

Constraint Satisfaction Problems (CSC2512)

Lecture 8

Lecturer: Prof. Fahiem Bacchus,
notes by Jorge Baier

March 18th, 2005

1 Some relationships between DPLL and Resolution

The first thing we observe is that a refutation by the DPLL corresponds to a refutation by resolution. To see this fact consider the following set of clauses.

1. (a, b) ,
2. (c, d) ,
3. (e, f) ,
4. $(a, -b)$,
5. $(-a, -c)$,
6. $(c, -e)$,
7. $(-d, -f)$.

An execution of DPLL is shown in Figure 1. Notice that this tree is in fact a resolution refutation. This happens in general with any refutation by the DPLL algorithm. Notice, furthermore, that the resolution produced is regular; i.e., resolves at most once on each variable on the same path. This happens always since DPLL sets the value of a variable only once on each path.

In our last lecture we defined tree resolution and regular resolutions, which are refinements of resolution. Now we define other types of resolutions.

Definition 1 (Linear Resolution) *Is a particular type of resolution in which each clause C_i in the resolution proof is either an initial clause (input clause) or is derived from C_{i-1} and some previous clause.*

Observation: Prolog uses linear resolution, since it always use the last clause resolved against one in the input set.

Definition 2 (α -derivation) *If α is an assignment for all variables, an α -derivation is one where each resolution step involves a clause that is falsified by α .*

Definition 3 (Negative, Positive and Semantic Resolution) *A resolution is:*

Negative iff it is an α -derivation s.t. α assigns all variables to true.

Positive iff it is an α -derivation s.t. α assigns all variables to false.

Semantic iff it is an α -derivation, for some α .

Definition 4 (Davis Putnam Resolution) *A resolution is Davis Putnam if it is regular and in the proof, the variables are resolved upon the same order along any path to the empty clause.*

David Putnam Resolution is precisely the kind of resolution that is made by the DP algorithm since DP performs resolution steps resolving variables in a predefined order.

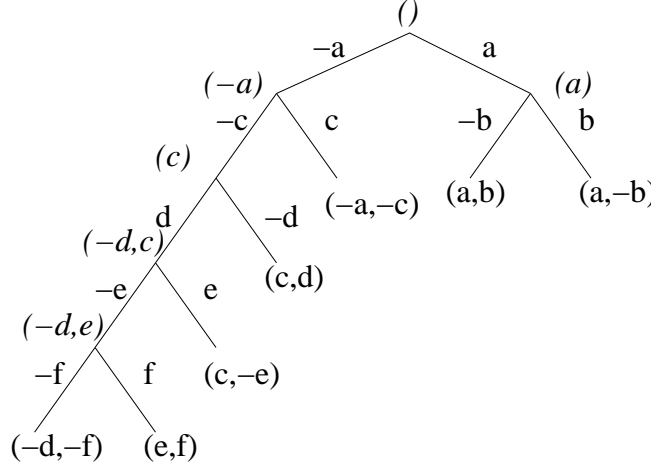


Figure 1: An execution tree of DPLL over an inconsistent set of binary clauses. Clauses on leaves are falsified by the choices of DPLL. Clauses in italic font are obtained by resolution of the clauses of the children.

Theorem 1 *Regular, linear, positive, negative, semantic, and DP resolutions are exponentially stronger than tree resolution.*

Nevertheless it is also true that tree resolution is exponentially stronger than DP.

Theorem 2 *Tree, regular, and DP resolutions are exponentially weaker than resolution.*

Some improvements to the DPLL algorithm, such as clause learning, can be seen as refinements of resolution. We say that a refinement is natural if the complexity of the refinement cannot increase after performing a reduction on an assignment. Formally,

Definition 5 (Natural Refinement) *A refinement of resolution S is natural if for all formulas F and partial assignments π ,*

$$C_S(\text{Red}(F, \pi)) \leq C_S(F)$$

Some interesting facts that are known are that regular, semantic, tree and DP resolutions are natural. However, it is not known whether or not clause learning is natural.

Assume CL is a DPLL algorithm with some scheme for clause learning. Moreover, assume CL- is a variant of CL which allows to branch on a literal that was already set. Then the following facts are known:

1. CL proofs can be exponentially shorter than those by tree, regular, and DP resolutions.
2. CL can be simulated by RES.
3. CL- with unlimited restarts is polynomially equivalent to RES.

The following theorem implies that if CL is natural then CL is as powerful as resolution.

Theorem 3 *For any $f(n)$ -proper, natural refinement S of RES, there exists a family of functions $\{F_n\}$ such that*

$$C_S(F_n) \geq f(n)C_{CL}(F_n).$$

2 Propagators

Propagators are algorithms that are used to achieve generalized arc consistency (GAC). GAC can always be achieved by exponential algorithms in the scope of the constraint, however we are obviously not interested in building exponential propagators. It turns out that for some common constraints, it is possible to build efficient propagators. These algorithms are very used in practice, most notably the commercial application ILOG is almost entirely based on them.

Before describing in more detail one of these propagators, we'll give some background on GAC.

We say that a constraint C is GAC if for all variables $V \in \text{scope}[C]$ and for all $d \in \text{Dom}[V]$ there exists an assignment $A \in C$ such that $V \leftarrow d \in A$. A is called a support for $V \leftarrow d$. If $V \leftarrow d$ has no support, we call it *inconsistent*. For any constraint C , there is a maximal set of inconsistent values, which we denote by $\text{MaxInc}(C)$. We can always make a constraint C GAC by removing $\text{MaxInc}(C)$ from the variable's domains.

There are some constraints for which $\text{MaxInc}(C)$ can be computed efficiently. One example is the "all-diff" constraint, which appears in many practical problems. The all-diff constraint for k variables, $C_{\text{diff}}(V_1, \dots, V_k)$ is satisfied iff for all $i \neq j$ ($i, j \in \{1, \dots, k\}$), $V_i \neq V_j$.

We can compute $\text{MaxInc}(C_{\text{diff}})$ in time $O(\sqrt{kn})$, where $n = \sum_i |\text{Dom}[V_i]|$. A reasonable assumption is that $|\text{Dom}[V_i]| = O(k)$, which implies that $\text{MaxInc}(C_{\text{diff}})$ can be computed in $O(k^{2.5})$.

For example, for the following variables,

$$\text{Dom}[X_1] = \{a, b\}, \quad \text{Dom}[X_2] = \{a, b\}, \quad \text{Dom}[X_3] = \{a, b, c\},$$

$X_3 \leftarrow a$ and $X_3 \leftarrow b$ are inconsistent for C_{diff} . Intuitively, X_1 and X_2 "chew up" the values a, b . Although there is a binary encoding for this constraint, enforcing arc consistency in this constraint produces no pruning since for every value of X_i there is a value of X_j ($j \neq i$) that satisfies the constraint.

To achieve GAC for $C_{\text{diff}}(X_1, \dots, X_k)$ in general, we do graph matching in a bipartite graph (which we call value graph) where there are arcs only between variables and their domain values. Formally, the value graph $G = (V, E)$ is such that $V = \{X_1, \dots, X_k\} \cup \bigcup_i \text{Dom}[X_i]$ and $E = \{(X_i, a) \mid \text{for all } a \in \text{Dom}[X_i], 1 \leq i \leq k\}$.

A subset of edges in a graph is called a matching if no two edges have a vertex in common. We say that a matching is *maximal* if it has maximum cardinality. A matching covers a set of vertices X if every vertex in X is incident on some edge in the matching.

Proposition 1 *Let G be a value graph for C_{diff} . C_{diff} is GAC iff every edge belongs to some matching covering the variables $\text{Scope}[C_{\text{diff}}]$.*

Proof: Notice that a matching covering all the variables has to be a maximal matching. C_{diff} is GAC iff every value $V \leftarrow d$ participates in a solution iff every edge in the value graph participates in a matching.

For a value graph, it is possible to find a maximal matching in time $O(\sqrt{|S_1|} \times \#\text{edges})$, where $|S_1| = k$ (no. of variables) and $\#\text{edges} = \sum_{i=1}^k |\text{Dom}[V_i]|$. Under the assumption that $|\text{Dom}[V_i]| = O(k)$, the matching can be found in $O(k^{2.5})$. Moreover, from one maximal matching we can find efficiently all edges that *do not* participate in any matching (these are the inconsistent values).

Figure 2 shows an algorithm that finds all inconsistent values for a C_{diff} constraint. It first computes a maximal matching and then finds all edges that do not participate in any matching.

2.1 Finding edges that are not in a maximal match

We now present an algorithm that finds all edges that do not participate in a maximal match. The algorithm is based in a theorem by Berge. First, we need some definitions.

```

GAC-diff( $C$ )
  build a value graph  $G$  for  $C$ 
   $m(G)$  = compute max. matching
  if  $|m(G)| < |scope(C)|$  return false // matching does not contain all vars.
  RemoveEdgesFrom( $G, M$ )
  return true

```

Figure 2: Algorithm for achieving GAC

Definition 6 (Free/matching vertices and edges for M) Let M be a matching. Edges in M are matching edges and those that are not are free edges. Vertices incident on an edge in M are matched otherwise they are free.

Definition 7 (Alternating path/cycle) An alternating path (cycle) in a value graph is a path (cycle) that alternates between free and matched edges.

Theorem 4 (Berge, 1970) An edge belongs to some, but not all maximal matches iff for an arbitrary matching M :

- it belongs to an alternating path that starts on a free vertex, or
- it belongs to an alternating cycle.

To find alternating path and cycles, from a value graph G and a matching M , we define the directed graph G_d^M such that G_d^M has the same vertices of G and for all variables X , and values y ,

$$\begin{aligned}
succ(X) &= \{a \in Values \mid (X, a) \in M\} \\
succ(y) &= \{Z \in Vars \mid (Z, y) \notin M\}
\end{aligned}$$

As an example, consider the following example for C_{diff} involving six variables.

$$\begin{aligned}
Dom[X_1] &= \{1, 2\}, & Dom[X_2] &= \{2, 3\}, \\
Dom[X_3] &= \{1, 3\}, & Dom[X_4] &= \{2, 4\}, \\
Dom[X_5] &= \{3, 4, 5, 6\}, & Dom[X_6] &= \{6, 7\}.
\end{aligned}$$

Figure 3 shows the value graph, one matching and a directed graph produced from both.

Observation 1: Every directed cycle in G_d^M corresponds to an even alternating cycle in G . This is true from the fact that G is bipartite.

Observation 2: Every directed path in G_d^M starting at a free vertex is an even alternating path in G . Indeed, if we have an odd alternating path, we can always extend it adding one more edge that is incident in the last vertex of the path without forming a cycle.

Figure 4 shows an algorithm, based in Berge's theorem that eliminates all vertices that do not participate in any matching.

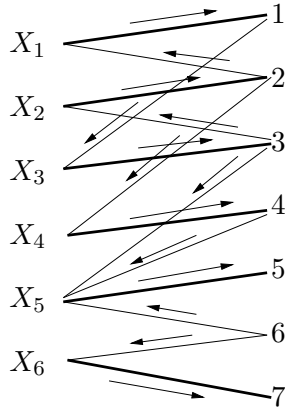


Figure 3: Edges of the value graph G are the undirected arcs. Thick arcs correspond to a matching M , and directed arcs are those of G_d^M .

```

RemoveEdgesFrom( $G, M$ )
  Build  $G_d^M$ 
  Mark all edges unused
  RE =  $\emptyset$ 
  Breadth-first-search from all free vertices in  $G_d^M$ 
  Mark as used all edges visited      // we are adding edges on an alternating
                                     // path from a free vertex

  Compute all strongly connected components
  Mark as used all edges that belong to some
    strongly connected component      // now we have added edges on an
                                     // alternating cycle

  Mark edges in  $M$  that are not yet marked
                                     // these edges are in all matches
                                     // unmarked edges are in no match

  Add unmarked edges to RE

```

Figure 4: A procedure that stores in variable RE all edges that do not participate in any maximal matching of G .