How to Use/Hack zChaff SAT Solver?

Yinlei Yu
(yyu@princeton.edu)

Outline

- Purpose of the Talk
- zChaff Algorithm Revisited
- Using zChaff
- zChaff code explanation
- Summary

Purpose for this talk

- Not a formal explanation of zChaff
- A guide on how to use zChaff
- Break down the zChaff code
- Point out places you need to modify
- Provide some details of code that you may ignore

zChaff Algorithm Revisited

- Flat clause database
- Two literal watching scheme
- VSIDS decision strategy
- Conflict clause generation
- Non-chronicle backtracking

Using zChaff

- Invoking zChaff in command line
  - Basic format:
    - `zchaff –t time_limit CNF_file.cnf`
  - CNF file is representing a Boolean formula in conjunctive normal form
  - zChaff can find a satisfying assignment for the CNF file or prove that such assignment doesn't exist.

CNF File format

- Comments – a line start with ‘c’
- Prelude: A line ‘`p cnf` num_of_vars num_of_clauses’ indicate the start of the CNF content
  - num_of_vars is the number of variables used in the formula
  - num_of_clauses is the number of clauses used in the formula

CNF Format (cont'd)

- Variables and Literals
  - Variables in CNF are expressed as numbers from 1 to num_of_vars
  - Literals: variables or their inverse.
    - The inverse of a variable is expressed as the negation of the number for the variable.
    - e.g.: a 6 means an occurrence of variable x6 and a -6 means an occurrence of variable x6'

CNF Format (Cont'd)

- Clauses
  - A Clause is a line of literals separated by space and ended with a 0
  - Example:

1 6 -10 3 -9 -2 0

(x1+x6+x10'+x3+x9'+x2')

- A line with a single 0 denote the end of the CNF file.

Using zChaff with interface functions

1. Initialize a SAT solver.
   - SAT_Manager SAT_InitManager(void);
2. Set the maximum CPU time (in seconds)
   - void SAT_SetTimeLimit (SAT_Manager mng, float runtime);
3. Set the number of variables
   - void SAT_SetNumVariables(SAT_Manager mng, int num_vars);

Using zChaff with interface functions

4. Iteratively add clauses until all the clauses are added
   - void SAT_AddClause (SAT_Manager mng, int *clause_lits, int num_lits);
5. SOLVE IT!! Get result from the return of SAT_Solve.
   - int SAT_Solve (SAT_Manager mng);
6. Check the result, if satisfiable, obtain the satisfying assignment
   - int SAT_GetVarAsgnment (SAT_Manager mng, int v_idx);
7. Free the SAT solver instance
   - void SAT_ReleaseManager(SAT_Manager mng);

Using zChaff with interface functions

- All the interface functions are in SAT.h while some tuning functions are in chaff_tune.h (Currently) don’t need to use chaff_tune.h
- Check sat_solver.cpp as an example of using interface functions

Overview of zChaff Code

- Several vital parts
  - Variables and clauses representation and access
    - zchaff_base.h
  - Clause Database management
    - CDatabase (zchaff_dbase.h, zchaff_dbase.cpp)
  - Core Solver
    - CSolver (zchaff_solver.h zchaff_solver.cpp)

Overview of zChaff Code

- Core Solver break down
  - Preprocessing
    - preprocess()
  - Core DPLL loop
    - real_solve()
  - Decision
    - decide_next_branch()
  - Binary Constraint propagation
    - deduce(), set_var_value()
  - Conflict Analysis
    - analyze_conflicts() conflict_analysis_firstUIP()
  - Periodic tasks
    - run_periodic_functions() (run before each decision)

Overview of zChaff Code

- Core Execution Loop (simplified)
  - preprocess();
  - while(1){
  - run_periodic_functions();
  - if (decide_next_branch())
  - while (deduce()==CONFLICT) {
  - if (analyze_conflicts()< 0)
  - return= UNSATISFIABLE;}
  - else return SATISFIABLE;}

Clause and variable management

- Clauses are implemented as an STL vector of class CClause
- Each clause has an index, a pointer to its first literal.
- Variables are implemented as an STL vector of CVariable
- Each variable has two watch lists (for two literal watching), assigned value(value()) decision level (dlevel(),-1 for undecided)
- The contents of clauses and variables are accessible by functions

Clause and variable management

- Literals in all clauses are put in a linear array of CLitPoolElement (an 32bit integer, in fact).
- A normal literal is expressed as bit fields like below
  - Bit 2(s) is the sign of the literal
  - The two w's stands for literal watching and direction for searching
  - Bit 30-3 are the variable ID of the literal
- A negative number is the negation of the clause ID

0	var id	s	w	w
31	30 … 3	2	1	0

- You can access literal contents with methods in class CLitElement
- s_var() is a literal which is composed as 2*var_id + sign

Database Management (CDatabase)

- The clause database will grow when conflict clauses are added
- Literals of a clause is cleared to zero and the clause index is invalidated and put into a free-list when the clause is being deleted. (mark_clause_deleted)
- Database compaction is done whenever the database need to expand, the clause pointers, watches, etc. will be adjusted accordingly (compact_lit_pool)

Implication Queue

- A queue that stores the assignment of variables to be applied on the formula
- Written in zchaff_impqueue.h but you can just consider it as a normal queue.
- queue is empty -> make decision -> enqueue the assignment
- Queue is not empty -> dequeue the top assignment, implicate the assignment -> get more implications into the queue
- Conflict in implication -> clear the queue, conflict analysis, backtrack

Preprocessing

- Run before actual solving
- Current preprocess
  - If a variable has ever used in the formula, if no, assign it as 0
  - If a variable occur only in itself or only in its inverse form, but not both, pick an assignment that satisfy this literal

Decisions

- A score on every positive and negative form of variables is maintained.
- A literal's score is increased by 1 when a new conflict clause contain such a literal
- Literal scores are halved after a period of time
- A sorted list of the scores is maintained and updated when a new clause is introduced (CSolver::ordered_vars)
- The top unassigned literal is selected as decision
- A Satisfiable assignment is found if no such unassigned variables exists
- You can write your own decision procedure!!!!

Decision Levels and Assignment Stack

- Decision Level (dlevel()) is the # of unconstrained decision
- (*_assignment_stack)[dlevel] record the decision and all its implications on dlevel in the sequence of actual implication.
- (*_assignment_stack)[dlevel][0] is the s_var of the decision.
- A implication at (*_assignment_stack)[dlevel][n] rely on the assignments in previous dlevel or the implications on (*_assignment_stack)[dlevel][i] i<n

Binary Constraint Propagation

- The deduce() keep popping from the implication queue
- No conflict will be popped from implication queue (unless incremental SAT is running).
- Implication by two literal watching (Read paper!)
  - set_var_value_current_dl
  - Conflicts will enqueue into _conflicts

Conflict Analysis and backtracking

- Check the first conflict
  - analyze_conflicts()
- Find the FirstUIP cut on the conflict's implication graph (DFS traversal mark_vars_at_level)
- Decide the backtrack level
  - conflict_analysis_FirstUIP()
- Forced implication is put into implication queue

Periodical Functions

- Decay variable score periodically
  - decay_variable_score()
- Delete old and unuseful conflict clauses
  - delete_unrelevant_clauses()
- You can add more!!!

Summary

- A chaff for a user’s perspective is presented
- The internal structure of chaff code is described.

Note

- Send me email at yyu@princeton.edu or come to C305 EQuad in case of questions (609-258-7143)