

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных
технологий

КУРСОВОЙ ПРОЕКТ

Sat-solver

по дисциплине «Алгоритмы и структуры данных»

Выполнил
студент гр. 3530901/90002

_____ Бурков Е. В.

Руководитель

_____ Степанов Д. С.

«__» _____ 2020 г.

Санкт-Петербург
2020

**ЗАДАНИЕ
НА ВЫПОЛНЕНИЕ КУРСОВОЙ РАБОТЫ**

студенту группы 3530901/90002

Буркову Егору Владимировичу

1. Тема проекта (работы): создание SAT-солвера с использованием входных файлов в формате DIMACS.

2. Срок сдачи законченной работы 11 декабря.

3. Исходные данные к работе: требования к реализовываемому проекту.

4. Содержание пояснительной записки: (перечень подлежащих разработке вопросов): введение, основная часть (текст программы, описание программы, испытания программы), заключение, список использованных источников.

Дата получения задания: «1» октября 2020 г.

Руководитель

(подпись)

Степанов Д. С.

Задание принял к исполнению

(подпись)

Бурков Е.В.

«1» октября 2020 г.

Оглавление

ВВЕДЕНИЕ.....	4
ОСНОВНЫЕ АЛГОРИТМЫ.....	5
ОПИСАНИЕ ПРЕДЛОЖЕННОГО РЕШЕНИЯ	7
ТЕСТИРОВАНИЕ ПРОГРАММЫ.....	10
СБОР СТАТИСТИКИ	11
ЗАКЛЮЧЕНИЕ.....	13
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	14

ВВЕДЕНИЕ

Цель работы: создать программу решающую задачу выполнимости булевых функций (далее SAT). Протестировать 2 разных алгоритма и сделать выводы. Программу реализовать на языке Java или Kotlin.

SAT-solver принимает на вход множество конъюнктивных нормальных форм объединённых дизъюнкцией и должен выдать ответ: есть ли решение для такого примера или нету.

ОСНОВНЫЕ АЛГОРИТМЫ

Для решения поставленной задачи будут использоваться 2 алгоритма поиска решения с возвратом.

Сущность почти всех алгоритмов такая: мы берём одну из переменных и присваиваем ей значение. Если при выборе одной из переменных получается, что хоть одна КНФ оказывается невыполненной, то возвращаемся и берём другую переменную. В моём проекте используются 2 алгоритма в которых механизмы возврата кардинально отличаются.

Первый алгоритм – **DPLL** (*алгоритм Дэвиса-Патнема-Логемана-Лавленда*). Возврат осуществляется хронологически, то есть если мы взяли переменную $X = \text{true}$, и в результате этого появился конфликт, то мы возвращаемся и устанавливаем $X = \text{false}$. Эффективность алгоритма сильно зависит от выбора следующей переменной. На рисунке 1 наглядно показывается как происходят возвраты.

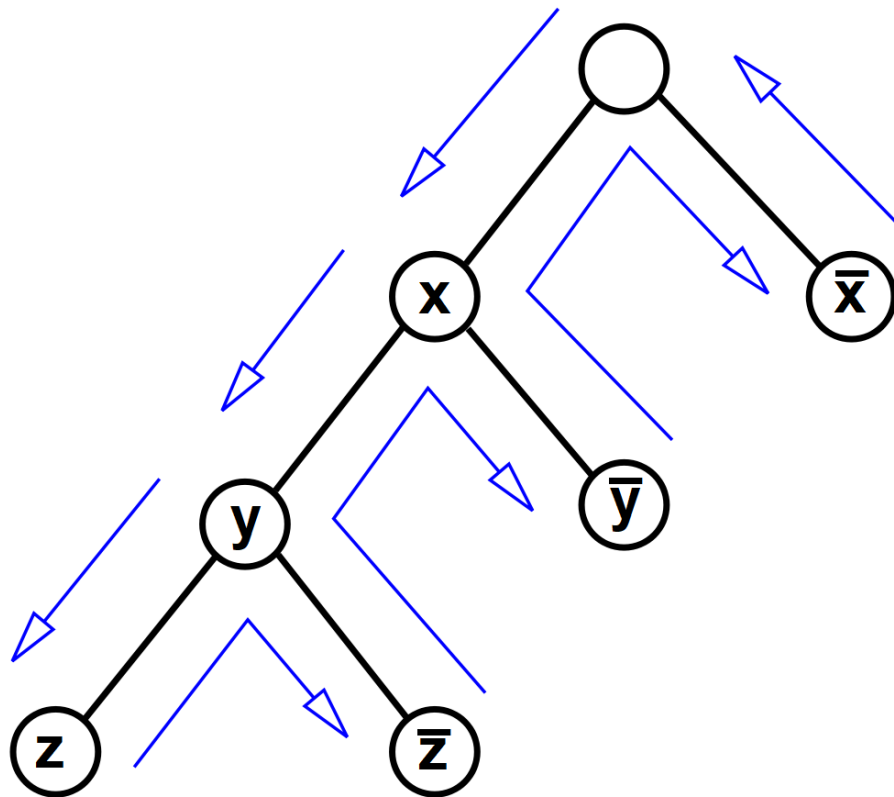


Рис. 1 Хронологический возврат

Второй, более “умный” алгоритм называется **CDCL** (*conflict-driven clause learning* — «управляемое конфликтами обучение дизъюнктам»). Он имеет большие различия с предыдущим, приведём основные:

- в ходе работы запоминаются новые дизъюнкты, полученные в ходе анализа конфликта
- возврат нехронологический
- при присвоении переменной другие переменные так же присваиваются по правилу чистой переменной

Правило чистой переменной – если в дизъюнкте есть 1 не назначенная переменная, а все остальные отрицательные, то можем назначить данную переменную.

Механизм возврата имеет уровни решения, и в общем случае представляет собой импликационный граф, фиксирующий назначения переменными и уровень решения.

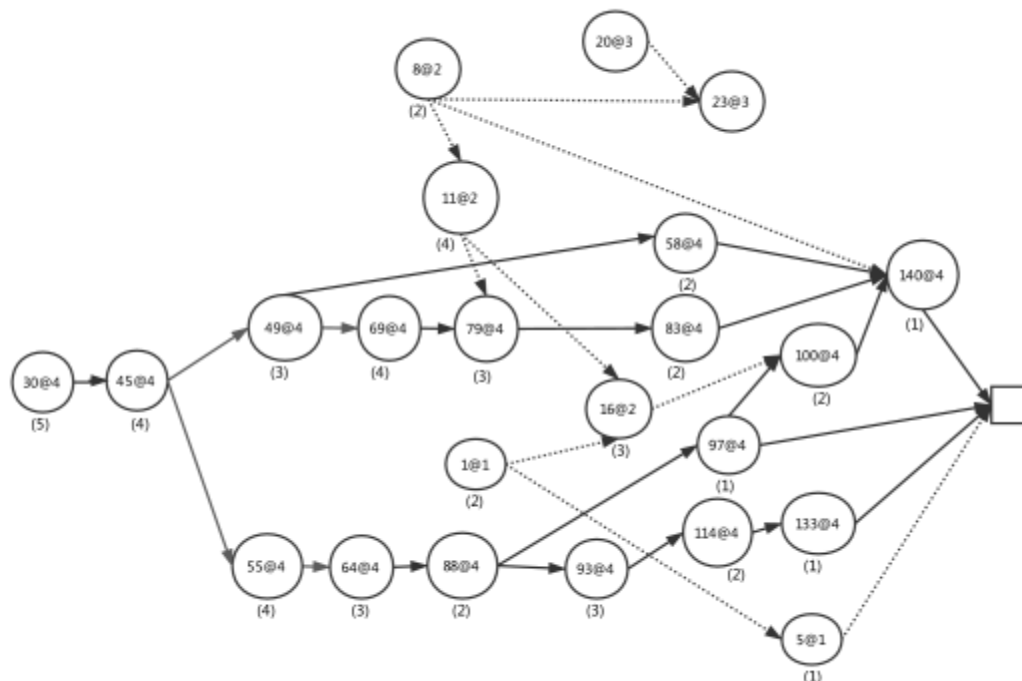


Рис. 2 Один из примеров импликационного графа

ОПИСАНИЕ ПРЕДЛОЖЕННОГО РЕШЕНИЯ

Проект написан на языке Kotlin версии 1.4.0, Java SDK версии 11.0.4.

Репозитории: https://github.com/wooftown/kalgs_sat

В ходе написания программы использовалась парадигма разработки ООП. Пакеты, классы, интерфейсы, перечисления представлены на рисунке 3.

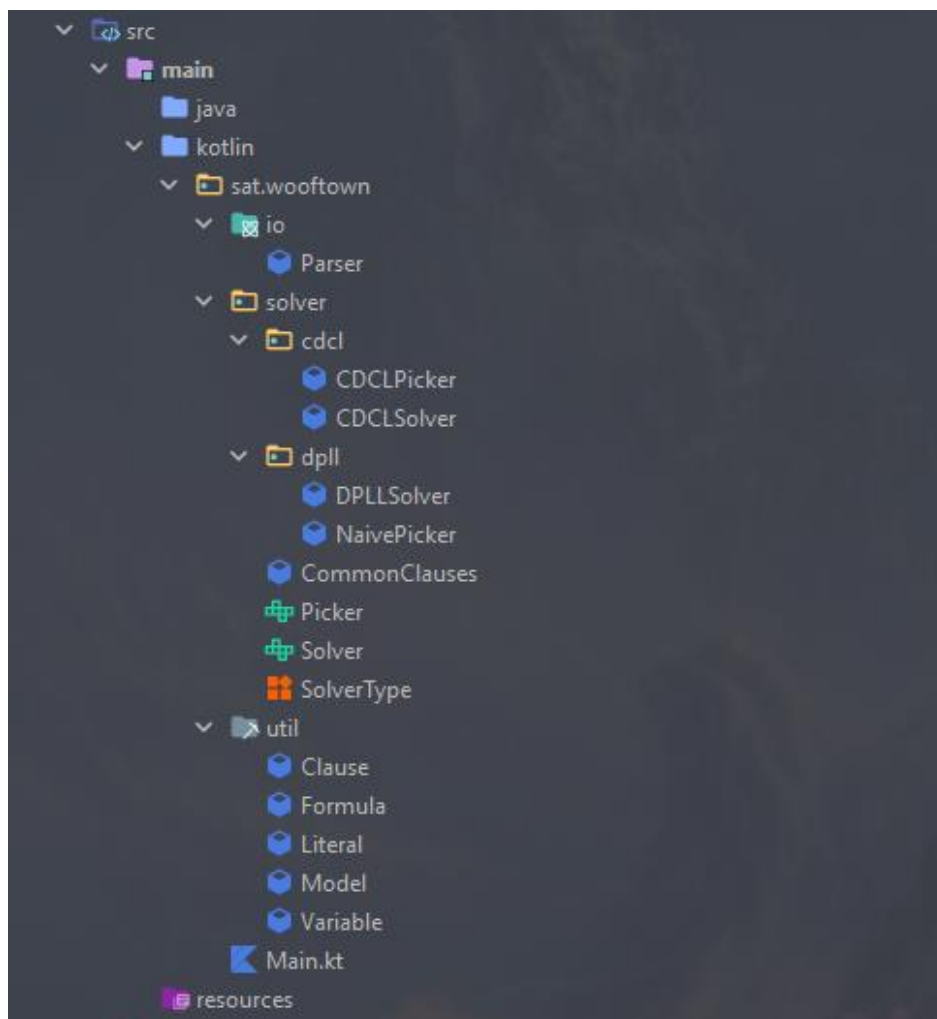


Рис. 3 Основное содержимое программы

Main.kt – главная точка входа в программу (запуск из jar).

Пакет util:

Данная часть проекта отвечает за необходимые структуры данных.

- class Variable – одна переменная. Имеет индекс : 1, 2, 3 и т.д. Не имеет знака.

- `class Literal` – одна переменная, но уже со знаком. Имеет поля для преобразований и получения сведений о ней. Определяется числом.

В данном проекте используется хитрый приём. Для литерала используется одно поле `value`, вместо двух полей (переменная и знак). `Value` определяет и номер переменной и её знак. Если литерал положителен, то `value` чётен, иначе нечётен. Номер переменной получается делением `value` на 2. Данный приём позволил реализовывать некоторые структуры данных необычным, и на мой взгляд весьма удобным способом.

- `class Clause` – данный класс отображает одну строчку из условия к задаче. То есть `clause` является ДНФ.
- `class Formula` – класс который хранит все `clause`. Является КНФ.
- `class Model` – хранение ответа к задаче.

Пакет solver:

Данный пакет отвечает за все алгоритмы решения SAT.

- `interface Solver` – интерфейс решателя. Любой решатель должен иметь класс, который отвечает за выбор переменной и функцию с помощью которой решается задача.
- `Interface Picker` – интерфейс для объектов который будут выбирать переменные и отвечать за составление ответа.
- `enumclass SolverType` – перечисление доступных видов решателя. Можно вызывать функцию решения отсюда, чем я активно пользовался.
- `class CommonClause` – класс для удобного хранения `Formula` в другой форме. Используется при анализе конфликтов.
- `class NaivePicker` – `Picker` реализующий выбор переменной для алгоритма DPLL. Имеет в себе двоичное дерево, реализованное через очередь и возвращение по переменной.
- `Class DPLLSolver` – SAT солвер использующий DPLL алгоритм. Данный алгоритм очень прост: если можно взять следующую переменную, то

берём ее, иначе решения нет. Далее анализируем, появляются ли конфликты, если есть, то возвращаемся назад чтобы выбрать другое значение для переменной. Если конфликтов нет, то подготавливаем следующие переменные для их назначения.

- `class CDCLPicker` – `Picker` с нехронологическим возвратом. Имеет в себе списки с родительскими выражениями (откуда последовало присвоение), отображение уровня решения для выбора литерала и пару других переменных для своей работы. Возврат в данной реализации нехронологический. При возврате мы оцениваем уровень решения и спускаемся на один вниз. Это значит, что при возврате мы можем отменить присвоения больше одной переменной. Так же существует функция `learn()`, которая добавляет новый набор литералов к задаче, для предотвращения конфликтов.
- `class CDCLSolver` – SAT-решатель с алгоритмом CDCL.

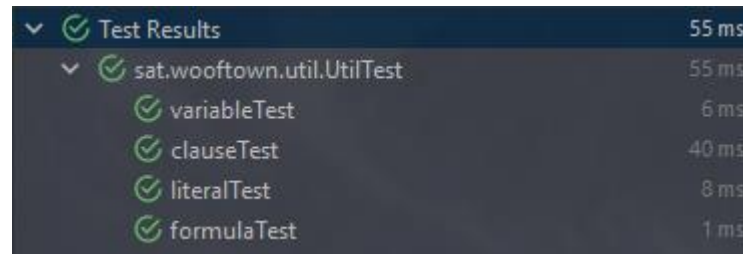
Пакет io:

В данном пакете содержится парсер, с помощью которого мы берём данные из файла формата DIMACS.

В данном проекте использовалась система автоматической сборки проекта Gradle. Далее рассмотрим тестирование и сбор статистики результатов работы программы.

ТЕСТИРОВАНИЕ ПРОГРАММЫ

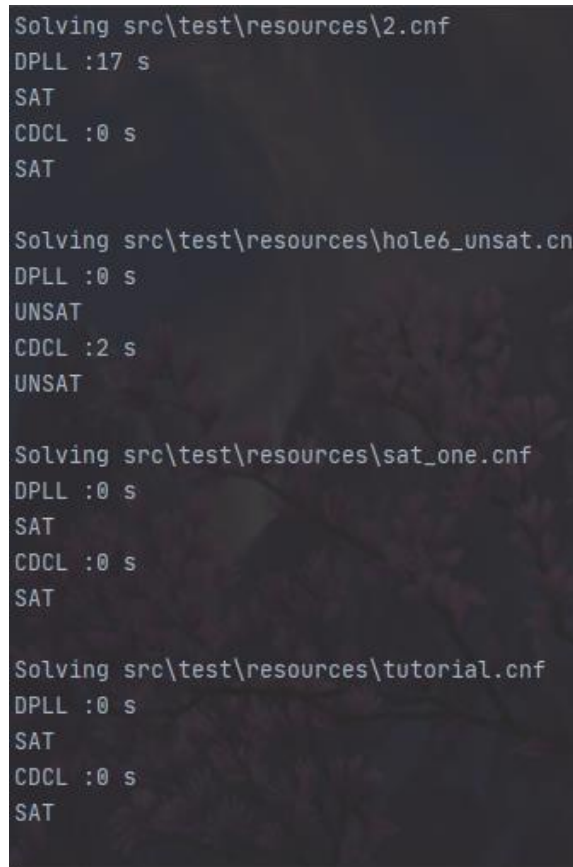
Для тестирования логики была использована библиотека JUnit версии 4.12. Были написаны автоматические тесты для структур данных из пакета util.



Test Results	55 ms
sat.wooftown.util.UtilTest	55 ms
variableTest	6 ms
clauseTest	40 ms
literalTest	8 ms
formulaTest	1 ms

Рис. 4 Полный перечень автоматических тестов

Для проверки работы самих решателей было решено написать тестирование с составлением отчёта. Был создан класс ReportGenerator. Главная функция данного класса берёт из указанной директории *.cnf файлы, решает их и выводит результат в отдельный файл.



```
Solving src\test\resources\2.cnf
DPLL :17 s
SAT
CDCL :0 s
SAT

Solving src\test\resources\hole6_unsat.cn
DPLL :0 s
UNSAT
CDCL :2 s
UNSAT

Solving src\test\resources\sat_one.cnf
DPLL :0 s
SAT
CDCL :0 s
SAT

Solving src\test\resources\tutorial.cnf
DPLL :0 s
SAT
CDCL :0 s
SAT
```

Рис. 5 Пример отчёта

СБОР СТАТИСТИКИ

Для более тонкой обработки данных о скорости решения была написана программа, которая берёт по 10 примеров из разных категорий (по сложности) и выводит время решения каждой из них. Примеры можно найти в директории “stats/benchmark/...”, отчёты по каждой категории находятся в директории “stats/benchmarkStats/...”. По полученных данным были построены графики зависимости времени решения от класса сложности примера. Рассмотрим графики.

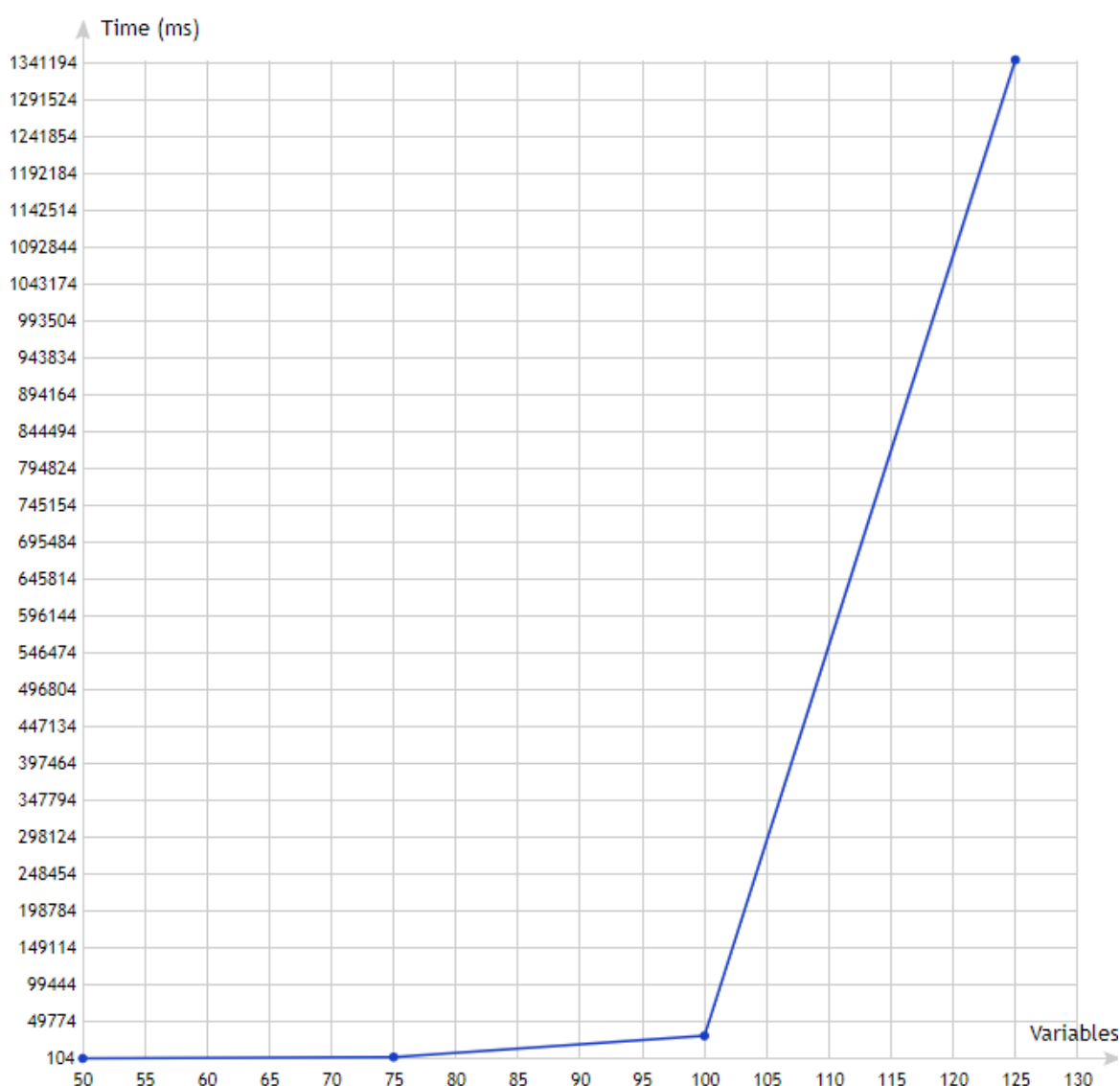


Рис. 6 CDCL - unsat

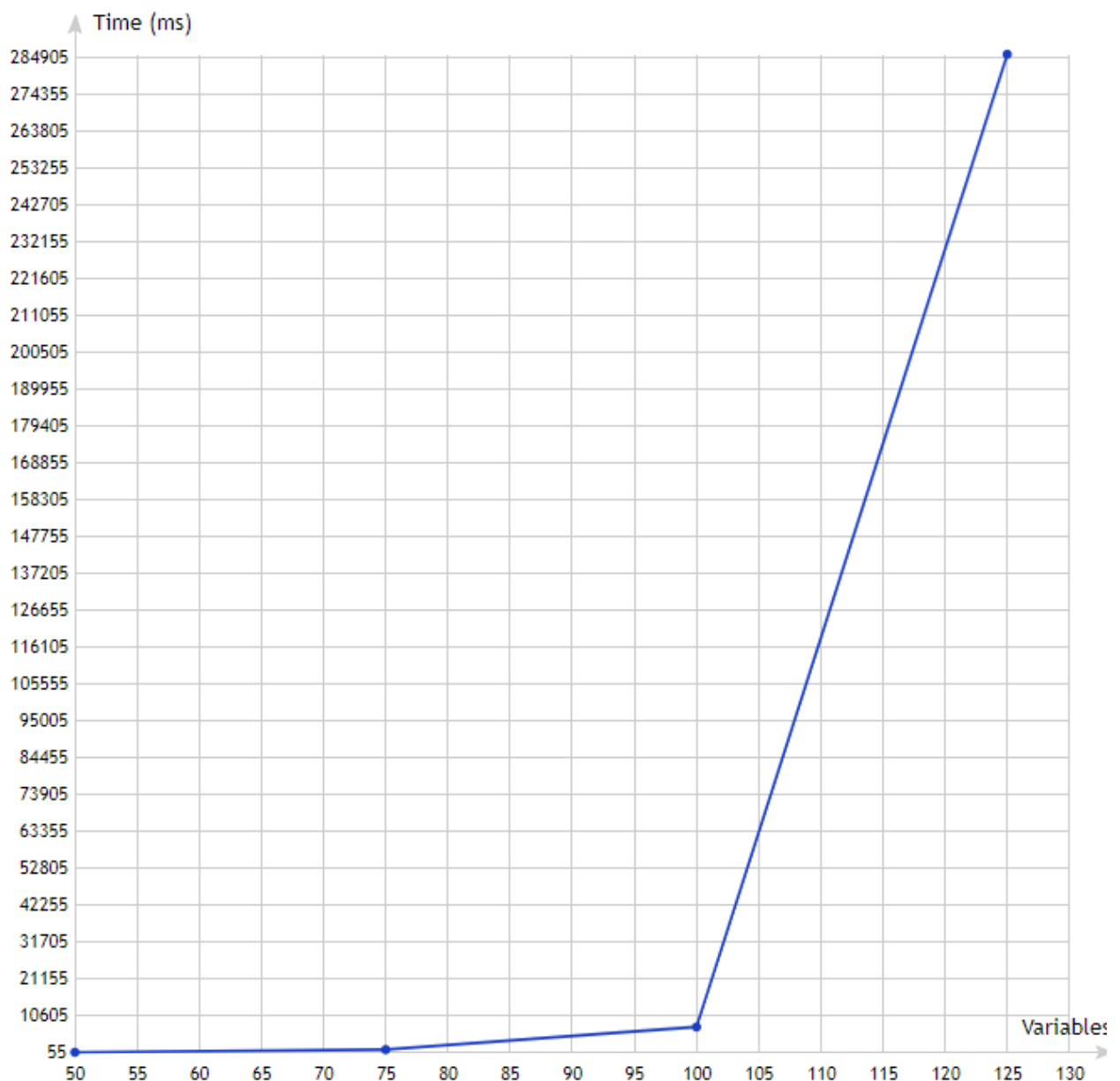


Рис. 7 CDCL - sat

Данные графики имеют похожую форму, и можно сказать, что *имеет пример решение или нет* влияет линейно на скорость выполнения. Можно сказать, что скорость выполнения экспоненциально зависит от количества входных данных.

Не вижу смысла приводить графики времени решения DPLL солвера. Это связано с тем, что за 6 часов решения алгоритму удалось решить только 20 самых легких проблем.

ЗАКЛЮЧЕНИЕ

Был создан SAT-solver на языке программирования Kotlin. Так же было проведено полное тестирование для проверки правильности работы. К сожалению, во время выполнения работы не было доступа к большим мощностям для более детального исследования зависимости времени выполнения от входных данных.

Список использованных источников

1. Документация языка Kotlin - <https://kotlinlang.org/docs/reference/>.
2. Таблица с собранной статистикой -
<https://docs.google.com/spreadsheets/d/15sQqe5EsysaCunL-88nYErk7WyyJgJTnDWQf8vZ0n2P8/edit?usp=sharing>
3. Общее описание алгоритмов и задач - <https://en.wikipedia.org>
4. SAT-solver для проверки выражений -
<https://www.comp.nus.edu.sg/~gregory/sat/>
5. Статья про CDCL-солвер - <http://satassociation.org/articles/FAIA185-0131.pdf>