CS 264A - Automated Reasoning: Theory & Applications

Project Report | Team Members: Saswat Padhi [304551256] | 7th June 2015

Design

- Data structures used:
  - FreeBSD implementation of linked `LIST` from `sys/queue.h`
  - Custom `ARRAY` implementation in `sat_api.h`
- Details of structures::
  - (`Var`) Variable contains:
    - 2 literals (not pointers), and an unsigned integer id
    - a decision structure containing:
      - level when the variable was instantiated
      - boolean value if it is instantiated
      - a pointer to the implicant clause due to which the variable was implied. If it was decided, then this clause is set to `NULL`.
      - a pointer to another variable, which is the immediate dominator from the last decision to contradiction. This field is only computed and used during during UIP computation
      - an integer indicating the topo-sort order of the variable in the implication graph
  - (`Lit`) Literal contains:
    - pointer to parent variable and an unsigned integer id
    - array of pointers to original clauses in which the literal appears
    - list of pointers to learned clauses in which the literal appears
    - list of pointers to clauses in which the literal is being 'watched'
  - (`Clause`) Clause contains:
    - unsigned integers id and assertion level (used only for learned clauses)
    - array of literals appearing in the clause
    - 2 pointers to the literals being watched
    - a boolean indicating if the clause has been subsumed
  - (`SatState`) State of the SAT solver contains:
    - unsigned integer level
    - a phantom variable indicating contradiction (for implication graph)
    - a false clause indicating UNSAT-ness. This is learned, if the knowledge base is inconsistent. It has assertion level 0, so backtracking till level 1 would still not resolve the conflict; resulting in UNSAT
    - array of variables in the knowledge base
    - array of original clauses in the knowledge base
    - array of literals that have been decided or implied
    - array of literal waiting to be propagated by unit resolution
    - list of learned clauses
    - list of subsumed clauses

Invariants

- For every parent variable in the SatState's decided literals array, decision level MUST be > 0
- The 2 watched literals always remain synced to the watch lists of those literals
- For every literal waiting in SatState's propagation array, the literal MUST already be implied
- For every clause in the SatState's subsumed clauses list, subsumed flag MUST be set
- For every learned clause, the literals appearing in the clause should have a pointer to the clause in their learned list.
- Any literal inserted into the SatState's decided literals array must be implied by the previous decided literals – i.e. the order is topological, not arbitrary.

Unit Resolution

Unit Resolution iterates over all literals in SatState's array of literals waiting for propagation; and calls `set_Lit_Decision` on them.

The function `set_Lit_Decision` sets the following:

- all clauses (original and learned) in which the literal appears are subsumed. Note that this does not just iterate on the literal's watch-list. It iterates on **all** the clauses which contain the literal. [*]
- all clauses in the literal's negative's watch-list are:
  - updated with new watch literal if it exists
  - if any of the literals has been asserted, clause is subsumed
  - if the other watched literal has been resolved, we have an empty clause – resolution fails
  - otherwise, the other watched literal is asserted, and is added to SatState's propagation array

Unit resolution fails if a literal and it's negative are both attempted to be set. In that case, instead of the negative literal, the contradiction literal is inserted into the array of decided literals, with conflicting clause as implicant.

[*] Subsuming **all** clauses of a literal immediately instead of just the watched list is necessary to ensure that the subsumption check for the clause is efficient, and for efficiently undoing subsumed clauses (as explained below).

Some optimizations I have made here is:

- Literals are immediately set first and then added to queue for further propagation, rather than setting and propagating in the queued order. This showed significant performance boost.
- I also tried FIFO vs LIFO propagation schemes (queue vs stack). FIFO performed much better than LIFO.

When unit resolution is undo'ed, we need to revert all the clauses that were subsumed and all variables that were set. I achieve this efficiently by:

- Removing all literals from SatState's decided literals array; which have decision level as current level. There's no memory overwritten here. I just decrement the count of the array.
- Because we check all clauses for subsumption every time a literal is asserted, the only clauses that were subsumed are because of implications are current level. Instead of storing and checking clauses' subsumption levels, I use a simple trick – I push a NULL value on to the list of subsumed clauses whenever I increment the SatState's level. So when I undo, I keep popping clauses off the subsumed list till NULL. Note that I did not use the same trick for decided literals array because it would have disturbed the topological ordering.
- The watches and watch lists need not be touched at all! :-)

Clause Learning

I have implemented the first-UIP method for learning asserting clauses in case of conflicts. One of the advantages I exploit with my design is the way I arrange decided literals in the array. At any given point, it would look like:

```
(a, 1, NULL) (-b, 1, 3) (f, 1, 2) (c, 2, NULL) (-e, 2, 4) ...
```

The triplet `(x, l, c)` is the literal struct:

`x` is the variable (- for negative)
`l` is the level at which the literal was asserted
`c` is the implicant clause, `NULL` if the literal was decided

One property of this list is, if Y appears after X in the list, then Y would appear after X in the topological ordering of the implication graph. This is because, deciding X first led to the implication of Y. So, I already have a topological ordering of the nodes of the graph in the decided literals array!

And instead of having multiple edges in the graph, I just store the implicant clause which contains the information about all the edges. Once we have a conflict, I first compute the first UIP in the implication graph which essentially is - computing the immediate dominator of the contradiction node in the subgraph containing all implications at the current level only.

Obtaining this subgraph is simple: one has to simply check if a node's level is same as current level. Once we have the subgraph, the dominator tree is computed using a simple union-find like algorithm:

```
common_dom(var1, var2):
  while(var1 != var2) {
    while(var1's order < var2's order) var2 = var2's dominator
    while(var2's order < var1's order) var1 = var1's dominator
  }
  return var1
```

```
UIP(SatState):
  dom[last decided variable] = last decided variable  // root of current-level subgraph
  for literal l in SatState's decided literals, after the last decided variable:
    // l iterates over literals implied in current level, in topological order
    for literal p in l's implicant clause:
      // p is l's predecessor
      lv = l's parent variable
      lp = p's parent variable
      if(dom[lv] is NULL)     dom[lv] = pv
      else                    dom[lv] = cdom(dom[lv], pv)
  return dom[contradiction]
```

After first-UIP computation, a cut is implemented by marking all the literals before the current level, which affect implication of literals in the current level. Assertion level of the learned clause is the highest decision level of these marked literals.

We know that learned clause would be unit when backtracked to assertion level, so one of the watch literals should be the UIP. I assign the other watch literals as the one that was decided at the assertion level.

Evaluation

Evaluation environment:

- Ubuntu 15.04 running on 64-bit Intel i7 4790 @ 3.6GHz (8MB SmartCache) & 16GB DDR3 Physical Memory
- Invocation: `c2D -c <file> -i -C -E`
- Time Limit = 3600 seconds & `T_base` is the total time from provided `c2D` implementation

Path (sampled/*)	| Mods(Δ) |	NNF Nodes	NNF Edges	T_base	T
2bitcomp_5.cnf	9840070722846720	162271	324192	1.057	1.060
medium.cnf	2	310	350	0.034	0.038
2bitmax_6.cnf	206829646435704880299088281600	3582489	7164210	30.772	32.537
uf250-017.cnf	-	-	-	TIMEOUT	TIMEOUT
uf250-026.cnf	-	-	-	TIMEOUT	TIMEOUT
C169_FW.cnf	3216989843619840	3095	3352	0.102	0.106
par16-1-c.cnf	1	633	632	0.137	40.606
par16-5-c.cnf	1	681	680	0.361	8.468
C250_FW.cnf	12047082580849261441833041796217700352	3705	4412	0.167	0.164
par16-2-c.cnf	1	697	696	0.352	4.371
tire-2.cnf	738969640920	13920	26392	0.344	0.348
tire-3.cnf	222560409176	27096	52630	0.645	0.683
cnt06.shuffled.cnf	1	1523	1522	0.627	1.569
ais10.cnf	296	29106	57488	2.649	115.079
C638_FVK.cnf	88271038388920812991420333220858974270038282631635257463703238139720235683038426809311735838694756426058714576694510878720	16290	28296	0.705	0.718
C211_FS.cnf	13762971811969386785872966835451223602449242343997055595165673062400	43050	82720	2.037	2.098
tire-4.cnf	103191650628000	64725	127290	2.155	2.187
ssa7552-038.cnf	28432833270798238107452185066189558382592	42707	79880	1.636	1.620
log-1.cnf	564153552511417968750	7029	11610	0.372	0.384
C171_FR.cnf	140512943652875954022283516249114329924007324094159428510106846665786562155390254260936140177065067255638517075673088	386292	768898	17.299	17.388
C210_FVF.cnf	301614289630681912033263984906544585332936879548933341265926004667161054639762155016043295109296623635226045945224867027826253455446929950648621026580647248629989250170880	2269570	4535348	102.466	96.024
par16-3.cnf	1	2029	2028	0.941	9.896
par16-5.cnf	1	2029	2028	0.788	10.132

Path (sampled/*)	| Mods(Δ) |	NNF Nodes	NNF Edges	T_base	T
par16-2.cnf	1	2029	2028	2.912	407.592
bw_large.a.cnf	1	917	916	0.631	0.661
C230_FR.cnf	3260832604367732895211617383190998941215121285134459272745077800497933182264564635334744947464083370477203064883688221437132026603896832	4466781	8929650	212.068	214.123
C215_FC.cnf	1855919107367037613983708599926441261440144615046475387203475349266106306906877503694181606516439831750695566978458225745409220894351510814456857755648	18425327	4934723	106.873	109.476
C163_FW.cnf	2967690652313618572821236005090928122016075578011504993093346803329801339054210965500348932304629584166525808139008559495667935969129424448640	1843380	3682750	112.686	113.414
huge.cnf	1	917	916	0.948	0.989
C638_FKA.cnf	220791349509536943869709461636899587488282386554415799773324795630458955101317332898512977946164174872108977446775762516652851200	941113	1877778	95.056	98.728
qg3-08.cnf	18	8245	14976	6.005	6.269
ra.cnf	18739277038847939886754019920358123424308469030992781557966909983211910963157763678726120154469030856807730587971859910379069462105489708001873004723798633342340521799560185957916958401869207109443355859123561156747098129524433371596461424856004227854241384374972430825095073282950873641	499941	994938	62.663	64.009
bw_large.b.cnf	2	2790	3368	6.370	12.389
prob004-log-a.cnf	-	-	-	TIMEOUT	TIMEOUT
qg6-09.cnf	4	3776	5646	34.893	34.277
qg7-09.cnf	4	3623	5340	31.863	33.187
log-2.cnf	32334741710	3286966	6569698	697.736	1067.124
4blocksb.cnf	4	1972	3012	27.238	45.962
log-3.cnf	279857462060	521343	1038450	204.490	424.067
qg1-07.cnf	8	3043	5092	148.300	140.749
qg2-07.cnf	14	4976	8918	155.845	150.303
Σ				**2072.2**	**3268.8**

Zero memory leak in all test cases, evaluated with valgrind.