RustによるSATソルバーの実装と監視リテラルによる単位伝播の実装

田中久温

やったこと

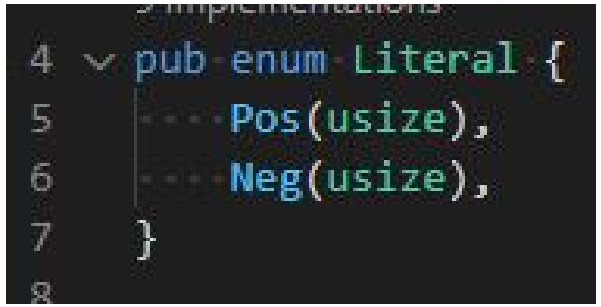
1. 配布されたソルバープログラム(以降 simple minisat)をRust言語で書き直した
2. Rust言語で書いたソルバーに監視リテラルによる単位伝播を実装した

目的: Rust言語による実装

- 他の言語で再実装することで, プログラムの動作の詳細を学べる
- RustはC/C++に置き換わることを目指しており, C++並 に速い
  - コンパイラが優秀なのでちょっと早くなるかも
- Rustで実装すれば, lit型やlbool型を綺麗に扱える
  - Cのように, 内部でintとして扱う処理とか書かなくていい
  - 慣れればC/C++よりも直感的にコードを読める

Rustの特徴

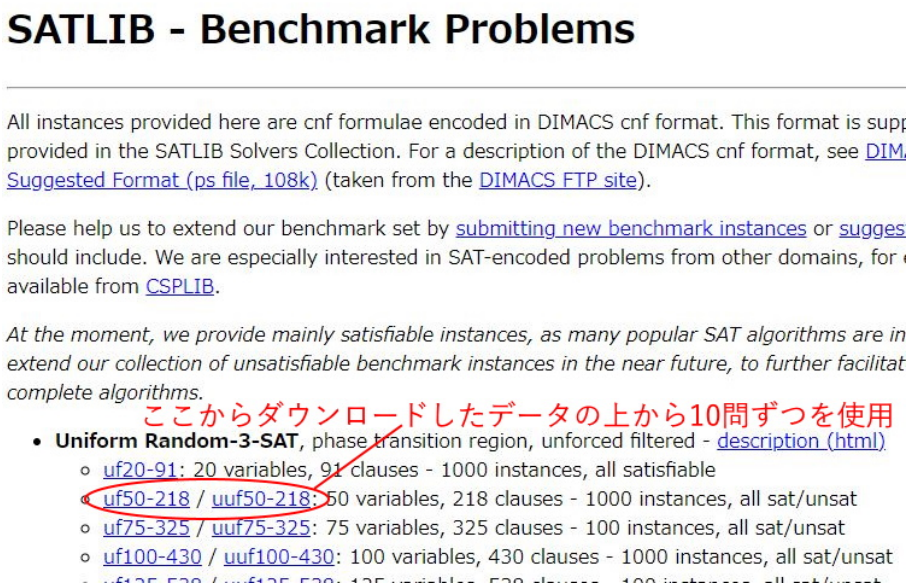
- いろんな種類の値をまとめて扱いやすくするEnumにより
  正リテラル負リテラルが直感的に



- Enumによって実装されたOption型により
  Undefや未割り当てのようなものも言語機能で整理できる

動作確認

- SATを10問, UNSATを10問用意
- SAT, UNSATだけでなく, simple minisatと解(割り当て)が一致することも確認
- ログ出力から正しい出力が得られていることを, スクリプトを書いて確認



ログ出力の例

Satisfying solutionの行が同じフォーマットなので, 抜き出して比較すれば解が一致しているか判定できる

uf50-01.cnf.log

test_log > my_simple_solver > uf50-01.cnf.log

```
[2021-11-24T14:09:46Z INFO  three_sat_solver_practice_2_h] input file: test/uf50-01.cnf
[2021-11-24T14:09:46Z INFO  three_sat_solver_practice_2_h::solver] ==========[MINIMUMSAT]==========
[2021-11-24T14:09:46Z INFO  three_sat_solver_practice_2_h::solver] | Conflicts |     ORIGINAL     |
[2021-11-24T14:09:46Z INFO  three_sat_solver_practice_2_h::solver] |           | Clauses Literals |
[2021-11-24T14:09:46Z INFO  three_sat_solver_practice_2_h::solver] ================================
[2021-11-24T14:09:46Z INFO  three_sat_solver_practice_2_h::solver] |         0 |     218      654 |
[2021-11-24T14:10:01Z INFO  three_sat_solver_practice_2_h::solver] ================================
conflicts    : 29381905
decisions    : 29381937
CPU time     : 14.868 sec
SATISFIABLE
Satisfying solution: x0=0 x1=1 x2=0 x3=1 x4=1 x5=1 x6=1 x7=1 x8=1 x9=0 x10=0 x11=1 x12=0 x13=1 x14=0 x15=0 x16=0 x17=0 x1
```

uf50-01.cnf.log

test_log > simple_minisat > uf50-01.cnf.log

```
=================================[MINISAT]==================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==============================================================================
|         0 |     218      654 |       0       0        0     0.0 |  0.000 % |
==============================================================================
restarts     :          1
conflicts    :   29381905          ( 1747161 / sec )
decisions    :   29381937          ( 1747162 / sec )
propagations :          0          (       0 / sec )
inspects     :          0          (       0 / sec )
conflict literals :     0          (    -nan % deleted )
CPU time     :      16.82 sec

SATISFIABLE

Satisfying solution: x0=0 x1=1 x2=0 x3=1 x4=1 x5=1 x6=1 x7=1 x8=1 x9=0 x10=0 x11=1 x12=0 x13=1 x14=0 x15=0 x16=0 x17=0 x1
```

ベンチマーク

- timeout 1000 で実行して, 1000秒間にいくつ解けるか調べた

SATLIB - Benchmark Problems

All instances provided here are cnf formulae encoded in DIMACS cnf format. This format is supp
provided in the SATLIB Solvers Collection. For a description of the DIMACS cnf format, see DIM
Suggested Format (ps file, 108k) (taken from the DIMACS FTP site).

Please help us to extend our benchmark set by submitting new benchmark instances or sugges
should include. We are especially interested in SAT-encoded problems from other domains, for
available from CSPLIB.

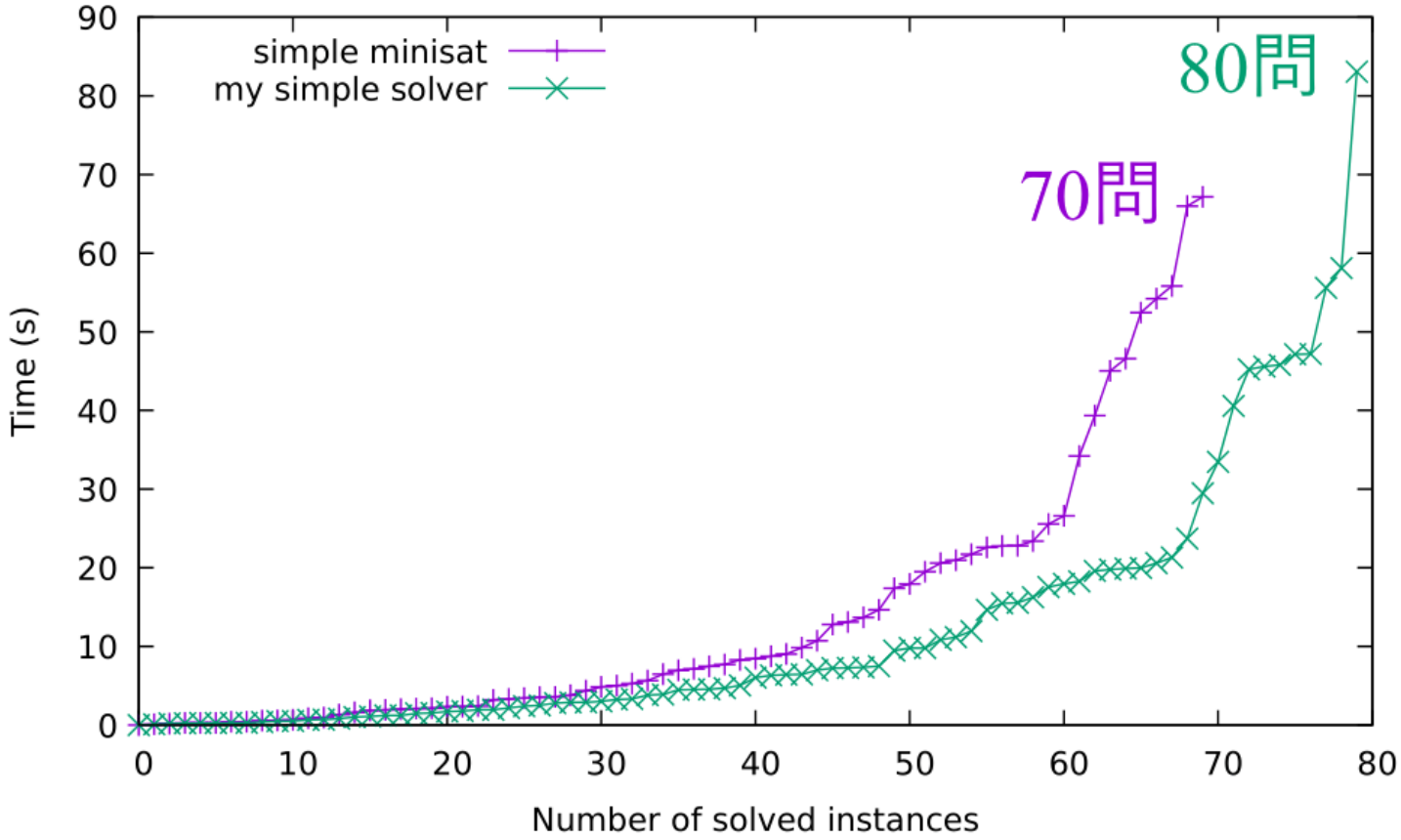
*At the moment, we provide mainly satisfiable instances, as many popular SAT algorithms are in
extend our collection of unsatisfiable benchmark instances in the near future, to further facilitat
complete algorithms.*

このデータをファイル名の上から順に実行

- **Uniform Random-3-SAT**, phase transition region, unforced filtered - description (html)
  - uf20-91: 20 variables, 91 clauses - 1000 instances, all satisfiable
  - uf50-218 / uuf50-218: 50 variables, 218 clauses - 1000 instances, all sat/unsat
  - uf75-325 / uuf75-325: 75 variables, 325 clauses - 100 instances, all sat/unsat
  - uf100-430 / uuf100-430: 100 variables, 430 clauses - 1000 instances, all sat/unsat

ベンチマーク結果

- アルゴリズム的には同じだが, Rustで実装した方が少し早くなった



目的: 監視リテラルによる単位伝播

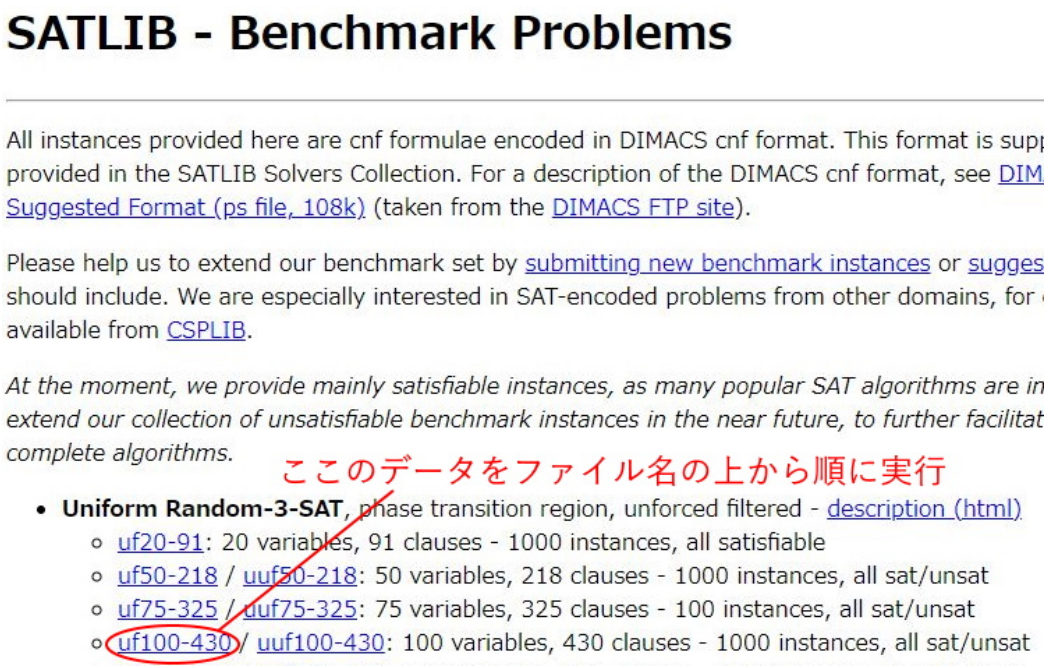
- ソルバーの高速化するために, 監視リテラルによる単位伝播を実装した
- 探索途中, 最後のリテラルがfalseになればただちに矛盾が起こる節が現れるので, その時点で次のリテラルの値も確定したい. そのような, 節を効率よく検出して無駄な探索を減らす

実装: 監視リテラルによる単位伝播

- アルゴリズムの詳細は割愛
- バックトラックすべきかどうかを判定する関数(is_falsified)を置き換え
- trailに入ってるリテラルと矛盾するリテラルを持つ節だけを参照
- ¬リテラルをキーにして監視リテラルにアクセス
- 監視リテラルは各節の先頭2つを監視, 節内でリテラルをスワップして監視するリテラルを変更
- MiniSatから監視リテラルに関する部分を抜粋したブログを見ながら実装
  https://www.ueda.info.waseda.ac.jp/~yuji/sb/log/eid9.html
  (ときどき本家のMiniSatのコードも参照しながら)
- 1つ目の実験と同様の方法で動作確認, 変数選択の順番は変えてないので, 出力される解も同じ

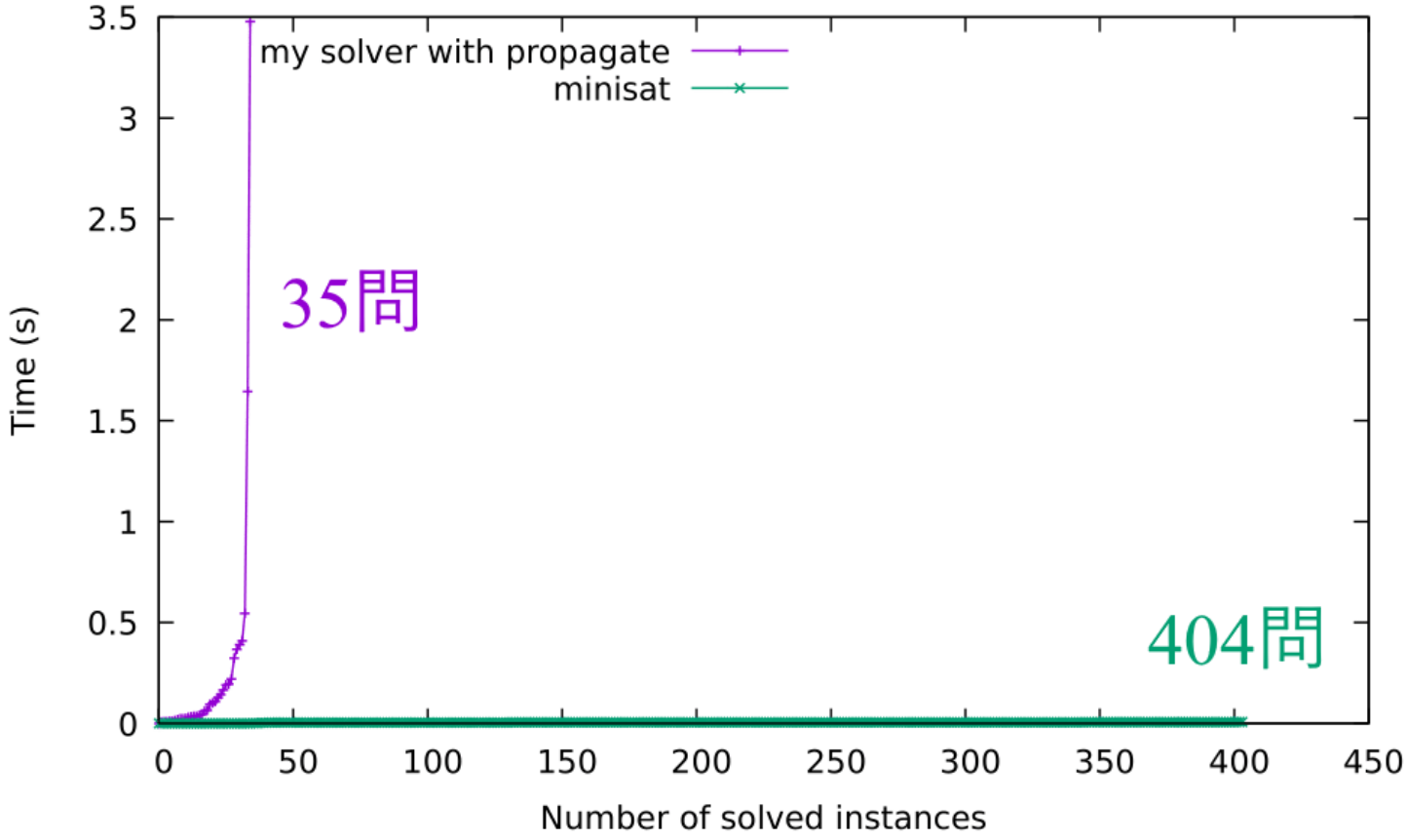
ベンチマーク

- simple minisatと同じベンチマークをすると, 14.8秒かかる問題が0.005秒で解けるので比較にならない
- 本家minisatと比較をしてみる
- timeout 10 で実行して 10秒でいくつ解けるか調べた



ベンチマーク結果

- 本家minisatと11倍くらい差が出た



まとめ

- simple minisatのRust実装, さらにそのRust実装に監視リテラルを実装したものに監視リテラルを実装した
- simple minisatをRustで実装するだけで少し早くなった
- さらに監視リテラルによる単位伝播を導入すると, 比べられないくらい早くなった
- 監視リテラルだけでは, 本家minisatの1/11程度の早さしかでなかった

実行環境

OS: Ubuntu 20.04.2 LTS (WSL on Windows10)

CPU: AMD Ryzen 7 5800X

Rust: rustc 1.56.1, cargo 1.56.0

canceluntileのコード

左がC 右がRust

```c
static inline void solver_canceluntil(solver* s, int level) {
    lit*    trail;
    lbool*  values;
    int     bound;
    int     c;

    if (solver_dlevel(s) <= level)
        return;

    trail  = s->trail;
    values = s->assigns;
    bound  = (veci_begin(&s->trail_lim))[level];

    for (c = s->qtail-1; c >= bound; c--) {
        int     x  = lit_var(trail[c]);
        values [x] = l_Undef;
    }

    s->qhead = s->qtail = bound;
    veci_resize(&s->trail_lim,level);
}
```

```rust
    pub fn canceluntil(&mut self, level: usize) {
        if self.dlevel() <= level {
            return;
        }       You, a week ago • feat: 1つのデータで正しく動作すること
        let bound: usize = self.trail_lim[level];
        for c: usize in bound..self.trail_tail {
            if let Some(lit: Literal) = self.trail[c] {
                self.assigns[lit.var()] = None;
            } else {
                error!("trail: 1度も初期化されてない部分にアクセス");
            }
        }
        self.trail_tail = bound;
        self.trail_lim.resize(new_len: level, value: 0);
    }
```

新しい監視リテラルを見つける部分のコード

左がC 右がRust

やっている処理はほぼ同等, Rust版は古い監視リテラルの削除まで行っている

```
// Look for new watch:
for (int k = 2; k < c.size(); k++)
    if (value(c[k]) != l_False){
        c[1] = c[k]; c[k] = false_lit;
        watches[~c[1]].push(w);
        goto NextClause; }
```

```
// 1番目のリテラルを取替
for k: usize in 2..clause.len() {
    // 未割り当てか, true となっているリテラルを探す
    let is_satisfied: Option<bool> = self.searcher.is_satisfied(lit: &clause[k]);
    if is_satisfied == Some(true) || is_satisfied == None {        You, 2 days ag
        // 変数の取替操作
        clause[1] = clause[k];
        clause[k] = false_lit;
        self.watched_lit_indices.get_mut(&clause[1].not()).unwrap().insert(i);
        self.watched_lit_indices.get_mut(&clause[k].not()).unwrap().remove(&i);
        continue 'clause; // 次の節へ
    }
}
```