Pueblo Pseudo-Boolean SAT Solver

Hossein M. Sheini hsheini@umich.edu
Electrical Engineering and Computer Science Department,
University of Michigan, Ann Arbor, MI

1. Introduction

This document is a brief description of the Pueblo Pseudo-Boolean (PB) Solver version 1.5. This solver is developed at the University of Michigan, Ann Arbor, MI by Hossein Sheini, supervised by Professor Karem A. Sakallah. For details on the algorithms adopted in Pueblo the reader is referred to [1]. The solver is available for download at

`http://www.eecs.umich.edu/~hsheini/pueblo`.

Pueblo is an extension to MiniSAT 1.12 [2]. The implementation details, covered in this document, should be read together with the MiniSAT description presented in [2]. Note that Pueblo can solve both PB satisfiability and optimization problems and has the capability to handle integer coefficients that can be represented with at most 32 bits.

2. Constraints

Exploiting the capability of MiniSAT to handle arbitrary constraints over Boolean variables through its *Constr* abstract base class, we added a *PseudoBool* constraint class in addition to its existing *Clause* constraint class. Unique procedures for propagating and calculating reasons for this class of constraints are presented in Figure 1. Using the base class enables Pueblo to use all of MiniSAT's solving procedures independent of the type of constraint being handled. These *PseudoBool* constraints are either created at the beginning of the search or learned through PB learning procedure. Each *PseudoBool* constraint is created or learned using the *PB_new* procedure, presented in Figure 2.

3. Accumulator

The accumulator is the PB constraint that contains the resolvent of the cutting plane based PB learning procedure. The PB learning process starts with the violated constraint considered as the accumulator and continues by adding it to the implying constraints and saving it back in itself. In order to avoid searching for variables, the accumulator is implemented as an array whose size is equal to the number of problem variables. In this scenario, CNF learning consists of detecting and separately storing the false literals of the accumulator. Details of the accumulator class are presented in Figure 3.

4. Pueblo Solver

Pueblo major modifications in MiniSAT solving procedure are listed below:

Figure 1. Implementation of *PseudoBool* constraint class in Pueblo

```
class PseudoBool : public Constr
    int rhs
    int watchsum
    int amax
    float activity
    bool learnt
    Vec<PBTerm> terms          - PBTerm comprises of a literal, a coefficient and watched bool.
                               - terms is sorted based on coefficients
    bool propagate(Solver S, lit p)
        int p_idx = terms.index(p)
        terms[p_idx].unwatch(this)
        - update watchsum
        for(int i = 0; i < size() && watchsum < amax + rhs; i++)
            Lit lit = terms[i].lit
            if(S.value(lit)≠ l_False && !terms[i].watched()) terms[i].watch(S, this)
        - check for conflict
        if(watchsum < rhs_goal)
            terms[p_idx].watch(S, this)
            return FALSE
        - check for satisfiability
        if(watchsum ≥ amax + rhs) return TRUE
        for(int i = 0; i < size(); i++)
            Lit lit = terms[i].lit
            int coeff = PBTerms[i].coeff
            if(watchsum ≥ coeff + rhs) break
            if(S.value(lit) == l_Undef)
                if(!S.enqueue(lit, this))
                    terms[p_idx].watch(S, this)
                    return FALSE - conflict in the Solver
        return TRUE

    void calcReason(Solver S, lit p, vec<lit> out_reason)
        - all learned constraints are initially active
        if (learnt) S.pbBumpActivity(this)
        - calculate the multiplier to eliminate p from accumulator
        int mul = (p == lit_Undef) ? 1 : S.accumulator.coeff(var(p))
        S.accumulator.goal += mul * rhs
        for (int i = 0; i < size(); i++)
            lit = terms[i].lit
            if(lit == p)
                S.accumulator.goal − = mul*terms[i].coeff
                continue
            - adds this literal to the accumulator
            UpdateAccumulator(S, mul, terms[i].lit, terms[i].coeff)
            - saves the false literals for CNF learning
            if (S.value(lit) == l_False) out_reason.push(¬lit)
```

4.1 Learning

Pueblo adopts the same learning flow as in MiniSAT augmenting it with cutting plane generation (PB learning) at each step. At each step in the backward traversal of the implication graph, the CNF or PB constraint involved in that implication is added to the

Figure 2. Implementation for creating and adding new *PseudoBool* constraints in Pueblo

```
bool PB_new(Solver S, int goal, Vec<lit_coef> pbs, PseudoBool out, Clause c, bool learnt)
    out = new PseudoBool
    out.rhs = goal
    for (int i = 0; i < pbs.size(); i++)
        out.terms[i].lit = Lit(pbs[i].lit)
        if(pbs[i].coef < 0)
            out.rhs += abs(pbs[i].coef)
            out.terms[i].lit = ¬ out.terms[i].lit
        out.terms[i].coeff = abs(pbs[i].coef)
    sort(out.terms, size())
    if( (out.terms[0].coeff == out.rhs && out.terms[size()].coeff == out.rhs) ||
      out.rhs == 1 ) - checking if PB constraint is equal to a CNF clause
        bool ret = convertPBtoCNF(S, out, c, learnt) - creates clause with literals in terms of out
        xfree(out)
        return ret
    out.amax = out.terms[0].coeff
    out.learnt = learnt
    if(out.rhs == 0) return TRUE
    for (int i = 0; i < size(); i++) - setting up the watch list
        Lit lit = out.terms[i].lit
        if (out.watchsum < out.rhs + out.amax) out.terms[i].watch(S, this)
        if (out.watchsum ≥ out.rhs + out.amax) break
    if(out.watchsum < out.rhs) return FALSE
    if(out.watchsum < out.rhs + out.amax)
        for(int i = 0; i < size(); i++)
            Lit lit = out.terms[i].lit
            if(S.value(lit) == l_Undef)
                if(out.watchsum ≥ out.terms[i].coeff + out.rhs) break
                if(!S.enqueue(lit)) return FALSE
    if(learnt) S.varBumpActivity(lit, coeff/out.rhs)
    return TRUE
```

accumulator while performing MiniSAT's *calcReason()* routine to eliminate the implied literal. The cutting plane is saved in the accumulator while the learned CNF clause is stored in the *out_reason* following the learning procedure of MiniSAT (refer to Figure 1). If an over-satisfaction is detected in the accumulator, the step resulting in over-satisfaction is undone and replaced by adding the weakened CNF clause to the accumulator. The learning stops when the first UIP is reached as detected by MiniSAT *analyze* procedure. The details of conflict analysis procedure of Pueblo based on the *analyze* method of MiniSAT is presented in Figure 4.

4.2 Backtracking and Constraint Recording

In Pueblo, the lowest decision level at which the learned PB constraint is unit is determined by checking the PB unit invariant [1] at each decision level. If such decision level was found, the solver backtracks to the minimum level between this level and the *backtrack_level* computed in MiniSAT's *analyze* routine. Otherwise, the highest decision level at which the learned PB constraint is not violated is determined and the solver backtracks to that level or the *backtrack_level*, whichever lower. This procedure is demonstrated in Figure 5. Both

Figure 3. Implementation of the *accumulator* class in Pueblo

```
class accumulator
    Vec<lit_coef> terms - lit_coef is a structure with a literal and an integer coefficient
    int goal
    - set all literals in terms to undefined
    void reset()
    - check if satisfied, subtracts pb and adds CNF weakened of pb
    void checkOverSatisfaction(PseudoBool pb)
    - multiplies all coefficients by mul
    void multiplyBy(int mul)
    - adds to the coefficient of var(lit)
    void updateCoef(Lit lit, int coef)
    - returns the coef of var
    int coeff(int var)
    - return literal having var
    Lit lit(int var)
    - returns sum of coefficients of false literal at level i
    int sumAssignedFalseAtLevel(int i)
    - returns the largest coefficient among unvalued literals
    int amaxAtLevel(int i)
    - constructs a vector of literals with non-zero coefficients
    void getLiterals( Vec<Lit> lits)
    - converts all coefficients to positive by changing the sign of literals
    void normalize()
```

learned PB and CNF constraints are recorded and their watched literals are properly setup, as demonstrated in Figure 6.

4.3 Activity Heuristics

The variable activity heuristic of MiniSAT is extended to PB constraints in such a way that it recognizes the coefficient of each variable in the learned constraint. Therefore, the activity of each variable that is present in a newly learned PB constraint is increased by the ratio of its coefficient to the right-hand side of that PB constraint.

4.4 Constraint Removal

In Pueblo, the number of active PB constraints is periodically reduced to a fixed number of constraints. This procedure basically removes all PB constraints that are not *locked* (to an implication) and are less active than a pre-set threshold limit. This limit is increased at each restart. Through our experiments on benchmarks used in the PB'05 evaluation [3], we found that an initial threshold of 50 and a growth rate of 10% produces the best results.

References

[1] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-boolean sat solver," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 2, p. 61, 2006.

[2] N. Eén and N. Sörensson, "An extensible SAT-solver." in *SAT*, 2003, pp. 502–518.

Figure 4. Implementation for conflict analysis method in Pueblo

```
void analyze(Constr confl, Vec<Lit> out_learnt, int out_btlevel)
    Vec<char> seen = analyze_seen, seen.growTo(nVars(), 0)
    int pathC = 0
    Lit p = lit_Undef
    Vec<Lit> p_reason
    bool inPB = FALSE
    out_learnt.push()
    out_btlevel = 0
    do
        if(inPB)
            - finding the multipliers to generate the cutting plane such that p is removed
            int mul = confl.terms.getCoeff[var(p)]
            accumulator.multiplyBy(mul)
        p_reason.clear()
        confl.calcReason(this, p, p_reason)
        - check if accumulator is satisfied and if so undo adding and replace with weakened confl
        accumulator.checkOverSatisfaction(confl)
        for (int j = 0; j < p_reason.size(); j++)
            Lit q = p_reason[j]
            if (!seen[var(q)])
                seen[var(q)] = 1
                if (level[var(q)] == decisionLevel())
                    pathC++
                else if (level[var(q)] > 0)
                    out_learnt.push(¬q)
                    out_btlevel = ::max(out_btlevel, level[var(q)])
        - Select next constraint to look at:
        do
            p = trail.last()
            confl = reason[var(p)]
            - check if this literal should be removed from the accumulator
            inPB = accumulator.coeff(var(p)) ≠ 0 && (value(accumulator.lit(var(p))) == l_False
            undoOne()
        while(!seen[var(p)])
        seen[var(p)] = 0
        pathC–
    while (pathC > 0)
    out_learnt[0] = ¬p
```


[3] Pseudo-Boolean Evaluation PB'05, `http://www.cril.univ-artois.fr/PB05/`.

Figure 5. Implementation of PB backtracking in Pueblo

```
void undoPB()
    accumulator.normalize() - convert all coefficients to positive
    int tmp_lhs = accumulator.goal
    for(int i = root_level; i < decisionLevel(); i++)
        - subtracts the sum of coefficients in the accumulator became false at level i
        tmp_lhs − = accumulator.sumAssignedFalseAtLevel(i)
        - check if accumulator is unit/conflict at this level
        if( tmp_lhs < accumulator.goal + accumulator.amaxAtLevel(i) )
            if(tmp_lhs < accumulator.goal) - no unit level exists
                bt_level = i-1
            else bt_level = i
            break
    cancelUntil(::max(bt_level, root_level))
```

Figure 6. Implementation of conflict-induced constraint recording method in Pueblo

```
bool recordPB( Vec<Lit> clause, int backtrack_level)
    PseudoBool pb
    Clause c
    Vec<Lit> PBLits - literals in the learned PB constraint
    accumulator.getLiterals(PBLits)
    undoPB()
    if(!PBnew(this, PBLits, pb, c)) return TRUE - learn the PB constraint
    if(pb ≠ NULL )
        pb_learnts.push(pb)
        pbDecayActivity()
    if(c ≠ NULL)
        learnts.push(c)
        claDecayActivity()
    Clause CNFlearnt
    bool CNFunit = FALSE
    if(decisionLevel() > backtrack_level)
        cancelUntil(::max(bt, root_level)) - backtrack to earlier level
        CNFunit = TRUE
    check(Clause_add(this, clause, CNFlearnt)) - learn the CNF clause
    if(CNFunit) check(enqueue(clause[0], CNFlearnt))
    if(CNFlearnt ≠ NULL)
        learnts.push(CNFlearnt)
        claDecayActivity()
    return FALSE
```