yes	0m3.393s
prime1369	989	2908	Yes	0m58.154s
prime1681	989	2908	No	1m0.348s
prime169	461	1342	Yes	0m11.945s
prime1849	989	2908	No	1m0.325s
prime841	791	2320	Yes	0m21.141s
prime961	791	2320	Yes	0m21.915s
sat10	3	7	Yes	0m1.172s
sat12	3	7	Yes	0m1.604s
sqrt10201	303	841	Yes	0m2.630s
sqrt1042441	659	1877	Yes	0m3.070s
sqrt10609	303	841	Yes	0m2.852s
sqrt11449	303	841	Yes	0m2.964s
uf20-01000	20	91	Yes	0m2.198s
uf20-0100	20	91	Yes	0m1.864s
uf20-0101	20	91	Yes	0m1.828s
uf20-0102	20	91	Yes	0m1.727s
uf20-0103	20	91	Yes	0m1.838s
uf20-0104	20	91	Yes	0m1.960s
uf20-0105	20	91	Yes	0m1.942s
uf20-0106	20	91	Yes	0m2.676s

Table 1: Details of SAT problems: number of variables, clauses, solver results, and solving time.

Due to unresolved compilation errors with the MiniSAT solver (logs in the repository root), I was unable to test its runtime. Given the approaching deadline (in 20 minutes), I am unlikely to resolve this issue in time. I have dedicated significant effort to this project and respectfully request consideration for this unforeseen problem :)

File Name	Variables	Clauses	Solved	Time
add4	60	157	Yes	0m0.691s
add8	149	400	No	1m0.326s
cnfgen-parity-9	36	261	Yes	0m2.148s
cnfgen-peb-pyramid-20	231	232	Yes	0m2.478s
cnfgen-php-5-4	20	45	Yes	0m2.014s
cnfgen-ram-4-3-10	45	330	No	1m0.327s
cnfgen-tseitin-10-4	20	80	Yes	0m3.553s
elimclash	5	8	Yes	0m2.116s
false	0	1	Yes	0m1.919s
full1	1	2	Yes	0m1.933s
full3	3	8	Yes	0m1.963s
full5	5	32	Yes	0m1.943s
full7	7	128	Yes	0m2.175s
ph6	42	133	Yes	0m8.727s
unit7	4	5	Yes	0m1.983s
uuf100-010	100	430	No	1m0.328s
uuf100-0117	100	430	No	1m0.326s
uuf100-0120	100	430	No	1m0.326s
uuf100-012	100	430	No	1m0.333s
uuf100-0130	100	430	No	1m0.349s
uuf100-0147	100	430	No	1m0.329s
uuf100-0151	100	430	Yes	0m51.011s
uuf100-0161	100	430	No	1m0.328s
uuf100-0175	100	430	No	1m0.336s
uuf100-0182	100	430	No	1m0.330s

Table 2: Details of UNSAT problems: number of variables, clauses, solver results, and solving time.