CSE301 FINAL PROJECT - SAT SOLVER

A smart SAT solver in Haskell

Maika Edberg, Remy Seassau

November 27, 2022

PROJECT OVERVIEW

As most SAT solvers, ours is based on backtracking. We started from a simple implementation and added on other optimisations afterwards.

Here is a list of our optimisations:

Optimisation Name	Works
Unit Propagation	Yes
Pure Literal Elimination	Yes
Greedy Branching Heuristics	Yes
Two Watched Literals	No
Subsumption	Yes
Self-Subsumption	Yes
3-CNF	No

TESTING

We decided to test our solver against Minisat as a reference. We resorted to python scripting in order to bulk-test on an assortment of cnf files.

We used the time reported directly by Minisat and we timed our own solver directly in python.

We first tested the correctness of our solver by comparing our satisfiability results with Minisat's.

N.B. All times above 30 seconds were cut short and simply reported as 30.

FRAME OF REFERENCE

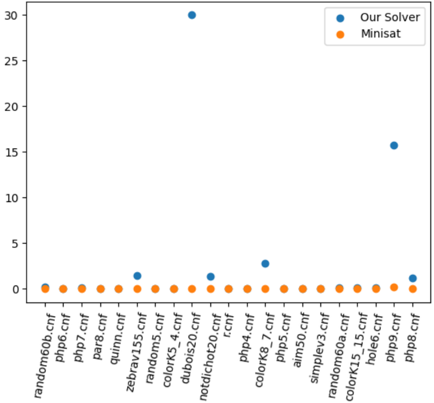

Figure: Unit Propagation vs Minisat

FRAME OF REFERENCE

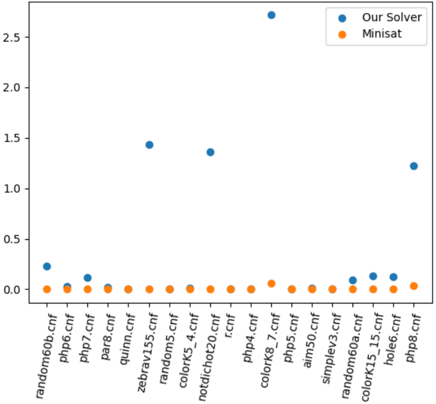

Figure: Unit Propagation vs Minisat (zoomed)

FRAME OF REFERENCE

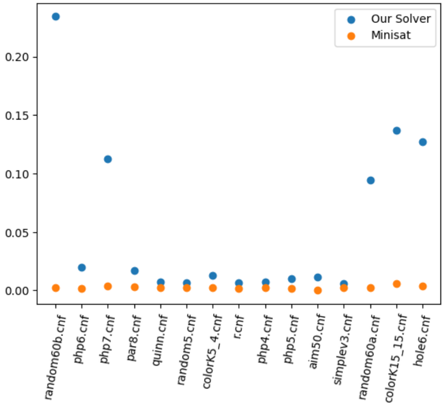

Figure: Unit Propagation vs Minisat (zoomed x2)

ALL OPTIMISATIONS

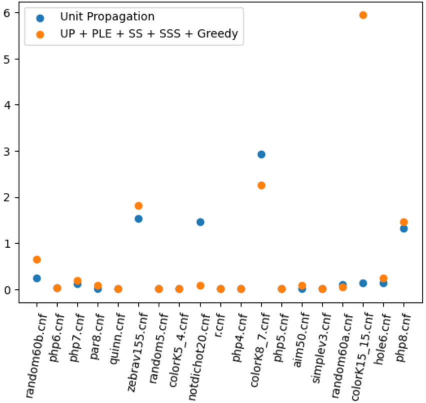

Figure: Unit Propagation vs All Optimisations

ALL OPTIMISATIONS

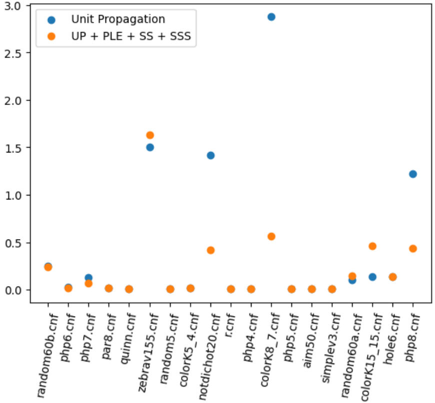

Figure: Unit Propagation vs All Optimisations w/o Greedy

SUBSUMPTION

```
simplify()
do
    S_0 = {set of clauses containing a literal occurring in some clause in Added}
    do
        S_1 = {set of clauses containing a literal occurring negatively in some clause in Added} ∪ Added ∪ Strengthened
        clear Added and Strengthened
        for each C ∈ S_1 do SelfSubsume(C)
        propagateToplevel()
    while (Strengthened ≠ ∅)
    for each C ∈ S_0 not deleted do subsume(C)
    do
        S = Touched; clear Touched
        for each x ∈ S do maybeEliminate(x)
    while Touched ≠ ∅
while Added ≠ ∅
```

```
simplify()
do
    S_1 = {set of clauses containing a literal occurring negatively in some clause in Added} ∪ Added ∪ Strengthened
    clear Added and Strengthened
    for each C ∈ S_1 do SelfSubsume(C)
    propagateToplevel()
while (Strengthened ≠ ∅)
for each C not deleted do subsume(C)
```

SUBSUMPTION

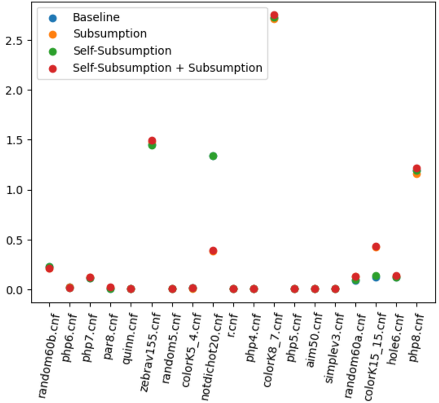

Figure: Subsumption combinations vs UP

SUBSUMPTION

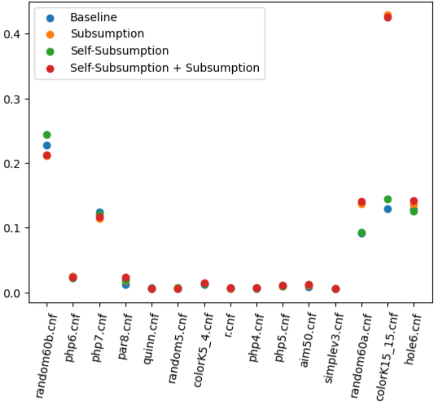

Figure: Subsumption combinations vs UP

3-CNF

We wanted to try applying 3CNF with the idea that it could potentially increase the use of Unit Propagation.

However, we are prefectly content with small clauses, so we did not implement 3-CNF, but rather max(3)-CNF.

Unfortunately the implementation is lacking in correctness as we currently fail to not find solutions.

3-CNF

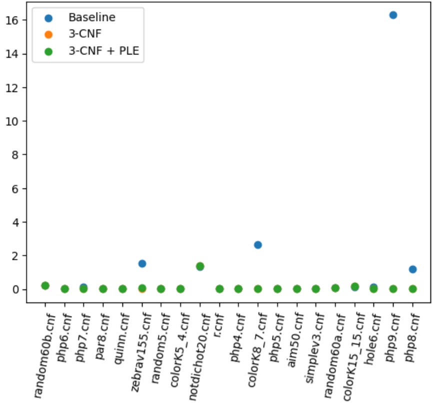

Figure: 3-CNF vs UP

3-CNF

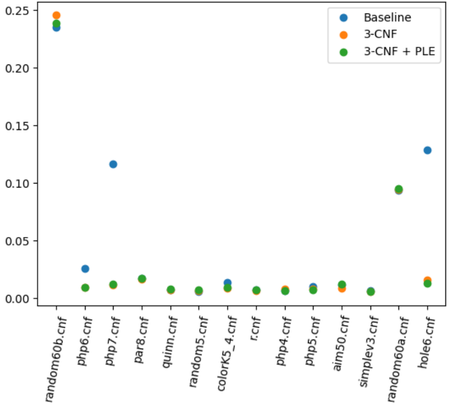

Figure: 3-CNF vs UP

UNIT PROPAGATION

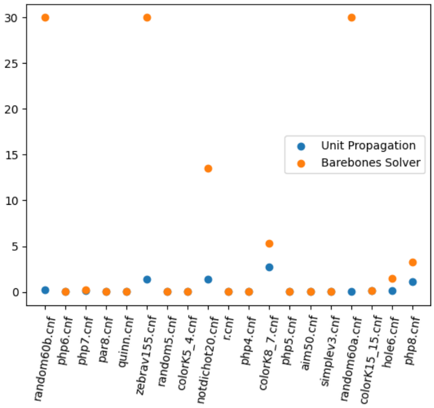

Figure: Unit Propagation vs Barebones Solver

UNIT PROPAGATION

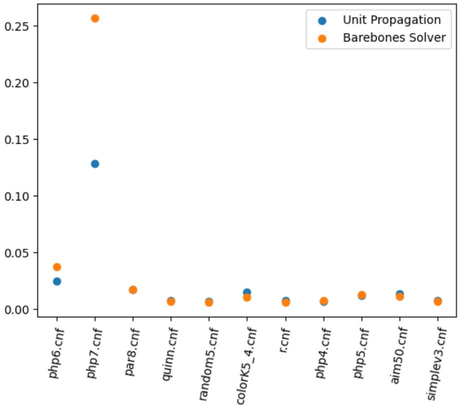

Figure: Unit Propagation vs Barebones Solver

PURE LITERAL ELIMINATION

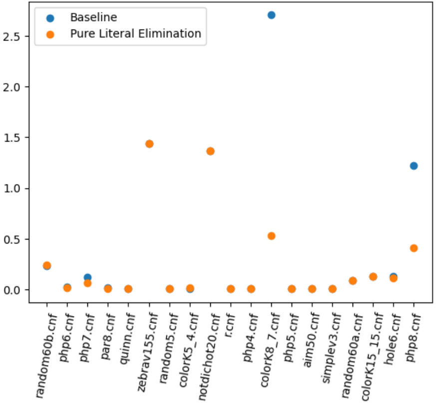

Figure: PLE+UP vs UP

PURE LITERAL ELIMINATION

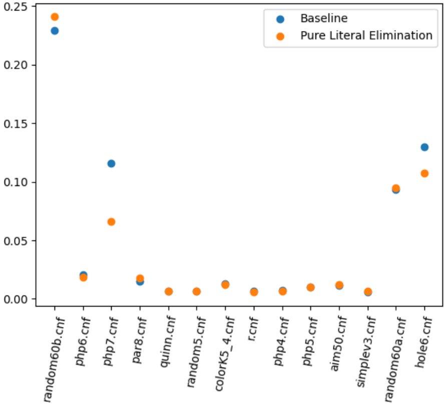

Figure: PLE+UP vs UP

GREEDY BRANCHING HEURISTICS

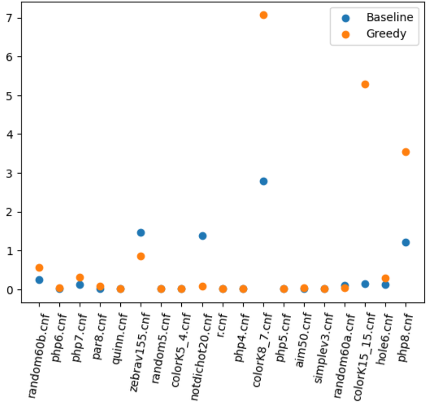

Figure: Greedy+UP vs UP

GREEDY BRANCHING HEURISTICS

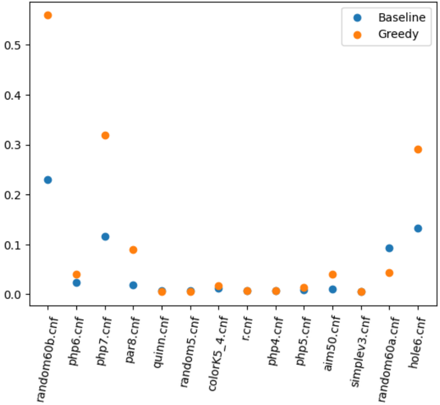

Figure: Greedy+UP vs UP

TWO WATCHED LITERALS

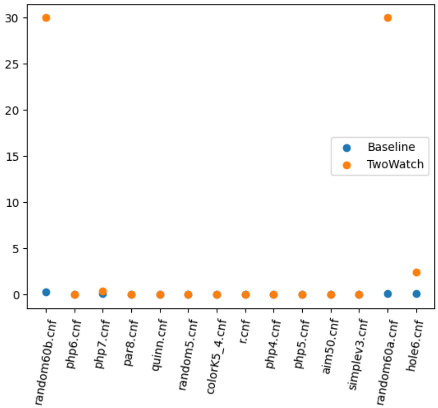

Figure: TwoWatched vs UP