A Tutorial on CDCL

CS3317: Artificial Intelligence

2022.11.11

CDCL Recap

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

Decision heuristics: choose the next literal to add to the current partial assignment based on the state of the search

Learning: F augmented with a conflict clause that summarizes the root cause of the conflict

Non-chronological backtracking: backtracks b levels, based on the cause of the conflict

Main Routines & Data Structures

- Routines
  - Unit Propagation (BCP)
  - Conflict Analysis
- Data Structures
  - 2-Watched Literals # to record two literals, preferably unassigned ones, for each clause
  - Trails # to record the partial assignment and why each has certain value

Unit Propagation

- Intuition behind unit propagation
  - To identify variables which must be assigned a specific value
  - If an unsatisfied clause is identified, a conflict condition is detected, and the algorithm backtracks
- How do we know when we can do unit propagation?
  - Key idea: detect the unit clauses => all but one literal is assigned false
  - Methodology: a naïve approach
    - For each clause, keep a count of the false literals in the clause
    - For each literal, keep a list of clauses it appears in
    - If x is made false, increment the count for every clause it is in
    - If that count is equal to the `clause_length - 1`, the clause has become unit

What we care now

What is handled next

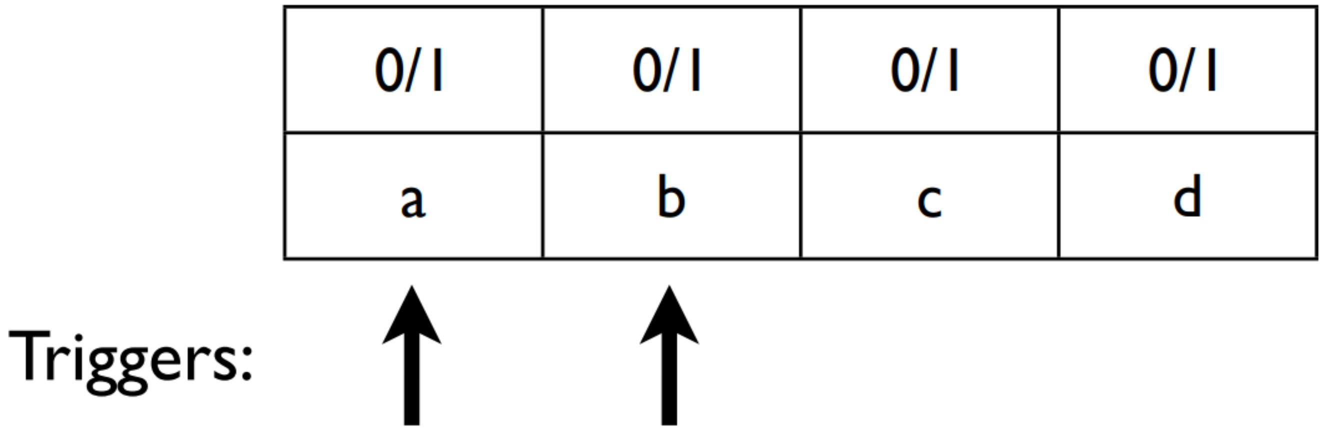
1. Requires work for every clause x appears in
2. Requires work to restore the counts on backtrack

Can be improve with a lazy data structure: 2-watched literals

2-Watched Literals

- Key idea: If two literals are either unassigned or assigned true, no need to do anything
- We distinguish two literals of each clause as being the watched literals

1. Maintains much less information
2. Requires no action when backtracking



- $a \lor b \lor c \lor d$

2-Watched Literals (A Watched is Assigned False)

0	0/1	0/1	0/1
a	b	c	d

Triggers:

- *a* assigned false.
- Update pointer.

2-Watched Literals (An Unwatched is Backtracked)

0/1	0/1	0/1	0/1
a	b	c	d

Triggers:

- Backtrack. *a* unassigned.
- **Pointers do not move back**

2-Watched Literals (A Watched is Assigned True)

0/1	**1**	0/1	0/1
a	**b**	c	d

Triggers:

- If *b* is assigned true, pointer doesn't move.

2-Watched Literals (An Unwatched is Assigned)

0	0/1	0/1	**0**
a	b	c	**d**

Triggers:

- If other variables assigned, nothing happens!

2-Watched Literals (A Watched is Assigned False and Update Fails)

0	0	**1**	0
a	b	**c**	d

Triggers:

- We can set the remaining literal
- i.e. do unit propagation since this clause is unit

2-Watched Literals (A Watched is Backtracked)

0	0/1	0/1	0
a	b	c	d

Triggers:

- Triggers in the right place to continue after backtracking.

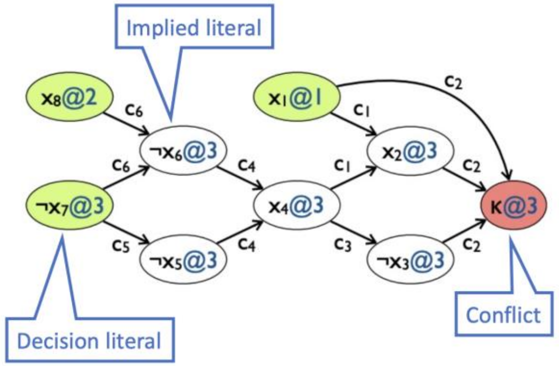
Unit Propagation (Pseudocode)

- if len(trail) == 0: # first unit propagation
  - add literals in the unit clauses to trail and set up_idx to 0
- while up_idx < len(trail) # more lits. to unit propagation
  - x = trail[up_idx]; up_idx += 1 # next lit. to unit propagation
  - for each clause C watched by –x # -x is now FALSE
    - y = second watched literal
    - if y == TRUE: continue
    - if there exists z = a non-false literal in c with z ≠ x and z ≠ y:
      - move C from x's watched clause list to z's watched clause list
    - else: # all lits. are false except possibly for y
      - if y == FALSE: return C # as the antecedent of the conflict
      - else: set y to TRUE and put (y, index of C) on the trail
- return None

Use l2c_watch (a dictionary mapping from literal to its watching clauses)

Use c2l_watch (a dictionary mapping from clause index to its watched literals)

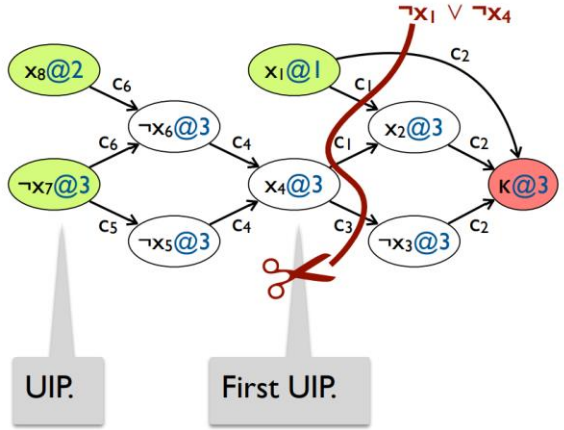
Conflict Analysis



- Intuition behind conflict analysis
  - Whenever a conflict is found, we can make use of it to learn a new clause
  - It forbids this set of literals from appearing again in this search
- How do we obtain the most valuable clause?
  - Key idea: find a cut of the implication graph!
  - Methodology: a naïve approach
    - Choose the decision literals directly
    - Set new clause as C = {-l: l is the decision literals}

Too specific to this part of the search to be useful later

Conflict Analysis w/ First UIP

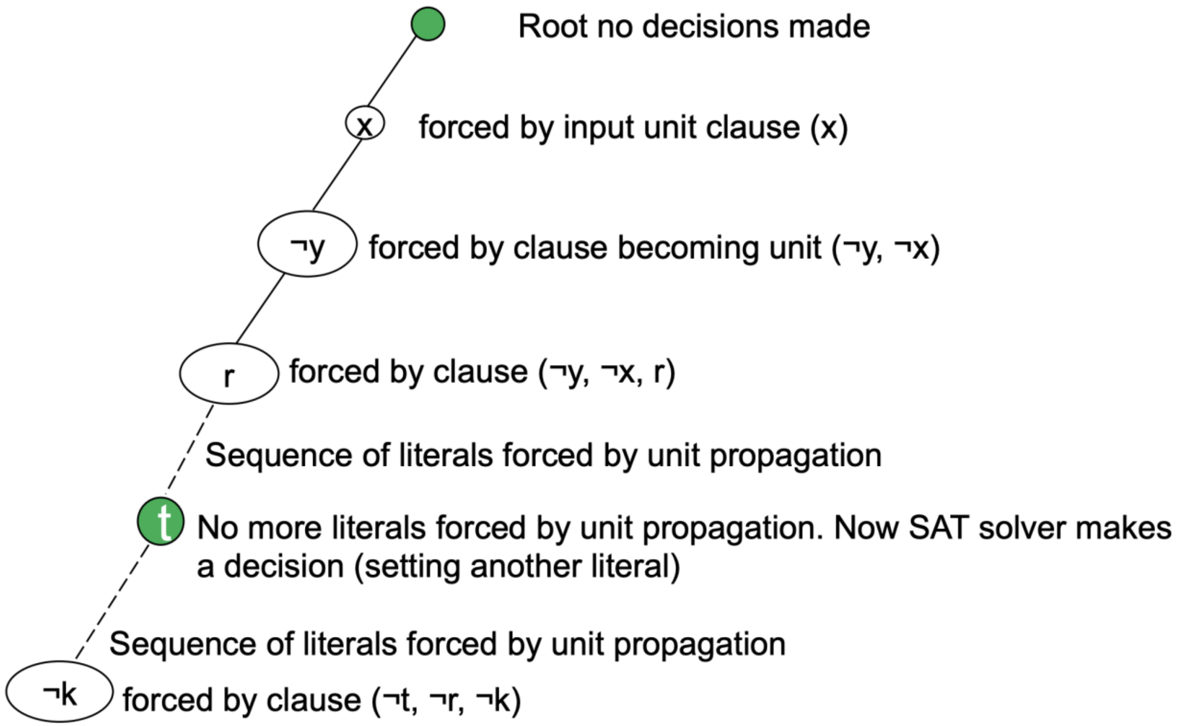


- UIP (Unique Implication Point)
  - A vertex in the implication graph if all the paths from the latest decision literal vertex to the conflict vertex go through it
  - UIP cut: the cut after the UIP
  - First UIP: the closest UIP to the conflict
- Finding the First UIP cut in practice
  - Implication graphs are easy to understand
  - But usually, we record trails instead of explicitly using DAGs

Trails

- A sequence of literals annotated either
  - with the symbol "decided"
  - With a clause that implies the literal

This corresponds to `assignment` in our codebase. You may want to manage it like [(l1, Null), (l2, cls_idx2), (l3, cls_idx3), …} and use `decided_idxs` to figure out the decision level of a decided literal



Conflict Analysis (Pseudocode)

- C = antecedent of the conflict — # starting with BCP's output
- d = level of conflict
- while C contains > 1 literal at d: — # repeatedly applying resolution
  - (l, cls) = pop(trail)
  - if –l in C: — # i.e., l in cls and –l in C
    - C = resolve(clause, C) — # e.g., {A, C} = resolve({A, B}, {-B, C})
- return (C, second highest decision level in C)

Since every resolution step replaces a literal by literals falsified higher up the trail, we must eventually achieve this condition

By construction, c is unit at this level (since it has only one literal at the current level d), and can ensure the next BCP works

An example is in P30 of L4

Other Components

- Deciding by VSIDS
  - Key idea: select the literal that appears most frequently over all the clauses
  - Data structure: `vsids_scores` is a dictionary mapping from literal to score
- Backtracking
  - Key idea: remove all lits. from trail with higher decision level
  - Data structure: `decided_idxs` is a list of indices of the decided literals in `assignment`

E.g., `decided_idxs[d] = k` tells that `assignment[k]` is a decided literal of level d (zero-indexed)

References

- CSC2512: Advanced Propositional Reasoning by University of Toronto
- CS-E3220: Propositional Satisfiability and SAT Solvers by Aalto University
- Hybrid Methods for Constraint Programming by The Association for Constraint Programming