Some Notes on the Basic Connection Graph Proof Procedure

Krysia Broda

January 22, 2003

1 The Procedure

The connection graph procedure can be viewed as imposing some *book-keeping* on unconstrained resolution. The book-keeping enables 'look-ahead' to be performed, allows redundant steps to be avoided and prevents duplication of steps.

An *initial connection graph* for clauses S is set up by linking all potentially complementary literals in S by *links*. The unifiers corresponding to each link are evaluated and stored as a link label. [In diagrams we will label links using the following scheme: Initial links labelled 1,2, ... Inherited links labelled by $<$ parent label$>$ $<$ step number $>$, where $<$ step number $>$ is a, b, ... The unifiers will be assumed stored, and accessible].

Thereafter, there are two classes of operations, which take place alternately: *reduction* and *link selection*. These are performed until *termination*. There are two possible final states (assuming the user has not prematurely terminated the procedure), (i) when a graph contains $\square$, and (ii) the empty graph $\emptyset$. If the former occurs, we deduce the initial sentences S are unsatisfiable. If the latter, we deduce S are satisfiable (for the procedure is indeed sound subject to some restrictions; see Sections 2.3 and 2.4). [Unfortunately, in that case there is no information left from which a possible model could be derived, for all information has disappeared(!) (unless storage of the transactions is maintained for later processing.)]

2 Reduction Operations

2.1 Purity Principle

Any clause with an unlinked literal (called a *pure* literal) can be deleted, together with all other links adjacent to the clause. The intuition behind this reduction is that a pure literal cannot be unified with any other and so no proof of unsatisfiability can involve such a clause (for the pure literal cannot be removed from the resolvents). The removal of pure literal clauses and their links often causes a cascade of subsequent removals.

2.2 Inconsistent Links

There may be *inconsistent* links in a clause. These are links which, if selected, would result in a resolvent that contains at least one pure literal P; *i.e.* none of the links from the parent of literal P is inherited – the unifier of the selected link, when *combined* with the unifier of any link incident to the parent of P, yields failure. (As clauses are assumed to be variable disjoint, two substitutions can be *combined* simply by applying the unification algorithm to their union.)

Two links are *incompatible* if the substitutions of the labels cannot be combined; *i.e.* they cannot be unified.

A link from a literal L in clause C, to another in clause D, labelled by a unifier σ involving substitutions to variables $\underline{x}$ in C and $\underline{y}$ in D, is *inconsistent* if:

> for some literal M in C (other than L), and for each link k from M,
> the substitution in the label of k is incompatible with σ.

An inconsistent link can be deleted. For, if it were chosen and the resolvent indicated by it formed, the literal M (now instantiated by some substitution) will *inherit* no links since none of the unifiers labelling links from M were compatible with the selected link.

2.2.1 Example

In Figure 1CGPP link 1 is incompatible with link 2, since if the corresponding resolvent is formed by selecting link 1, (leaving the two clauses Qa, $\neg Qb$), the descendant of link 2 would be labelled by the substitution $\{a == b\}$ which is not possible. *i.e.* The substitution $\{x == a\}$ is combined with the unifier $\{x == b\}$. Thus Qa in the resolvent would be pure and the resolvent could immediately be deleted. Instead of forming the resolvent, the inconsistent link is just deleted.

2.3 Subsumption Deletion

A clause A may be *subsumed* by another clause B. In addition to the usual requirement for subsumption (*i.e.* $B\theta \subseteq A$) the links from B must have links corresponding to those from A. That is, for each literal L in A that has a subsuming literal L' in B, if L has a link to a literal M in clause C, then so must L' in B have a link to M. Without this restriction it is possible, from an unsatisfiable set of clauses, to derive the empty graph (supposedly the terminating condition for satisfiable graphs). The reason is that without the restriction the link necessary for deriving □ may be deleted along with a subsumed clause.

2.4 Tautologies and Tautological Links

With some care, links whose selection would give rise to a tautology can be removed. Clauses that are tautologies can always be removed from the initial graph. In this section we consider the criterion for making sure that a link that results in a tautology can indeed be deleted.

2.4.1 Example

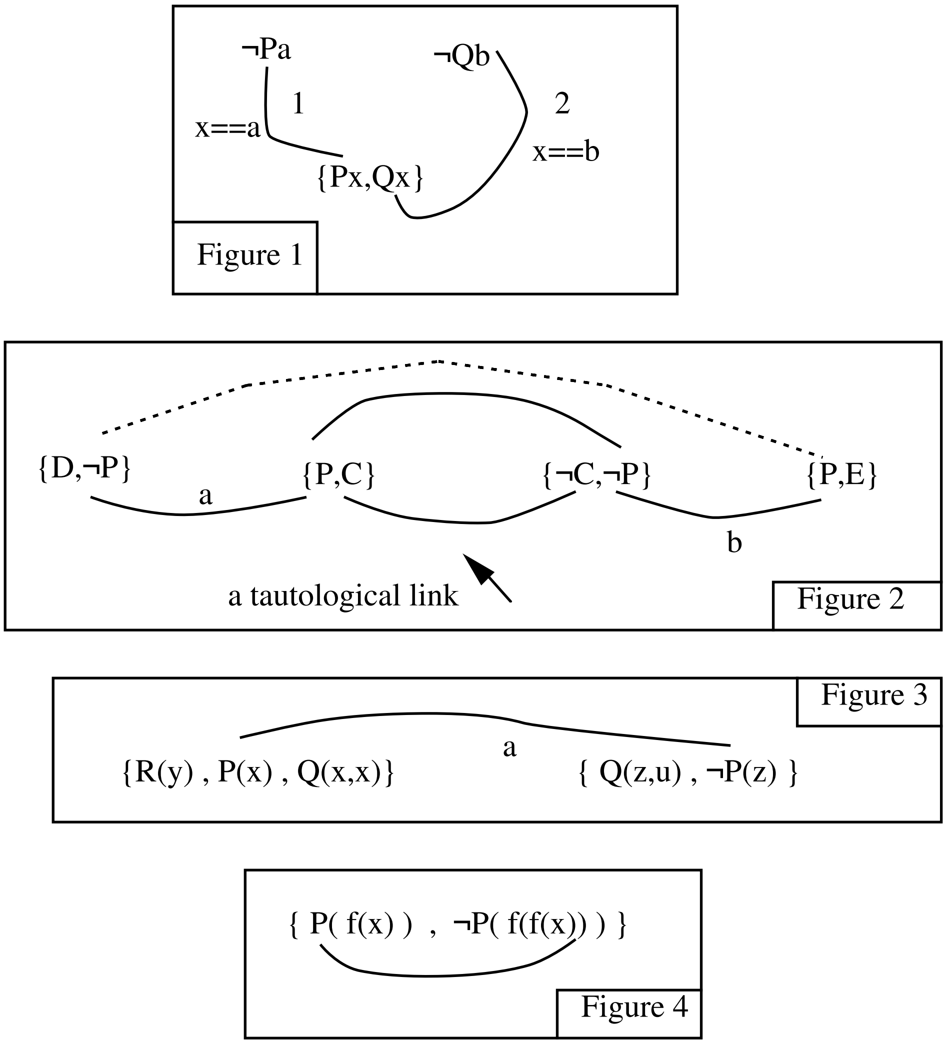


Figure 1: Figures 1 to 4 CGPP

In Figure 2CGPP suppose the tautology link between C and $\neg C$ is selected, giving resolvent $P \vee \neg P$. A typical pair of links from this clause will be the descendants of a, b. (Suppose these are a' and b'). Suppose now that a' is selected. The resolvent will be $\neg P \vee D$ – which is subsumed by the parent clause as long as the links in the parent correspond to those in the resolvent. The links incident to D are directly inherited from D in the parent clause, so there is no difficulty with them. The links incident to $\neg P$ are those inherited from b' (and other similar ones). For subsumption, $\neg P$ in the parent $D \vee \neg P$ must have a link corresponding to that inherited from b'. *i.e.* like the dotted link in Figure 2CGPP. Only if this criterion is satisfied may tautological links just be deleted. [Otherwise, they are best not selected for they give rise to numerous additional inherited links].

2.5 A Special Reduction

Some special reduction operations are possible when a resolvent subsumes a parent clause. For example, $Q(u) \vee P(x)$, and $\neg P(a)$ give $Q(u)$, which subsumes $Q(u) \vee P(x)$. So $P(x)$, in the latter can simply be deleted along with its links. The idea can be extended; for example, in Figure 3CGPP deleting link a gives rise to $R(y) \vee Q(z,z) \vee Q(z,u)$, which can be factored to $R(y) \vee Q(z,z)$, which subsumes the parent $R(y) \vee P(x) \vee Q(x,x)$, which can then be deleted. The whole series of operations could be simulated by deleting Px from the parent (and hence link a).

2.6 Factoring

A clause may be *factored*, sometimes producing a factor which subsumes the parent.

For example, the clause $Qa \vee Qx \vee Py$ factors to $Qa \vee Py$. When two literals are merged in this way, it is only necessary to retain the links from one of the two literals. [The justification (informally) is that since without factoring both literals would have to be removed for a refutation, each using its own links, the merged literal can mimic either of its two parents]. Even if the parent is not subsumed and not deleted, this *mono-inheritance* is allowed.

(**Note:** Some links may occur in the graph between literals in the same clause (see Figure 4CGPP for example). These are called *self-resolving-links* and should only be used for generating descendant links (merely because of the complexity of deriving the inherted links). (It has not been proved in general that they *never* need be selected, but it does not seem that they need to be.) See the example in the next section illustrating the links derived from selecting a self-resolving link.

3 Selection Operation

After reduction has taken place (as much or as little as is desired) a link is selected. The operation consists of:

Choose link k incident to literals L and K between two clauses C and D.
Form resolvent, add to graph.
Add links derived from the parent links to corresponding literals in the resolvent.
Delete link k.

This is implemented as follows:

copy the parent clauses C and D (without literals L and K) and rename the variables (substitution σ);
copy each link to literals not incident to L or K and rename the variables in the bindings using σ; the copied literals and links will become the resolvent R after applying the binding of k;
apply the binding of link k to the copied literals (forming R);
delete k;
for each literal M in the resolvent and each link m incident to M combine the binding of m with that of k. If the unifier does not exist remove m;

The copying of links means that a link in C between a literal X and some other Y is copied to a link between the copy of X in R and Y.

When combining unifiers to form inherited links the usual unification algorithm will be assumed. In order to combine unifiers σ, ρ from two links from some clause and making substitutions to a set of variables $\underline{x}$, the result is obtained by unifying the two sets of substitutions. For example,

Combining $\{x == f(y)\}$ and $\{x == z\}$ gives $\{z == f(y)\ ,\ x == f(y)\}$
Combining $\{x == a\}$ and $\{x == f(y)\}$ fails.

The final substitution θ is restricted so that $vars(\theta)$ are mentioned in at least one of the two literals incident to the link.

The bindings on the inherited links (those copied from the parent clauses) are computed by combining the binding on the selected link with that on the copied link. The link is retained if the combination exists. It is assumed that clauses are always maintained with their variables distinct.

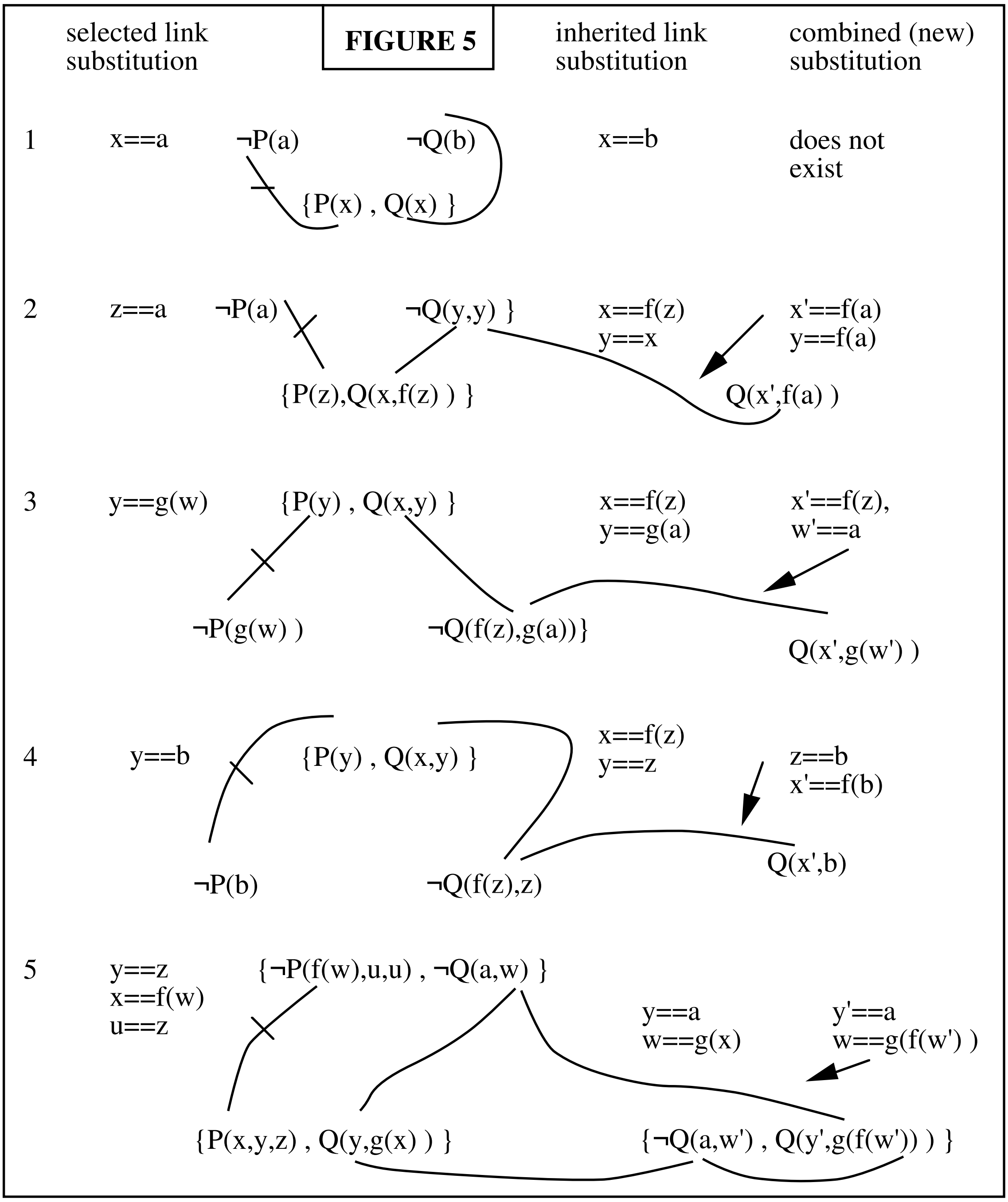


Figure 2: Figure 5 CGPP

3.1 Examples

In Figure 5CGPP the bindings labelling the parent link (*i.e.* the selected link) can involve variables in both the parent clauses They are labelled as x, y, z, etc. from one parent clause and u, v, w from the other. The variables in the resolvent are renamed to maintain variables disjointness as x', y', etc.

In example (5) in Figure 5CGPP there is also a self-resolving link in the resolvent. Labels of such links use a renamed copy of the clause. Selecting a self-resolving link is done by taking two distinct copies of the parent clause. The result is a proliferation of links and such deletions are better avoided! As an **exercise**, try selecting the link between the two literals of $Pfx1 \vee \neg Pffx1$. The resolvent is $Pfx2 \vee \neg Pfffx2$. Assume the literals in the original clause link elsewhere, so that it is not deleted by purity.

Selection criteria for links are suggested in the slides.

It is also possible to operate on different parts of the connection graph in parallel, if the parts are "far enough away from each other". Formalising the notion of distance is not easy, because care has to be taken that two resolvents formed in parallel do not inherit a link between them that could be missed by the lack of synchronisation. In Figure 6CGPP, for example, links 6 and 4 might be selected safely in parallel, but links 4 and 5 probably could not be. Suppose link 4 is chosen for deletion, then the resolvent $D(c, b)$ will inherit links 3 and 2, as 3a, 2a respectively. If link 5 were to be selected in parallel, the resolvent $\neg D(x, b) \vee \neg T(x)$ would inherit link 2, and additionally 2a, if the first resolvent was completed first. If not, the link might not be inherited, most likely causing loss of soundness.

4 Properties of the Connection-Graph Procedure

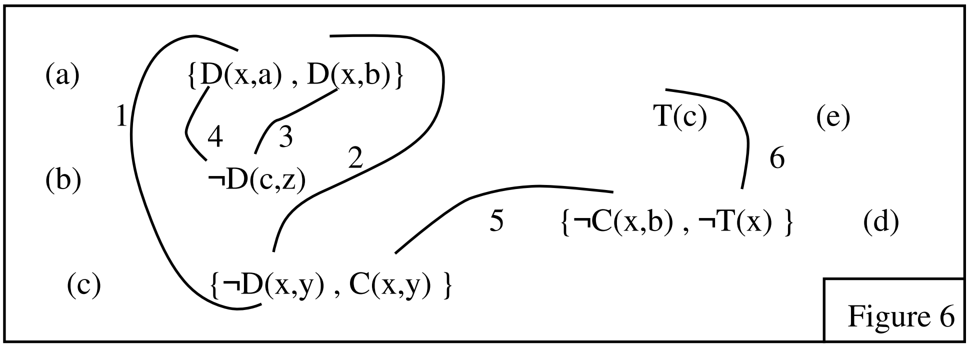
Given clauses S, initial graph G $(= INIT(S))$

$$INIT(S) - >^* STATE$$

means $INIT(S)$ can be transformed to $STATE$ (another connection graph) by none or more connection graph operations – which may either be selection of a link or a reduction step.

The connection graph proof procedure possesses the following properties:

Refutation Soundness: $INIT(S) - >^* \Box \Rightarrow S$ unsatisfiable,
where $\Box$ is a state containing the empty clause.
This is almost obvious, since a connection graph operation is a resolution step, and resolution is sound, and reduction operations only reduce the graph; they do not affect the resolution steps.



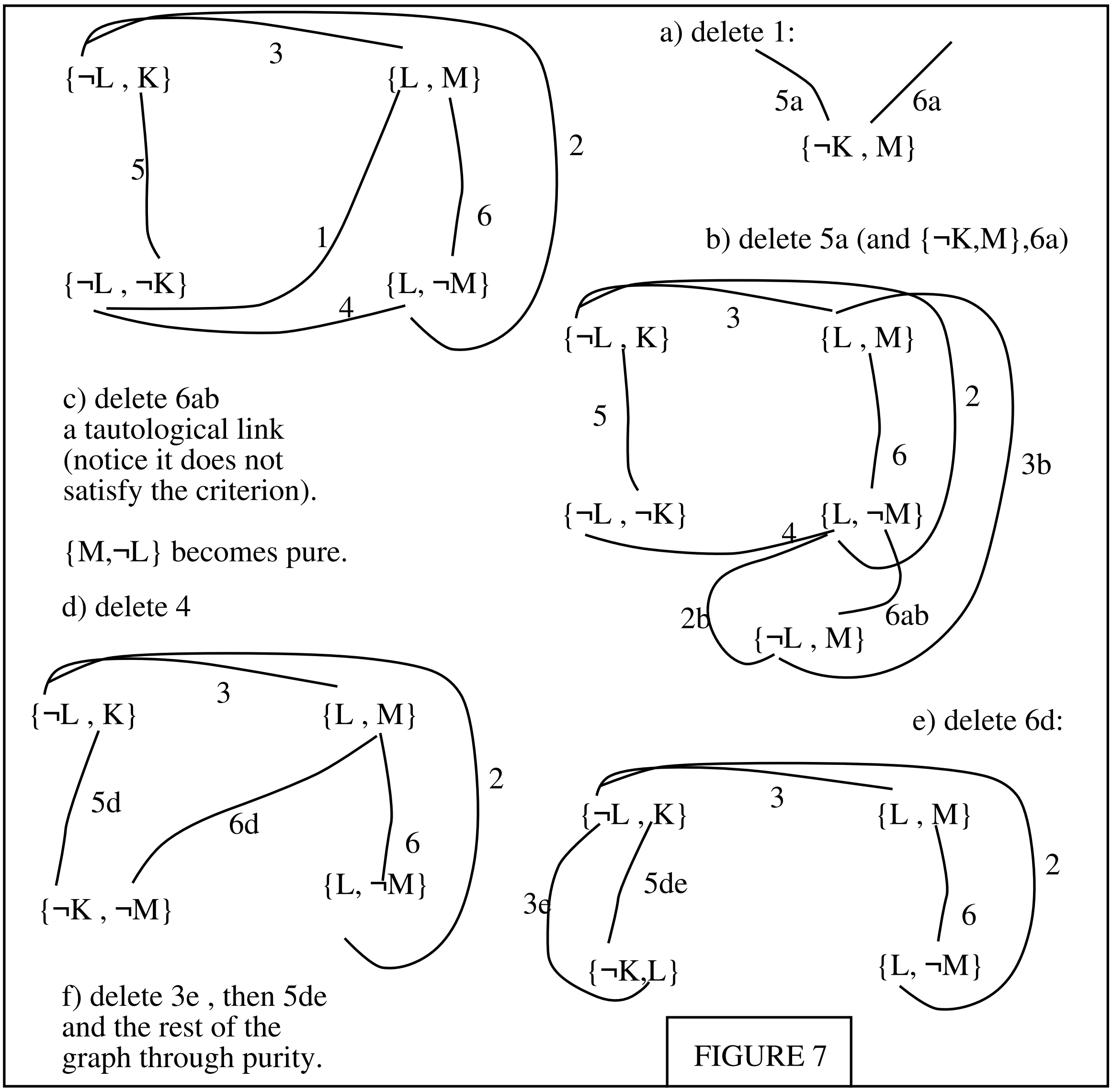


Figure 3: Figures 6 and 7 CGPP

Refutation Completeness: S unsatisfiable $\Rightarrow INIT(S)- >^* \Box$;
i.e. at least one sequence of steps leads to the empty clause.

Refutation Confluence:

$$S \text{ unsatisfiable}, INIT(S)- >^* G_1, INIT(S)- >^* G_2$$
$$\Rightarrow \exists G : G_1- >^* G \text{ and } G_2- >^* G$$

These various properties are proved in [Eis91]. It is also important to know whether production of an empty graph (all clauses removed) implies S satisfiable. It does, as long as the restrictions on subsumption and tautology deletion are adhered to. Eisinger calls this *Affirmation Soundness.*

Since it is important that only one graph need be stored at a time, for the overheads for each state are large, we should be sure that for any graph G_1 such that $INIT(S)- >^* G_1$ (for unsatisfiable S), it is possible to reach $\Box$ from G_1. Confluence assures this. (Put $G_2 = \Box$. Then see that, because of subsumption, the only graph that is derivable from $G_2 = \Box$ is the graph containing only $\Box$.)

It is possible to place restrictions on link selection and the effect is generally to remove confluence, but not completeness. Suppose however that such a restriction exists (anyway the null restriction exists!) which is complete and confluent, then *Strong Completeness* would mean that, for any ordering of link selection which is fair, the connection graph procedure is complete, confluent and *Noetherian* (meaning it eventually yields the empty clause for initially unsatisfiable S).

For several "obviously fair orderings", the procedure is not Noetherian, as Eisinger illustrates. *i.e.* after several steps a new graph isomorphic to the old is obtained. It seems that the notion of fairness must take more into account than simply the fact that all original links disappear after a finite number of steps. Even if fairness is weakened(?) to enforce that all original links or those derived from them must disappear after a finite number of steps, strong completenss is not assured. To date, it is not known if there is, or if there is not, a strong completeness theorem (for sets of general clauses), although there are some "nearly strong" completeness theorems.

4.1 Proving properties of connection graphs for Horn clauses

In this section an outline of the strong completeness proof for sets of Horn clauses is given. The description is based on work by Eisinger. The main concept is that of a *kernel* for a set of Horn clauses S, which is a refutation formed from ground instances of S in the following, particular way: Let N be an "initial" clause with only negative literals. Each negative literal of N is resolved with the positive literal in a clause instance. The resolvent is another all-negative clause, which is similarly treated. Some of the resulting negative literals may match positive facts and eventually all resulting negative literals will match positive facts. Such a refutation can be represented as a tree, with positive leaf literals and negative non-leaf literals, as shown in Figure 8CGPP. Each leaf literal indicates a *closure*. It is not difficult to show by induction that for a finite set of unsatisfiable ground instances such a tree can be formed. The tree is called a *kernel* and satisfies the following two properties:

- Each match in the kernel corresponds to a link in the initial c-g;
- each clause instance is an instance of a clause in the initial c-g.

At each step of transforming the c-g the kernel can be transformed as well, such that these two properties are maintained and, in addition, the size of the kernel reduces, where the *size* of a kernel is the number of closures in it. Figure 9CGPP gives a c-g and the corresponding kernel, with closures marked by the corresponding link.

It can be shown that inconsistent links never correspond to any closure, nor are instances of pure clauses or tautologies present. Figure 10CGPP then illustrates the kernel transformation that results from the following c-g steps applied to the connection graph in Figure 9CGPP: first remove link 2 (inconsistent), then select link 3. In the c-g the clause $green(x) \vee \neg blue(x)$ is pure. The closure in the kernel that corresponds to the selected link is removed and the siblings of the positive literal take the place of the matching negative literal. You can check that the two properties above are maintained for the new graph and tree. Removing pure clauses or inconsistent links cannot affect the kernel as these are not present in it. Each time a step is made the kernel size decreases, so eventually it will be reduced to the empty tree corresponding to $\Box$.

For a large set of clauses there may be several kernels, one for each different subset of unsatisfiable ground instances. If links are always selected from one kernel then this will lead to the generation of $\Box$. But it is not known which such links are, so a *fair* criterion for selecting links is needed that guarantees all links will eventually be selected. Subsumption deletion may also delete a kernel altogether, but in this case it can be shown there will be a smaller kernel remaining, as long as the links on the subsuming clause correspond to those on the subsumed clause. Repeated subsumption cannot occur for ever, so some resolution steps have to be made and reduction will occur.

5 Good and Bad Points

5.1 Advantages

- Link inheritance (no need to recompute unifiers).
- Few redundant steps.

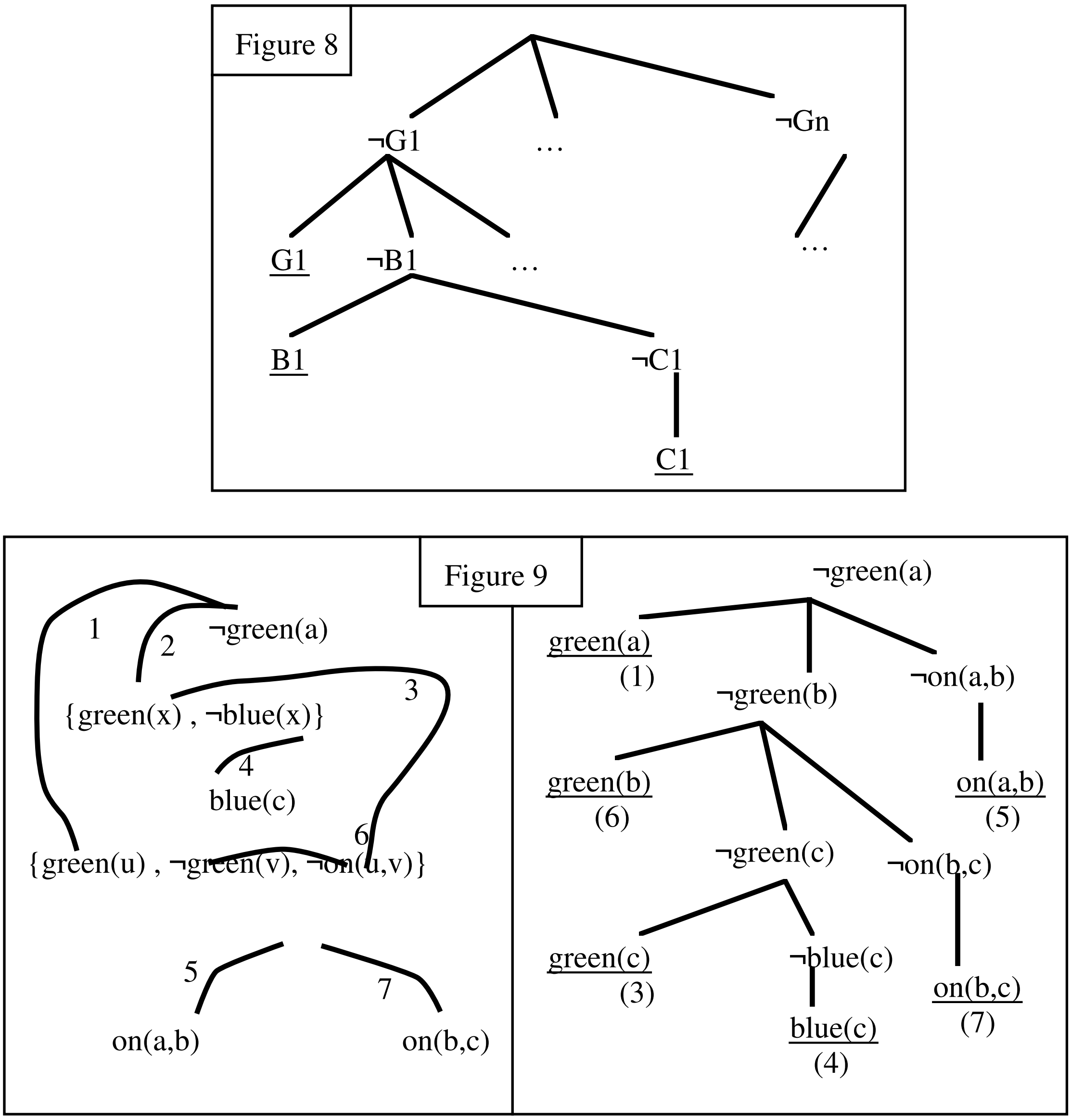


Figure 4: Figures 8 and 9 CGPP

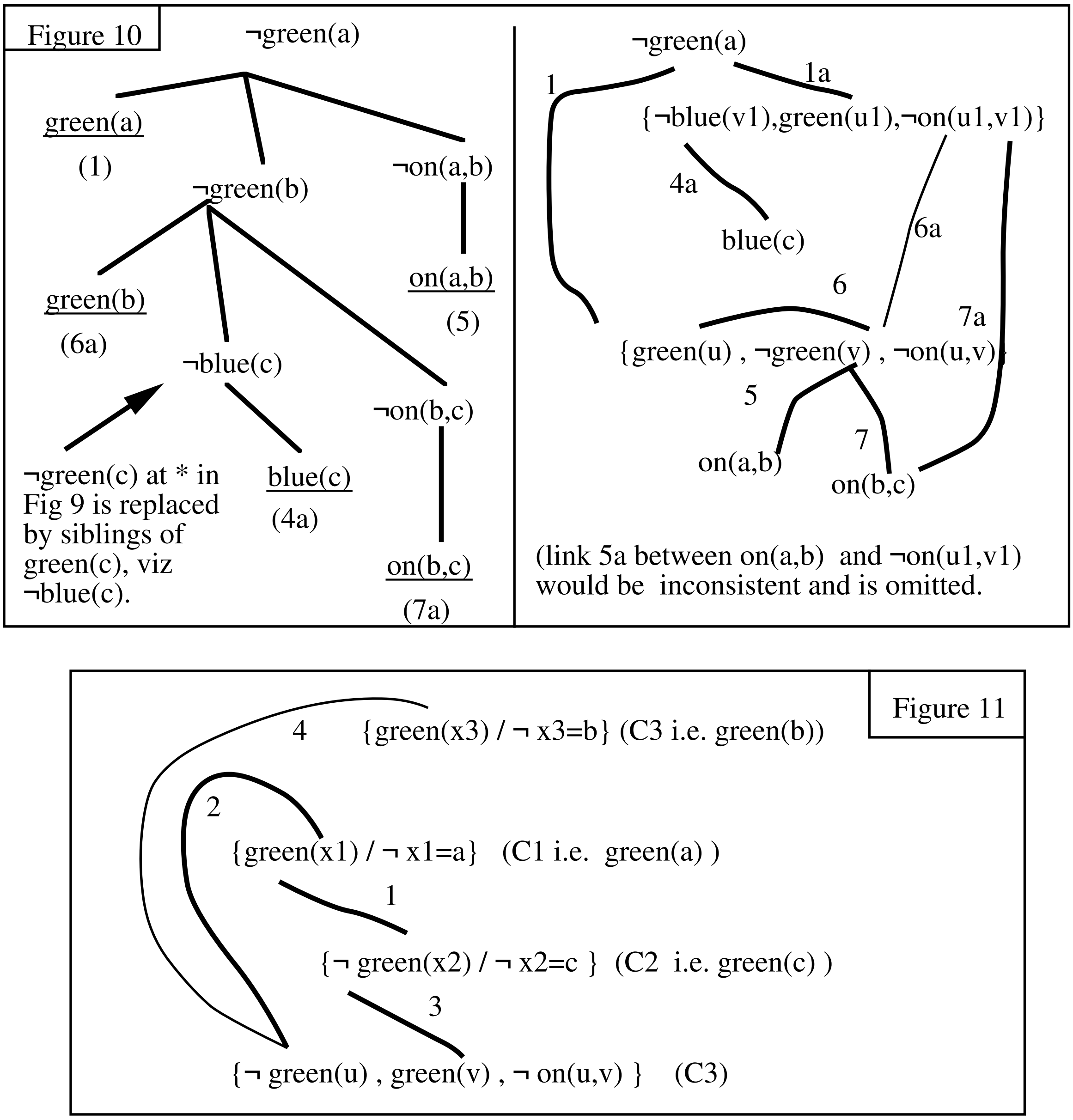


Figure 5: Figures 10 and 11 CGPP

- Powerful reduction operations, which use all available information (unifiers, potential factors, subsumption, etc).
- Can be extended to non-clausal sentences and tableaux.
- Few problems for Horn clausesets and can be utilised for such sets in useful ways (not all of which have been exploited).
- The notion of link has proved useful in other refinements, such as Hyper-linking and tableau methods.

The idea of potential unifiers can be used to find potential instances of a set of clauses that can then be tested using the Davis Putnam Procedure or Model Generation Procedure to see if they are unsatisfiable. One such method was suggested in [Plais92] and operates as follows. For each clause, one link is selected from each literal and the combination of the link bindings is made. The substitution is applied to the whole clause to generate a clause instance. If the instance is not ground then a new (unique) ground constant is substituted for all unbound variables in the clause. (The same constant is substituted for every variable.) These clauses are tested for unsatisfiability. If there is no success it could be due to some self-resolving links needing to be unravelled; *i.e.* two copies of the clause containing such a link are added and the self-resolving link becomes a link between the two copies. Or it may be that the necessary binding for a clause can only be found after several such steps. In any case, all the clauses found in round 1 that are not subsumed are added to the graph and the procedure of finding instances is performed again. See the slides for an example.

On the slides we mention some benefits of using the CGPP with Horn clause logic programming.

Finally, we briefly suggest how the labels can be interpreted as constraints on the universal quantifiers of clauses. This is illustrated in Figure 11CGPP. Any clause can be generalised by replacing its arguments by new variables and adding a negative literal to assert that the new variable equals the original argument. These new literals are called *constraints* and constrain the universal quantifier. In Figure 11CGPP, the constraint in clause $C1$ is $x1 == a$; the clause could also be written as $\forall x1(x1 = a \rightarrow green(x1))$. Such constraints are normally propagated to the links between literals and evaluated before selection.

After selecting a link, which now has a very simple "variable - variable" binding, its resulting constraint is associated with the resolvent. Selecting link 2 in Figure 11CGPP results in the constrained clause $green(v) \vee \neg on(u, v)/\neg u = a$.

The approach is not particularly interesting for standard unification. But when extended to other constraints, such as membership of sets, or non-membership of sets, it can be quite powerful. An implementation of these ideas was made by an MSc. student Ilya Pozdeev (1996). In particular, facts give rise to such membership constraints. For example, in

Figure 11CGPP links 2 and 4 could be combined into a single link, link 24 (say), which went to a new clause $green(x)/\neg x \in \{a, b\}$, which represents both clauses $C1$ and $C3$.

Combining constraints is treated as an extended unification process, and any algorithm should be subject to the usual properties of the unification algorithm:

- possession of an "m.g.u".
- a canonical form for "unifiers"
- an algorithm for computing constraints

In fact, if substitutions are interpreted as equations, then for any two terms X and Y and unifier λ, $\forall[\lambda \models X = Y]$ is true. For a m.g.u. θ, $\forall[\theta \equiv X = Y]$ is true and $\forall[\lambda \models \theta]$, where the quantifier is over variables in the terms X and Y. It is these properties that are maintained for other kinds of constraints.

As an example, consider combining the following constraints:

$$
\begin{array}{ll}
 & mem(x, [0, 1]) \wedge mem(y, [0, 1]) \wedge mem(0, [x, y]) \wedge x = 1 \\
\Rightarrow & x = 1 \wedge mem(y, [0, 1]) \wedge mem(0, [1, y]) \\
\Rightarrow & x = 1 \wedge mem(y, [0, 1]) \wedge y = 0 \\
\Rightarrow & x = 1 \wedge y = 0
\end{array}
$$

$$
\begin{array}{ll}
 & mem(x, [a, b, c]) \wedge mem(x, [b, c, d]) \wedge x \neq b \\
\Rightarrow & mem(x, [b, c]) \wedge x \neq b \\
\Rightarrow & x = c
\end{array}
$$

For membership constraints some of the simplification steps that are part of a "unification" algorithm as as follows, although there are quite a few others.

- $mem(x, [a, b, c]) \wedge mem(x, [b, c, d])$ simplifies to $mem(x, [b, c])$;
  it could also simplify to $x = b$ but this is *not* the 'most general'.
- $mem(x, [a, b, c]) \wedge x = u$ simplifies to $x = u$ if x is a, b or c;
  otherwise, if u is ground, it simplifies to false, else is unchanged.
- $mem(x, [a, b, c]) \wedge x \neq u$ simplifies to $mem(x, [a, b, c] - [u])$ if u is a, b or c;
  otherwise if u is ground, it simplifies to $mem(x, [a, b, c])$, else remains unchanged.

Notice that constraints such as $=, \neq, mem, \neg mem$ also have the convenient property that they can be split into a conjunctive form of literals that mention variables from a single literal. They can then be propagated across links in much the same way as is done in the standard connection graph.

5.2 Disadvantages

- Storage.
- Expensive operations.
- Not easy to integrate equality.

6 Unsoundness of Unrestricted Tautology Deletion

In the example in Figure 7CGPP the empty graph has been derived and yet the initial graph was unsatisfiable.

References

[Eis91] N. Eisinger, Completeness and Confluence Properties of Clause Graph Resolution, Pitman, 1991

[Plais92] Hyperlinking, Journal of Automated Reasoning, Vol. 9, 1992.