SAT Solving and its Relationship to CSPs

Fahiem Bacchus
University of Toronto

Overview

- Tremendous gains have been achieved over the last 5 years in SAT solving technology.
- Systematic backtracking based systems have become the best method for solving structured SAT instances.
- New theoretical insights have been gained into the behaviour of backtracking SAT solvers via their close relationship with resolution proofs.
- These insights have direct relevance to finite domain CSP solvers.

Resolution Proofs

All (complete) backtracking algorithms for CSPs are implicitly generating resolution proofs.

On problems with solution, still have to backtrack out of failed subtrees.

Baker (1995), Mitchell (2002)

Resolution (SAT)

- A complete proof procedure for propositional logic that works on formulas expressed in conjunctive normal form. (Robinson 1965)
- Conjunctive Normal Form (CNF)
  - Literal: a propositional variable p or its negation ¬p
  - Clause: a disjunction of literals (a set).
  - CNF theory: a conjunction of clauses.

Resolution

- From two clauses (A, x) (B, ¬x) the resolution rule generates the new clause (A, B), where A and B are sets of literals.
  - (A,B) is the resolvant.
  - x is the variable resolved on
  - duplicate literals are removed from the resolvant
  - denote by C = **R**(C_1,C_2)

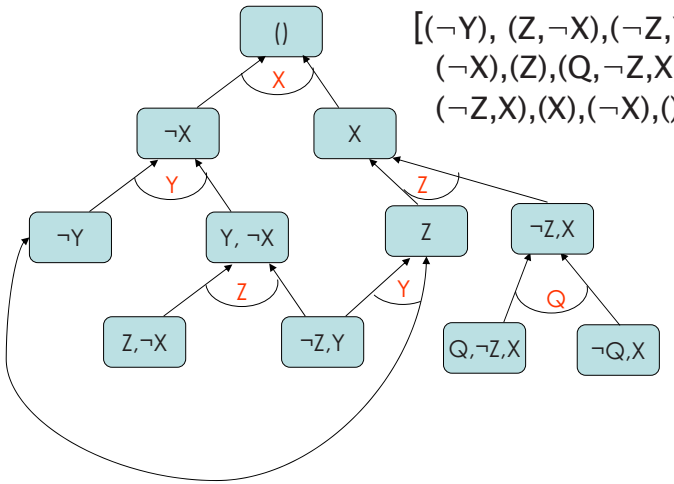
Resolution

- A resolution refutation π of a CNF theory $\mathcal{F}$ is a sequence of clauses C_1, C_2, …, C_m such that
  - each C_i is either a member of $\mathcal{F}$ or
  - C_i is a resolvant of two previous clauses in the proof: $C_i = R(C_j,C_k)$ $j,k < I$
    - Clauses arising from resolution are called the derived clauses of π.
  - C_m = () the empty clause.
- π is also called a resolution proof.
- The SIZE of π is the number of resolvants in it.

Resolution DAG

- Any resolution proof can be represented as a DAG.
  - nodes are clauses in the proof.
  - Every clause Ci that arises from a resolution step has two incoming edges. One from each of the clauses that were resolved together to obtain Ci.
  - The arcs are labeled by the variable that was resolved away to obtain Ci.
  - Clauses in $\mathcal{F}$ have no incoming edges.

Resolution Dag



[(¬Y), (Z,¬X),(¬Z,Y),(Y,¬X), (¬X),(Z),(Q,¬Z,X),(¬Q,X), (¬Z,X),(X),(¬X),()]

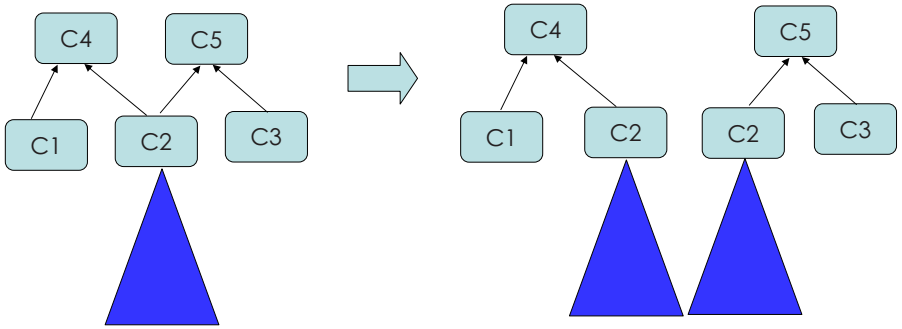
Restrictions of Resolution

- A number of restricted forms of resolution can be defined, where, e.g., we require the DAG to be a tree.
- The reason the restricted forms have been developed is that the restrictions can make it easier to find a proof.

Tree Resolution

- Tree resolution
  - The DAG is required to be a tree.
    - ➔ Clauses derived during the proof can only be used once.
  - Work must be duplicated to rederive clauses that need to be used more than once.

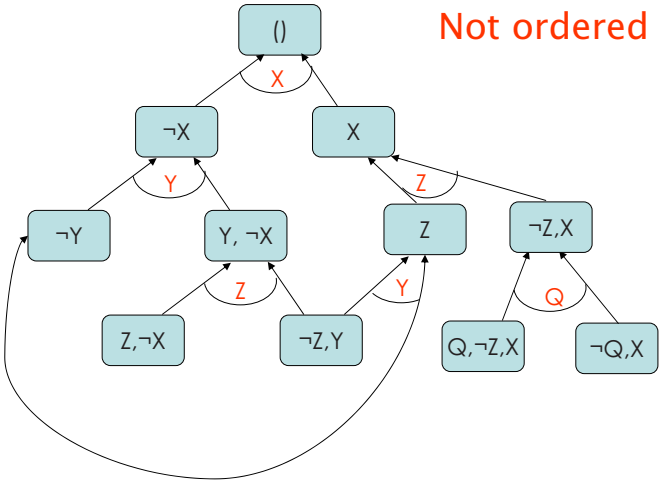
Tree Resolution



Ordered Resolution

- The variables resolved on along any path in the DAG to the empty clause must respect some fixed ordering of the variables.

Ordered Resolution

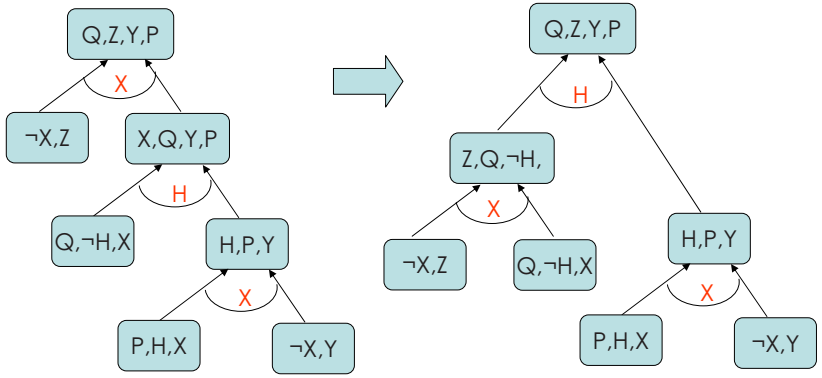


Not ordered

Regular Resolution

- Along any path in the DAG to the empty clause the sequence of variables resolved away cannot contain any duplicates.

Regular Resolution



Not Regular

Regular

Negative Resolution

- One of the clauses in each resolution step must contain only negative literals. (a negative clause)
  - This is complete!
  - Note $\mathcal{F}$ must contain at least one negative clause else the "all true" truth assignment is a satisfying model.

Relative Power

- A general formalism for comparing the power of different proof systems was developed by Cook and Reckhow 1997.
- For our purposes we simply look at the minimal size refutation proof (the number of clauses in the proof).

Relative Power

- #R(F)—the minimal size R-refutation of $\mathcal{F}$ among all possible R-refutations of $\mathcal{F}$.
- For a family of formulas $\mathcal{F}_i$ we look at how #R($\mathcal{F}_i$) grows with i.
- Let S and T be two restrictions of resolution. S p-simulates T if there exists a polytime computable function f such that:
  - For any S-refutation π of a formula $\mathcal{F}$, f(π) is a T-refutation of $\mathcal{F}$.
  - Note that this means that f(π) can't be more than polynomially longer than π ➔ #T(F) no more than polynomially larger than #S(F) for any formula F.

Relative Power Known Results

- Buresh-Oppenhiem, Pitassi (2003) many new results and a summary of previously proved results.

Relative Power Known Results

	Regular	Negative	Ordered	Tree
Regular		No	Yes	Yes
Negative	No		No	Yes
Ordered	No	No		No
Tree	No	No	No	

- Regular always yields shorter proofs than either Ordered or Tree
- Negative and Regular are incomparable
- Ordered and Tree are incomparable.

Relative Power Known Results

- It is also known that none of these restrictions can p-simulate general resolution.

Solving Sat

DP & DPLL (DLL)

- Two earliest algorithms for solving SAT actually predate resolution.
- DP: Davis-Putnam (1960) a variable elimination technique.
- DPLL: Davis-Logemann-Loveland (1962) a backtracking search algorithm.

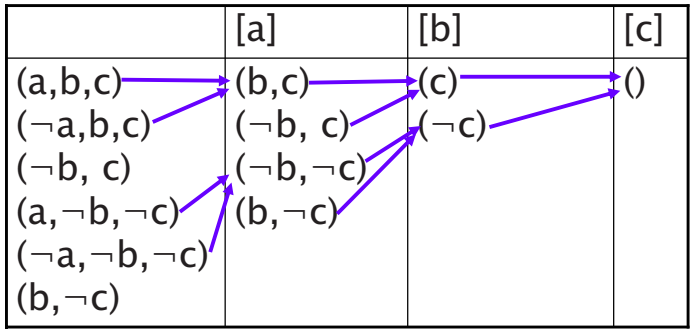
DP

- Pick a variable ordering (one that has a low elimination width if possible): X1, X2, …, Xn
- Starting with the original set of clauses $\mathcal{F}$
- At the i-th stage:
  - Add to $\mathcal{F}$ all possible resolvants that can be generated by resolving on Xi.
  - Remove from $\mathcal{F}$ all clauses containing Xi or ¬Xi.
  - If the empty clause is generated stop
- The input set of clauses (the formula $\mathcal{F}$) is UNSAT iff this process generates the empty clause.

DP

	[a]	[b]	[c]
(a,b,c) (¬a,b,c) (¬b, c) (a,¬b,¬c) (¬a,¬b,¬c) (b,¬c)	(b,c) (¬b, c) (¬b,¬c) (b,¬c)	(c) (¬c)	()

DP



Potentially many redundant clauses are generated, but an ordered resolution is contained in these clauses.

DP

- Every DP proof contains an ordered resolution, and thus it can never be shorter than an ordered resolution refutation.
  - Note lower bounds are wrt any possible ordered resolution (i.e., any ordering).
- In practice, DP's space requirements are prohibitive
  - Although some attempts using ZBDDs to represent the clauses compactly.
  - Still not competitive with current best techniques.

DP⇔Ordered Resolution

- Note also that every ordered resolution can be found inside of a DP refutation:
  - just follow the same order.
  - Since DP generates all possible ordered refutations along that order, it might terminate before completing the specified ordered refutation (by finding a shorter ordered refutation).
  - DP can also waste a lot of time generating clauses that are not needed for the refutation.

DPLL

- Developed shortly after DP, DPLL is based on backtracking search. The connection to resolution was realized later.
  - One picks a literal (a true or false variable)
  - simplify the formula based on that literal
  - recursively solve the simplified formula.
  - if the simplified formula is UNSAT, try using the negation of the literal chosen.

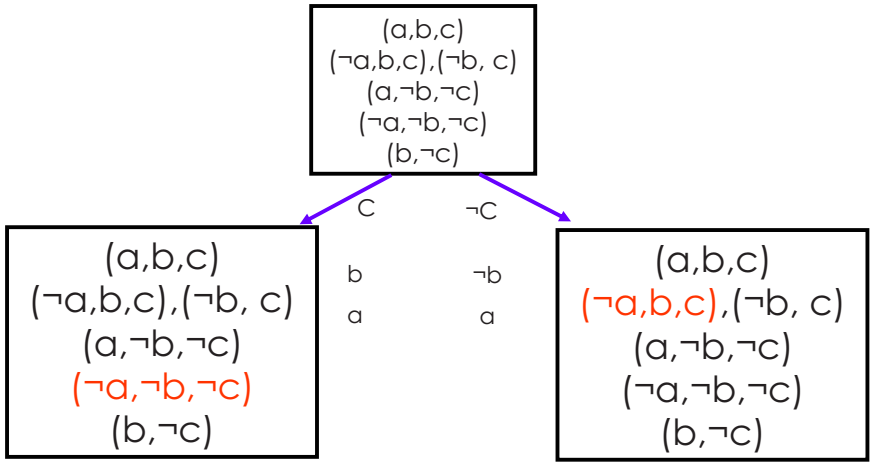
DPLL Simplification

- Given a clausal theory $\mathcal{F}$, we can simplify it by a literal ℓ as follows:
  - $\mathcal{F}_{|\ell} =$
    - Remove from $\mathcal{F}$ all clauses containing ℓ
    - Remove ℓ from all of the remaining clauses.

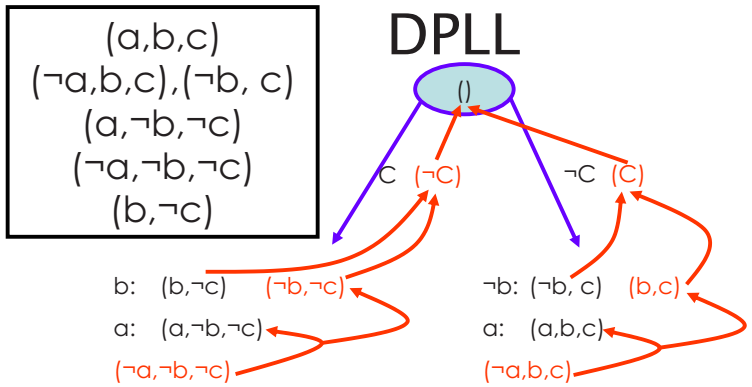
Unit Propagation

- In addition DPLL employs unit propagation:
  - if $\mathcal{F}_{|\ell}$ contains any unit clauses, e.g. ($\neg$x) then further simplify $\mathcal{F}_{|\ell}$ by the literal in the unit clause, i.e., generate $(\mathcal{F}_{|\ell})_{|\neg x}$
  - Unit propagation is the iterative application of this simplification until the resultant theory has no unit clauses (or contains the empty clause).
- More powerful forms of propagation examined in Bacchus (2002)

DPLL



DPLL



- A tree refutation is embedded in every DPPL proof of UNSAT.
  - every resolvant consists of literals negated by the prefix of assignments.

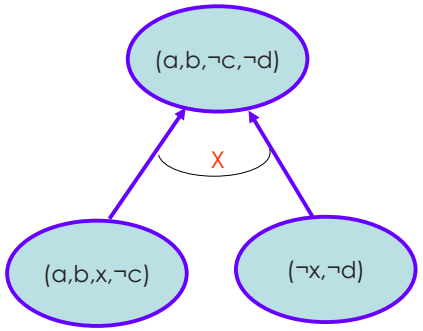
DPLL

- Every DPLL proof contains an tree resolution, and thus it can never be shorter than an tree resolution refutation.
  - Note it need not be ordered. So the minimal size DPLL tree can be bigger or smaller than the minimal size DP proof.

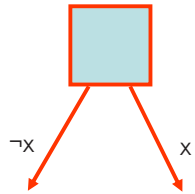
DPLL⇔Tree Resolution

- Note also that every tree resolution can be found inside of a DPLL refutation:
  - Make the DPLL search mimic a depth first search of the tree refutation.
    - always instantiate the negation of the literal that was resolved on in the child node.

DPLL⇔Tree Resolution



Node of tree resolution



Corresponding node of DPLL search

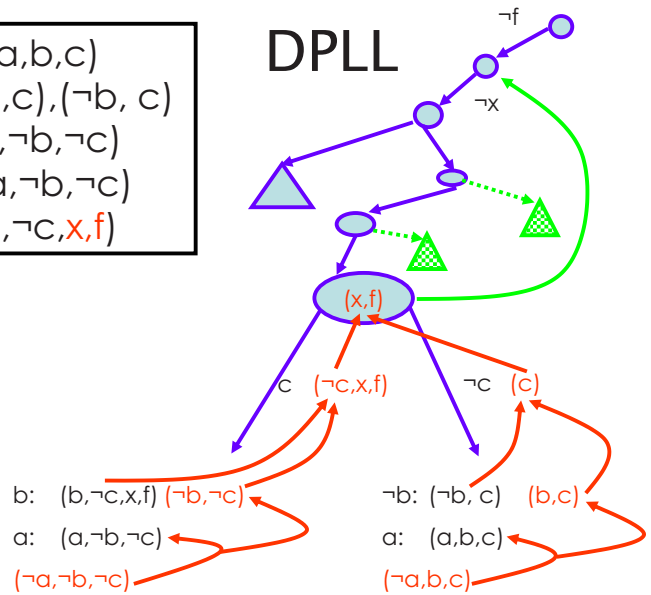
DPLL⇔Tree Resolution

- In general DPLL search will also do a lot of extra work not required for the tree resolution since it did not employ intelligent backtracking.
- Hence, DPLL in its original form is a pretty poor algorithm. Although is "reasonable" for proving UNSAT for random problems.

Resolution and Intelligent Backtracking

- If we keep track of the refutation being generated, we can use the derived clauses to perform intelligent backtracking.
- Keeping track of the resolution refutation is precisely what CBF (conflict directed backjumping) does.

(a,b,c)
(¬a,b,c),(¬b, c)
(a,¬b,¬c)
(¬a,¬b,¬c)
(b,¬c,x,f)



Resolution and Intelligent Backtracking

- If instrumented to keep track of the resolution refutation and thus perform "intelligent backtracking" (non-moronic backtracking), it can find tree resolutions fairly effectively.
- Still some inefficiencies
  - can spend time in subtrees that don't contribute to final refutation.
- Still limited to relatively weak tree resolution.

Resolution and Intelligent Backtracking

- Modern Techniques move DPLL beyond the limited power of tree resolution.

Solving Finite Domain CSPs

Translation to Propositonal Logic

- Set of variables V_i and constraints C_j
- Each variable has a domain of values $Dom[V_i] = \{d_1, \ldots, d_m\}$.
- Consider the set of propositions $\blacktriangleleft V_i=d_j \blacktriangleright$ one for each value of each variable.
  - $\blacktriangleleft V_i=d_j \blacktriangleright$ means that V_i has been assigned the value d_j.

Translation to Propositonal Logic

- $\neg \blacktriangleleft V_i=d_j \blacktriangleright$ means that V_i has not been assigned the value d_j
  - perhaps not been assigned any value, or has been assigned a different value.
  - True when d_j has been pruned from V_i's domain.

Translation to Propositonal Logic

- For simplicity
  - Write $V_i=d_j$ instead of $\blacktriangleleft V_i=d_j \blacktriangleright$
  - $V_i \neq d_j$ instead of $\neg \blacktriangleleft V_i=d_j \blacktriangleright$
  - But be aware that these are actually propositional variables that can be assigned true or false.

Translation to Propositonal Logic

- Each constraint C is over some set of variables $X_1,\ldots,X_k$: $C(X_1,\ldots,X_k)$
- Typically a constraint is defined to be a set of tuples of assignments to its variables that satisfy the constraint.
- Equivalently, we look at the complement
  - The set of tuples of assignments that falsify the constraint.
    - E.g., $(X_1=a,X_2=b,\ldots,X_k=h)$ falsifies $C(X_1,\ldots,X_k)$

Translation to Propositonal Logic

- A falsifying tuple is typically called a nogood: a set of assignments that cannot be extended to a solution of the CSP.
  - If the tuple falsifies a constraint of the CSP, it can't be extended to a solution of the CSP.
- Nogoods are clauses.
  - A nogood $(X_1=a,X_2=b,\ldots,X_k=h)$ asserts
    - $\neg(X_1=a \wedge X_2=b \wedge \ldots \wedge X_k=h)$
    - $\rightarrow (X_1\neq a \vee X_2\neq b \vee \ldots \vee X_k\neq h)$ (a clause).

Translation to Propositonal Logic

- So each constraint is a set of clauses.
- All of the constraints of the CSP thus form a set of clauses.

Translation to Propositonal Logic

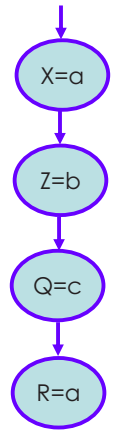
- Finally, we must deal with the fact that the variables have non-binary domains.
- For each variable V with Dom[V]={d_1,…,d_k} we obtain the following clauses:
  - ($V=d_1$,$V=d_2$,…,$V=d_k$) (must have a value)
  - ($V\neq d_1$,$V\neq d_2$), ($V\neq d_1$,$V\neq d_3$), …, ($V\neq d_1$,$V\neq d_k$) ,…, ($V\neq d_2$,$V\neq d_3$), …, ($V\neq d_{k-1}$,$V\neq d_k$) (must have a unique value)

FC

- For simplicity look at Forward Checking, and we will see that
  - embedded in a failed FC search tree is a tree resolution.
  - Keeping track of the resolution refutation gives us CBJ.
  - The resolution also makes the improvement of backpruning (Bacchus 2000) obvious.

FC

- FC maintains node consistency.
  - when a constraint becomes unary (all but one of its variables have been instantiated), we enforce node consistency on that constraint to prune the domain of the sole remaining variable.
  - This definition works with both binary and n-ary constraints.

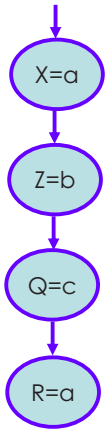


Domain wipe-out of V

FC

- Each value of V was removed because it falsified some nogood from some constraint.

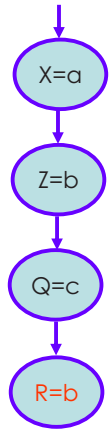
FC



Domain wipe-out of V

- Dom[V] = {a, b, c}
  - (V≠a,X≠a)
  - (V≠b,R≠a,X≠a)
  - (V≠c,R≠a,X≠a)
- Resolving these against (V=a,V=b,V=c), we obtain the new clause (X≠a,R≠a): a clause containing the current value of R.
- FC now backtracks and tries a different value for R.

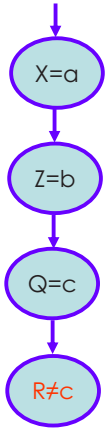
FC



Domain wipe-out of Y

- Dom[Y] = {a,b,c}
  - (Y≠a,X≠a,Z≠b)
  - (Y≠b,R≠b)
  - (Y≠c, X≠a,R≠a)
- Resolving these against (Y=a,Y=b,Y=c), we obtain the new clause (X≠a,Z≠b,R≠b)
- Again we backtrack and try a different value for R.

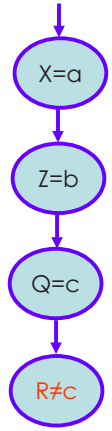
FC



Domain wipe-out of Y

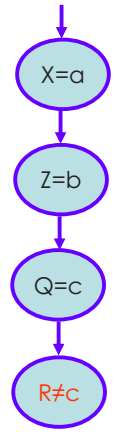
- Perhaps R=c has already been pruned by FC before we reached this node.
- Then there is a clause forcing by R≠c, e.g.
  - (Z≠b,R≠c)
- Now we have a clause forcing the removal of each of R's values
  - Either computed via resolution from the subtree below that assignment.
  - Or from forward checking above.

FC



Domain wipe-out of Y

- Now we have a clause for each value of R:
  - (X≠a,R≠a)
  - (X≠a,Z≠b,R≠b)
  - (Z≠b,R≠c)
- Resolve these against (R=a,R=b,R=c) to obtain
  - (X≠a,Z≠b)

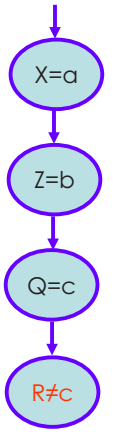


Domain wipe-out of Y

FC-CBJ

- Ordinary FC would then contine with the next value of Q.
- But embedded in each failed subtree of the FC search tree a is a tree resolution.
- FC-CBJ simply keeps track of the resolution refutation, and uses the clause produced
  - (X≠a,Z≠b)

  to backtrack to undo Z=b.



Domain wipe-out of Y

Extended FC

- The clause (X≠a,Z≠b) tells us that we can soundly backtrack to undo Z=b.
- The clauses we learned for the values of R
  - (X≠a,R≠a)
  - (X≠a,Z≠b,R≠b)
  - (Z≠b,R≠c)
- Tell us that we also need not try the value R=a again until we backtrack even further to undo X=a.
- Keeping track of this information allows us to "backprune" values. Bacchus (2000)

Negative Resolution

- Notice the resolution steps involved
  - (R=a,R=b,R=c),(X≠a,R≠a) ➔ (X≠a,R=b,R=c)
  - (X≠a,R=b,R=c),(X≠a,Z≠b,R≠b) ➔ (X≠a,R=c,Z≠b)
  - (X≠a,R=c,Z≠b),(Z≠b,R≠c) ➔ (X≠a,Z≠b)
- Negative resolution steps. (One of the clauses in always negative).
- I.e., FCCBJ actually embeds a negative tree resolution. Even more limited in power. Mitchell (2003)

Negative Resolution

- In fact in the standard techniques all clauses (nogoods)
  - In the original constraints are negative.
  - Learned during search are negative.
- Return to this later.

Modern Sat Solvers

Clause Learning (CL)

- The main feature of modern SAT solvers is the development of new techniques to support effective clause learning
  - Without clause learning DPLL and CSP backtracking algorithms are both limited to tree resolution, (negative tree resolution in the case of CSPs).
  - Modern solvers are N-orders of magnitude faster than the best implementations of standard DPLL on many problems. Where N is probably >6.

DPLL+CL

- DPLL
  - picks a literal
  - reduces the theory with that literal
  - this perhaps induces some sequence of further literals all forced by unit propagation.
  - Stops and backtracks when some clause becomes falsified.

Failed Path

- X
  - A
  - ¬B
  - C
- ¬Y
  - D
  - ¬E
  - F
- Z
  - H
  - I
  - ¬J
  - ¬K

(K,¬I,¬H, ¬F,E, ¬D,B)

- X,Y,Z: Decision Variables.
- A,¬B,C,D,¬E,F,H,I,¬J,¬K: forced by unit propagation
- (K,¬I,¬H, ¬F,E, ¬D,B): Falsified clause. This clause is called a conflict clause: it is falsified by the current path.

Forced Literals

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- Each forced literal was forced by some clause becoming unit.
- We keep track of the forcing clause as part of the unit propagation process.

Forced Literals

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- Each clause reason contains
  - One true literal on the path (the literal it forced)
  - Literals falsified higher up on the path.

Forced Literals

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- Hence we can resolve away any forced literal in the conflict clause.
- This will yield a new conflict clause.

1. (K,¬I,¬H, ¬F,E, ¬D,B), (D,B,Y) ➔ (K,¬I,¬H, ¬F,E,B,Y)
2. (K,¬I,¬H, ¬F,E, ¬D,B), (¬K,¬I,¬H,E,B) ➔ (¬I,¬H, ¬F,E, ¬D,B)
3. (K,¬I,¬H, ¬F,E, ¬D,B), (H,B,E,¬Z) ➔ (K,¬I,¬F,E, ¬D,B,¬Z)
4. …

Conflict Clauses

- Any forced literal in any conflict clause can be resolved on to generate a new conflict clause.
- If we continued this process until all forced literals were resolves away we would end up with a clause containing decision literals only (All-decision clause).
- But empirically the all-decision clause tends not be very effective.
  - Too specific to this particular part of the search to be useful later on.

Conflict Clauses

- Various choices exist as to how to generate a conflict clause on failure.
- The most popular form of clause learning is 1-UIP learning (Zchaff). (Now almost the standard).

1-UIP Clauses

- Start with C equal to the original conflict clause
1. Let n be the number of literals in C at or below the last decision variable.
2. If n > 1
   - Let C be equal to the result of resolving away the deepest forced literal.
   - Goto 1
3. Else store C for future use and use it for backtracking.

1-UIP Clauses

- This process must terminate. As when we resolve away a literal can only introduce literals above it on the path.
- The last remaining literal from the deepest level in the 1-UIP clause may or may not be the decision literal.

1-UIP

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

1. (K,¬I,¬H, ¬F,E, ¬D,B), (¬K,¬I,¬H,E,B)
   ➔ (¬I,¬H, ¬F,E, ¬D,B)
2. (¬I,¬H, ¬F,E, ¬D,B), (I,¬H,¬D,¬X)
   ➔ (¬H, ¬F,E, ¬D,B,¬X)

Backtracking

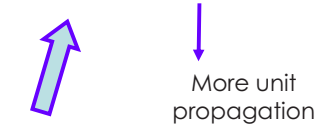
- The advantage of a 1-UIP clause (or any unique implication point clause) is that it forces the single literal from the deepest level.
- We can backtrack to the point that literal is forced and augment the set of forced literals at that level by the new unit prop.

1-UIP

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
  - ¬H ← (¬H,¬F,E, ¬D,B,¬X)



(¬H, ¬F,E, ¬D,B,¬X)

Backtracking

- Note that the decision literal has not been exhausted. We don't know if the current prefix with ¬Z instead of Z might have a solution.

Far Backtracking

- "Far Backtracking", i.e., backtracking to the point we have the new unit implicant instead of backtracking to undo the deepest decision. has two motivations:
  - ¬H is implied at this higher level, so undoing all of the work and starting again is the easiest way to take this constraint into account. (See Bacchus 2000 for an alternate approach to back forcing of backpruning values).
  - Perhaps heuristically it might be better to start the search under the newly discovered implication all over again.

Far Backtracking

- E.g., with far backtracking whenever a unit conflict is discovered, the search returns to the root: a complete restart.
  - Unclear if there is any real empirical evidence about whether or not this is more efficient.

Managing Large number of Clauses

- Once we start learning a clause at every backtrack point, we soon have the problem of having to deal with lots of new clauses.
- The learned clauses often are far more numerous than the input clauses.

Watch Literals

- Some other techniques have been developed in the SAT literature that have made clause learning feasible.
  - More efficient unit propagation by the technique of watch literals.
    - in order for a clause to be come unit all but one of its literals must become false.
    - Assign two watch literals per clause. Only when the watch literal becomes false do we check the clause.
    - Try to find another watch, or determine that the clause has become unit or empty.

Watch Literals

- Like ideas in current GAC algorithms where only a single support is maintained. But no reliance on lexographic ordering. Thus watches have an important benefit of requiring no work on backtrack.
- Also clever empirically tuned techniques for where to locate the watches and how to store clauses in memory designed to maximize cache hits.

VSIDs Heuristics

- Additional success has been obtained from dynamic variable ordering heuristics that are very quick to compute: don't require examining all unassigned variables.
- These heuristics favor literals that have appeared in recently learned clauses.
  - Intuition is claimed to be: learning new clauses that can be resolved against recent clauses.

The Power of Clause Learning

- Beame, Kautz, and Sabharwal (2003) showed that regular resolution cannot p-simulate clause learning.
  - I.e., there exists formulas with short CL proofs but long regular resolution proofs.
- Recently (Pitassi & Hertel, not yet published) have shown that CL can p-simulate regular resolution.
  - I.e., with clause learning DPLL becomes strictly more powerful than regular resolution, and thus a major advance over standard tree-resolution limited DPLL.

The Power of Clause Learning

- We are still working on the question of whether or not CL is as powerful as general resolution.

Improving CSP solvers

Improving CSP Solvers

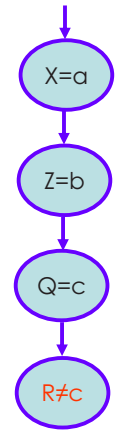
- FCCBJ no more powerful than tree resolution, also bounded by negative resolution.
- How do we gain from advances in SAT?
- Ideas along this line have been pursued by my PhD student George Katsirelos who has written a general CSP solver toolkit called EFC based on our ideas.
  - http://www.cs.toronto.edu/~gkatsi/efc/

Improving CSP Solvers

A. Unrestricted Clause (nogood) learning.
B. Learning non-negative clauses.
C. Improving the input clauses.
D. Better clauses (nogoods) from GAC.
E. Better clauses (nogoods) from propagators.

A. Unrestricted NoGood Learning

- Standard works on clause (nogood) learning has concentrated on various restricted forms, e.g., k-relevance bounded, length bounded, etc.
- Our first attempt was to utilize SAT techniques for the efficient handling of large numbers of clauses (watch literals, clause databases) to allow storing as many nogoods as can fit into main memory.
- With standard techniques are used to learn the nogoods the results were occasionally very good, however often the results were not great.



Domain wipe-out of Y

Negative Resolution FC (reminder)

- With clause for each value of R:
  - (X≠a,R≠a)
  - (X≠a,Z≠b,R≠b)
  - (Z≠b,R≠c)
- Resolve these against (R=a,R=b,R=c) to obtain
  - (X≠a,Z≠b)

Negative Resolution

- Each resolution step is a negative resolution.
- All learned nogoods are negative clauses.
  - Restricts the power of the search to negative resolution.

No Unit Propagation

- Unit propagation over the learned clauses never propagates!
  - (X≠a,Z≠b) when X=a this becomes unit forcing Z≠b (the pruning of b from Z's domain)
  - But now Z≠b can only make other learned clauses true, it cannot make any of them unit.
  - So one never gets very much further than the original clauses.
  - In contrast unit propagation in SAT can often value hundreds of literals after each decision.

B. Learning Non-Negative Clauses

- Idea is simple, when learning a new clause simply don't resolve away all of the positive literals!
- With clause for each value of R:
  - (X≠a,R≠a)
  - (X≠a,Z≠b,R≠b)
  - (Z≠b,R≠c)
- Resolve these against (R=a,R=b,R=c) to obtain
  - (X≠a,R=b,R=c)
  - or (Z≠b,R=a,R=c)
  - instead of doing all of the resolution steps.

B. Learning Non-Negative Clauses

- We developed a first-decision clause learning scheme, where we replace all literals (assignments/non-assignments) in the clause until we have only the decision literal at the deepest level.
- This allows us to learn non-negative clauses, perform intelligent backtracking, and it seems to work better than the 1-UIP scheme in CSPs.

Unit Propagtion

- (Z≠b,R=a,R=c), now if we prune a from R's domain and assign Z=b, this mixed clause forces us to assign R=c.
- That assignment can make other clauses unit, pruning other values or forcing other assignments.

B. Learning Non-Negative Clauses

- "Generalized Nogood" learning often gives dramatic performance improvements over FCCBJ+learning standard nogoods (negative clauses), which in turn is better than FCCBJ without any learning.
- Also provably adds power over standard negative clauses (Katsirelos & Bacchus 2005)
  - See also Hwang & Mitchell (2005) on how 2-way branching also has the potential to get around negative resolution.

C. Improving Input Clauses

- The input clauses are all negative clauses. This also limits the effectiveness of resolution.
- One can generalize these clauses, e.g., say the constraint C(X,Y,Z) with Dom={a,b,c} contains the clauses
  - (X≠a,Y≠a,Z≠a),
  - (X≠b,Y≠a,Z≠a),
  - ➔ these two clauses can be replaced by the single clause (X=c,Y≠a,Z≠a)

C. Improving Input Clauses

- Constraints can be optimized (and converted from negative clauses) using, e.g., information theory based decision tree algorithms.
- We haven't as yet completed an empirical evaluation of this idea.
  - Note same idea can be used to make GAC-schema checking faster.

GAC

- Most CSP solvers use GAC, and GAC is empirically much more effective than FC.
- How do we use clause learning ideas to improve GAC?

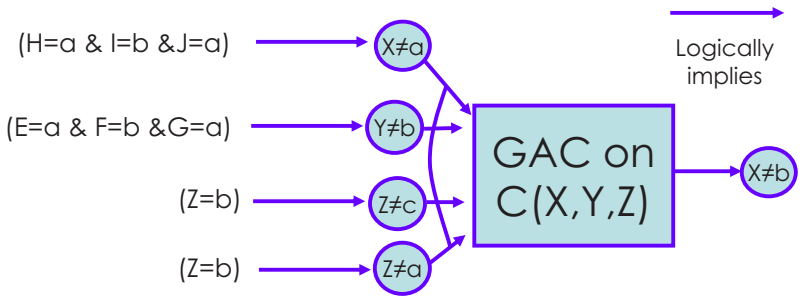
Standard Technique

- Inductively assume that every pruned value is labeled by a (negative) clause that caused the pruning.
- How do we compute a clause to label a value newly pruned by GAC on a constraint C?

Standard Technique

- The standard technique is to use the union of the clauses that pruned any value of any of the variables of the constraint. [Chen 2000].

Standard Technique



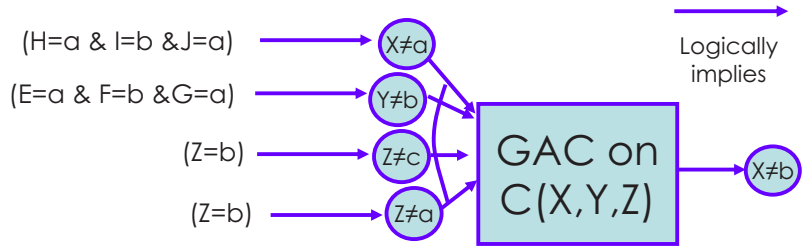
Therefore
H=a & I=b &J=a & E=a & F=b &G=a & Z=b ⇒ X≠b

In clause form
(H≠a,I≠b,J≠a,E≠a,F≠b,G≠a,Z≠b,X≠b)

Standard Technique

- We obtain only negative clauses.
- The resulting clause is very specific to this particular part of the search space, and can be quite long (not as powerful).

D. Better clauses from GAC



An immediate and computationally inexpensive clause we can obtain is simply the set of pruned values that caused the new pruning.

X≠a & Y≠b & Z≠c & Z≠a ⇒ X≠b
➔ (X=a,Y=b,Z=c,Z=a,X≠b) ➔ (Y=b,Z=c,Z=a,X≠b)

D. Better clauses from GAC

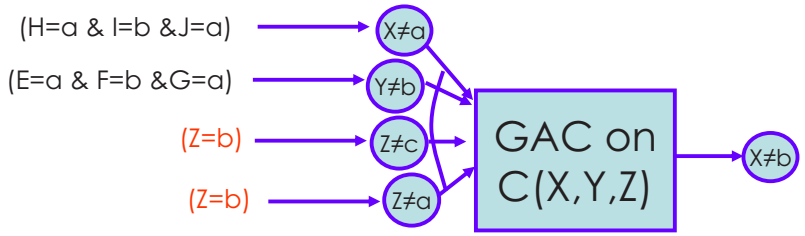
- This "all of the values pruned" clause is actually already captured by GAC processing itself.
  - (Y=b,Z=c,Z=a,X≠b)
  - Under any situation where we make all but one of these literals true GAC will infer the remaining literal.
    - E.g., if we prune b from Y's domain, c and a from Z's domain, GAC will detect that b must be pruned from X's domain.
    - Similarly if X=b, a and c have been pruned from Z's domain, GAC will prune b from Y's domain.

D. Better clauses from GAC

- However, even though this clause is in some sense "redundant" it can still be resolved against other clauses to produce powerful new clauses.
  - Increases the power of the search.
- In some sense this method is "converting" GAC inferences to clauses on the fly, and these clauses can then be used as inputs to more powerful resolution refutations.

D. Better clauses from GAC

- We can also resolve away various literals from this clause, to yield a variety of different clauses.
  - (Y=b,Z=c,Z=a,X≠b) ➔ (Y=b,Z≠b,X≠b)

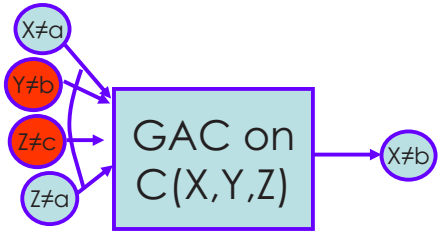


D. Better clauses from GAC

- The "all values pruned" clause empirically is often quite useful.
- Empirical analysis of the other possible clauses one could generate remains open.

Exploiting Constraint Structure

- The "all values pruned" clause fails to exploit information about the constraint.
- It could be that from the structure of the constraint only a subset of the currently pruned values contributed to the newly pruned value.



- E.g., perhaps X≠b follows from just Y≠b and Z≠c.
- If we can detect this efficiently, we could learn even better clauses from GAC.

Exploiting Constraint Structure

- In general, any set of pruned values that suffices to remove all of the supports of X=b is a minimal reason for the pruning.
- It is feasible to find such sets when the constraint relatively small (e.g., small enough to perform GAC-Schema)
- In this case such clauses can be more effective than the "all values pruned" constraint.

E. Better clauses from propagators

- Another critical technique in CSPs is the recognition that some constraints have a specialized structure, and thus specialized algorithms can be used to achieve GAC.
- These specialized algorithms can work even when the constraint is too large to be represented as a set of clauses.
- In these cases it should be feasible to additionally exploit this structure to obtain better clause reasons for the values pruned.

E. Better clauses from propagators

- E.g., all-diff.
  - The propagator (Regin 1994) works by identifying sets of variables S that consume all of the values in their domain.
    - E.g., a set of 3 variables all of which have the same 3 values remaining in their domain.
    - In this case these values are consumed, they cannot be used by any other variable in the all-diff.

E. Better clauses from propagators

- E.g., all-diff.
  - Say that X=b, Y=b, Z=b are all pruned because we have that b must be consumed by one of the variables in the set {V,W}.
  - Then a shorter, structure specific, clause explaining each of these pruned values is simply the set of values already pruned from the domain of V and W.
    - "b" is consumed by V and W because these other values are no longer available.
    - Other values pruned from the domains of other variables are irrelevant.

E. Better clauses from propagators

- In general, getting better clauses by exploiting structure specific to particular constraints remains an area where much additional work needs to be done.

Conclusions

Social Golfer

- From Katsirelos & Bacchus 2005.
  - Note: no sophisticated symmetry breaking techniques being used!

w,g,s	GAC	GAC+S	GAC+G
2-7-5	1586.0s	218.0s	4.4s
2-8-5	>2000.0s	1211.9s	5.5s
3-6-4	>2000.0s	869.7s	5.0s
3-7-4	>2000.0s	549.6s	1.6s
4-7-3	843.4s	91.5s	0.3s

Conclusions

- These techniques can yield a significant improvement in CSP solvers.
- Many other issues remain to be explored
  - The impact of learning different kinds of clauses from GAC.
  - Heuristics based on recently learned clauses–very successful in SAT, seemingly less so in CSPs.
  - Theoretical power in the presence of propagators.
  - Extending specialized constraints to be able to extract better clauses.