Multiple labelling generators in non monotonic RMS graphs

Jérôme Euzenat

N° 2076

Octobre 1993

PROGRAMME 3

Intelligence artificielle,
systèmes cognitifs
et interaction homme-machine

ISSN 0249-6399

Multiple labelling generators in non monotonic RMS graphs

Jérôme Euzenat[*]

Programme 3 — projet Sherpa

Rapport de recherche n° 2076 — Octobre 1994 — 49 pages

Abstract: Non monotonic reason maintenance systems (RMS) are able, provided with a dependency graph (which represents a reasoning), to return a weakly grounded labelling of that graph (which represents a set of beliefs that the reasoner can hold). There can be several weakly grounded labellings. This work investigates the labelling process of these graphs in order to find parts of the graph which lead to multiple labellings: the multiple labelling generators (MLG). Two criteria are presented in order to isolate them. It is proved that:

- they do not belong to stratified even strongly connected components (SCC) of the complete support graph.
- they are successive initial SCC of unlabelled part of alternate even SCC.

Previous algorithms from Doyle and Goodwin are considered and new ones are put forward. This leads to a better understanding of labelling generation mechanisms and previous algorithms. They are discussed from the stand-point of the properties of correctness and potential completeness (the ability to find one but any of the labellings).

Key-words: Non monotonic reasoning — truth maintenance — TMS — reason maintenance — RMS — dependency graph — label propagation.

Version du Jeudi 27 Janvier 1994

[*] `Jerome.Euzenat@imag.fr`

Unité de recherche INRIA Rhône-Alpes
IMAG-LIFIA, 46, avenue Félix Viallet, 38031 Grenoble Cedex 1 (France)
Téléphone: +33 76 57 47 77 — Télécopie: +33 76 57 47 54

Les générateurs d'étiquetages multiples dans les graphes de SMR non monotones

Jérôme Euzenat

Résumé: Les algorithmes de maintien d'un raisonnement non monotone sont capables, à partir d'un graphe de dépendance (représentant le raisonnement) de trouver un étiquetage faiblement fondé (représentant les croyances que peut tenir le raisonneur). Les graphes de systèmes de maintien du raisonnement (SMR) non monotones admettent plusieurs étiquetages. Ce travail se concentre sur le processus d'étiquetage du graphe afin de caractériser les parties du graphe qui engendrent ces différents étiquetages: les générateurs d'étiquetages multiples. Deux critères sont présentés afin de les isoler. Il est ainsi montré que:

- Ils ne font pas partie des composantes fortement connexes (CFC) paires stratifiées du graphe de supports complets.
- Ils sont dans les différentes CFC initiales de la partie non étiquetée des CFC paires alternées.

Ces résultats permettent d'éclairer les algorithmes antérieurs et d'en proposer de nouveaux qui peuvent atteindre tous les étiquetages possibles du graphe. Les algorithmes sont comparés à la lumière des propriétés de correction et de complétude potentielle.

Mots-clé: Raisonnement non monotone — maintenance de la vérité — TMS — maintien du raisonnement — SMR — graphe de dépendances — propagation d'étiquettes

Multiple labelling generators in non monotonic RMS graphs

Jérôme Euzenat

Common-sense reasoning includes reasoning in incomplete environment. To cope with this incompleteness, one can reason about a completion of his knowledge. This completion has been implemented in several AI systems (for instance, frame like systems) and the investigation on the behaviour of such systems reveals their non monotonic character. Reason maintenance systems (RMS for short) are algorithms able to maintain a certain coherence in a non monotonic reasoning. They use a dependency graph between formulas in order to determine which formulas are to be considered as holding. To that extent, they produce a labelling of such graphs and maintain it incrementally, i.e. they react upon each new modification of the graph by updating the labelling.

Nonetheless, it is useful to have a formal model of the computations with completion mechanisms available. This had led to the design of non monotonic logics. In non monotonic logics, there traditionally exist several theories (called extensions) which represent alternative possible worlds considering the available knowledge and the non monotonic inference rules. RMS also generates several labellings of a dependency graph (which represents the reasoning). The further investigation on non monotonic reasoning spreads in two directions:

- Mapping these specifications into a logical framework in order to show that such RMS provide a useful tool in the implementation of logic provers. This has been done for autoepistemic logic [Fujiwara& 89, Reinfrank& 89] or default logic [Reinfrank& 88, Junker& 90, Levy 91].
- Designing a program which implements these specifications and respects some properties. This activity has been considered as "serious hacking" [MacDermott 91] for long. But designing a correct, complete and efficient implementation is something very difficult.

Taking advantage of the results already provided by the first trend, this work uses the second one. Its aim (as that of previous ones [Goodwin 87, Petrie 87, Reinfrank 90]) is the elucidation of the points which make RMS algorithms so clumsy if they got to work efficiently. For that purpose, RMS are considered through a semantic characterisation and the algorithms are exposed with regard to their properties. The semantic characterisation considered here is not yet that of non monotonic logics but that of the weakly grounded labelling of the dependency graph manipulated by the RMS. In continuation with our former work [Euzenat 90, 91] this one is aimed at designing RMS which are able to take into account simultaneously every possible labelling of a graph.

The first task — in order to retrieve the different alternatives occurring in labelling the RMS dependency graph — is the recognition of the existence of these alternatives. The classical reason maintenance algorithms such as Doyle's do not really recognise them. A more recent algorithm by Ulrich Junker and Kurt Konolige [Junker& 90] is able to produce every possible labelling. However, such an algorithm finds the possible alternatives but does not record them. As a consequence, this algorithm is not incremental any more. In order to implement such a system, the extensions must be indexed with particular generators (like the generating defaults of an extension in default logic). It is thus necessary to isolate the parts of the graph which generate the multiple labellings of the graph: the multiple labelling generators (MLG). They can also be used in order to be able to quickly switch from one alternative labelling to another.

For that purpose, the topological concepts involved in the labelling process are progressively presented. This leads to a, progressive as well, reconstruction of previously known algorithms; they are considered with regards to these topological entities and under the light of the three properties of:

- *Correctness*: the program must respect the specifications of RMS (finding a weakly grounded labelling).
- *Potential completeness (*or *non deterministic completeness)*: the program must be able to find any of the weakly grounded labellings.
- *Efficiency*: the program must have the lower complexity as possible.

Unfortunately, this reveals to be a very difficult task provided that these generators arise dynamically during the labelling process. So, the way labellings can be found will be studied through the classical algorithms. This work does not lead to simple results:

- The generators are not parts of the graph but parts of a partially labelled graph.
- Moreover, the characterisation of these generators is not simple enough in order to design a fast algorithm for labelling the graph.

This shed some light on the mechanisms used by previous algorithms and their properties. In particular, it is shown that Goodwin's algorithm, despite the fact that it is a very fast algorithm for finding a labelling is not able to find each possible labelling of the graph (potential completeness). This lead to a better understanding of the labelling in a RMS and to more natural but less efficient algorithms.

Thus, this report does not solve the main problem of designing a correct, efficient and potentially complete RMS. Its contribution stems from the new presentation of RMS, the proposal of an efficient potentially complete RMS algorithm which mirrors that of Goodwin. It is published in this form and with the same title because it has already been used and quoted by others.

The paper is organised in three main parts; the first one (§1) is dedicated to the presentation of RMS and weakly grounded labellings, the second one (§2 and 3) methodologically circumscribes the multiple labelling generators and the last one (§4) discusses the current RMS algorithms and presents new algorithms. A more detailed description of the sections is as follows:

First (§1), the RMS definition in terms of graph labelling is given. The concept of weakly grounded labelling of a graph is declaratively defined. It is be the norm for RMS labellings.

Second (§2), it is shown that the multiple labelling generators are neither patterns of the dependency graph nor of Doyle's minimal support graph but of the complete support graph. The characterisation of generators must be foreseen with regard to easily computable parts of a graph: the strongly connected components (SCC for short). Then, the place where they can occur is restricted by eliminating some graph configurations which do not allow for alternatives. This second result shows that alternative generators are to be found in the alternate even SCC (AE-SCC). Then, some examples are given of AE-SCC which generate alternatives as a whole and other which fall apart. As a conclusion, the general result expected is not here. The multiple labelling generators are not the whole AE-SCC but rather the so-called even cycles. The problem to address is that of detecting these cycles.

Third (§3), a notion of constraints, able to restrict the generation of alternatives in these AE-SCC, is defined. This leads to partition the graph inside the SCC on a second criterion (constraints) and to find out alternative generators in initial unlabelable SCC of the unconstrained support graph.

Fourth (§4), how known RMS algorithms manage for labelling the unlabelable subSCC is discussed. This leads to stress some drawbacks of these algorithms and propose some modifications in order to overcome them. This allows to understand what the real work performed by classical RMS algorithms (or the necessary work to do) is.

Note that proofs are included in the body of the text since they are part of the argumentation. They can be omitted in a first reading.

1. Truth maintenance systems: definitions

The definitions of classical reason maintenance systems, in Doyle's fashion, are reminded here and in the next subsection. They are based on the exposition of [Reinfrank 90] while differing in syntax and in the distinction between weakly and strongly grounded labellings (instead of globally grounded labelling). Another worth reading general presentation is that of Drew Mac Dermott [Mac Dermott 91]. A description of the processing of various RMS (in French) can be found in [Euzenat 90].

1.1. Dependency graph

Reason maintenance systems are aimed at managing a knowledge base considering different kinds of reasoning. Such a system is connected to a reasoner (or problem solver) which communicates every inference made. The RMS has in charge the maintenance of the reasoner's current belief base. RMS developed so far focused on non monotonic reasoning or multiple contexts reasoning. They record each inference in a *justification* which relates *nodes* representing propositional formulas plus a special atom ($\perp$) representing contradiction. A justification ($<\{i_1,\ldots i_n\}\{o_1,\ldots o_m\}>: c$) is made of an IN-list ($\{i_1,\ldots i_n\}$) and an OUT-list ($\{o_1,\ldots o_m\}$). Such a justification is said to be valid if and only if all the nodes in the IN-list are known to hold while those in the OUT-list are not; a node, in turn, is known to hold if and only if it is the consequence (c) of a valid justification. The recursion of the definition is stopped by nodes without justification and by the axioms which are nodes with a justification containing empty IN- and OUT-lists.

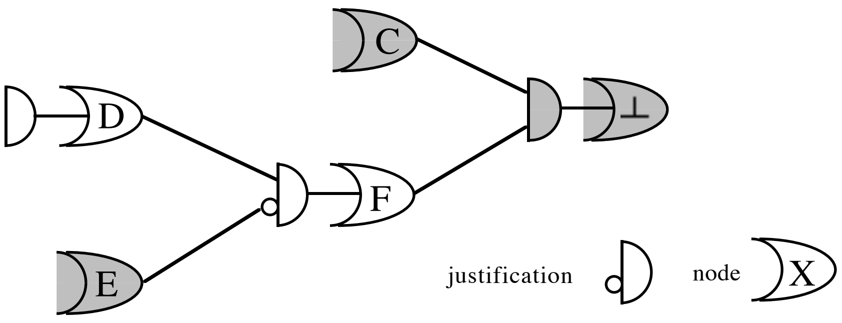


Figure 1. A dependency graph is here represented as a boolean circuit where or-gates are nodes and and-gates are justifications where the nodes in the IN-list come directly while nodes in the OUT-list come through a not-gate. Nodes which have a justification whose IN- and OUT-lists are empty (e.g. *D*) represent true formulas because they do not need to be inferred. White nodes and justifications are considered valid while hatched ones are invalid. Of course, the value propagation satisfies the rules implied by the circuit components. So, the formulas in the base are ensured having a valid justification (i.e. corresponding to a valid inference).

Definition 1. Given a finite set $\mathcal{N}$ of nodes, a subset $\mathcal{C}$ of $\mathcal{N}$ (the nodes known as inconsistent), a set $\mathcal{J}$ such that $j\in\mathcal{J}:\mathcal{P}(\mathcal{N})\times\mathcal{P}(\mathcal{N})\times\mathcal{N}$, ($j$ is noted $<\{i_1,\ldots i_n\}\{o_1,\ldots o_m\}c>$), $<\mathcal{N},\mathcal{C},\mathcal{J}>$ is called a *dependency graph*.

For simplicity's sake, it is often considered that $\mathcal{C}$ is reduced to $\{\perp\}$. Some practical notations about justifications are first introduced.

Definition 2. $\forall j=<\{i_1,\ldots i_n\}\{o_1,\ldots o_m\}c>\in\mathcal{J}$, $\text{InList}(j)=\{i_1,\ldots i_n\}$

Definition 3. $\forall j=<\{i_1,\ldots i_n\}\{o_1,\ldots o_m\}c>\in\mathcal{J}$, $\text{OutList}(j)=\{o_1,\ldots o_m\}$

Definition 4. $\forall j=<\{i_1,\ldots i_n\}\{o_1,\ldots o_m\}c>\in\mathcal{J}$, $\text{Cons}(j)=c$

The potential support graph represents the graph of precedence between nodes abstracting from justifications. As a consequence, such a graph cannot account for co-occurrence (or no co-occurrence) of nodes in the same justification.

Definition 5. The *potential support graph* is an oriented graph $G_p=(V,E_p)$ where V is the set of nodes considered by the reason maintenance system and $E_p=E_p^+\cup E_p^-$ is a set of edges such that

$(n_1, n_2)\in E_p^+ \Leftrightarrow \exists j\in\mathfrak{J};$ Cons$(j)=n_2$ & $n_1\in$InList(j)

$(n_1, n_2)\in E_p^- \Leftrightarrow \exists j\in\mathfrak{J};$ Cons$(j)=n_2$ & $n_1\in$OutList(j)

This allows to speak of cycles which are of first importance in reason maintenance systems because their absence ensures good properties for algorithms and, above all, because they enable multiple labellings.

Definition 6. A *cycle* is a path $n_0,\ldots n_m$ in the potential support graph such that $n_0=n_m$ and $\forall i\in[1\ m], (n_{i-1},n_i)\in E^+\cup E^-$.

Definition 7. An *even cycle* is a cycle such that there is an even number of negative transitions ($(n_{i-1},n_i)\in E_p^-$) in the path.

Definition 8. A *stratified even cycle* is a cycle such that there is no negative transition ($(n_{i-1},n_i)\in E_p^-$) in the path.

Definition 9. An *alternate even cycle* is a cycle such that there is a non null even number of negative transitions ($(n_{i-1},n_i)\in E_p^-$) in the path.

Definition 10. An *odd cycle* is a cycle such that there is an odd number of negative transitions ($(n_{i-1},n_i)\in E_p^-$) in the path.

At last, some useful notations about nodes are introduced and some properties of these concepts are mentioned.

Definition 11. The *justification-list* of a node n is the set of all the justifications which have it as consequent. $\forall n\in\mathfrak{N}$, L-Just$(n)=\{j\in\mathfrak{J};$ Cons$(j)=n\}$

Definition 12. The *consequents* of a node n is the set of all the consequences of justifications in which n takes part (either in the IN- or OUT-list): $\forall n\in\mathfrak{N}$,

$$\text{Csq}(n) = \{n'\in\mathfrak{N};\ \exists j\in \text{L-Just}(n');\ n\in\text{InList}(j)\cup\text{OutList}(j)\}$$

Consequence. $\forall n\in\mathfrak{N}$, Csq$(n)=\cup_{j\in\mathfrak{J};\ n\in \text{InList}(j)\cup\text{OutList}(j)}$Cons$(j)$

Definition 13. The *antecedents* of a node n is the set of all the antecedents of the justifications of its justification-list: $\forall n\in\mathfrak{N}$,

$$\text{Ant}(n)=\{n'\in\mathfrak{N};\ \exists j\in\text{L-Just}(n)\ \&\ n'\in\text{InList}(j)\cup\text{OutList}(j)\}$$

Consequence. $\forall n \in \mathcal{N}$, $\text{Ant}(n) = \cup_{j \in \text{L-Just}(n)} \text{InList}(j) \cup \text{OutList}(j)$

Lemma 1. $\forall n, n' \in \mathcal{N}$, $n' \in \text{Ant}(n) \Leftrightarrow n \in \text{Csq}(n')$

proof.

$n' \in \text{Ant}(n)$

$\Leftrightarrow$ $\exists j \in \text{L-Just}(n);\ n' \in \text{InList}(j) \cup \text{OutList}(j)$

$\Leftrightarrow$ $\exists j \in \mathcal{J};\ \text{Cons}(j) = n \;\&\; n' \in \text{InList}(j) \cup \text{OutList}(j)$

$\Leftrightarrow$ $n \in \text{Csq}(n')$ ◊

Notation. A node without antecedents is called a premise node. It can be a node with an empty justification list or a node with a justification with empty IN- and OUT-lists.

1.2. Labelling

Jon Doyle's TMS [Doyle 79] proceeds by labelling the nodes of the graph with IN and OUT tags which reflect whether the formulas associated to the nodes are known to hold or not. A labelling respecting the constraints stated above is an admissible labelling and a labelling which gives the label OUT to the $\perp$ node is a consistent labelling. The TMS algorithm finds a weakly grounded labelling, i.e. a consistent admissible labelling which relies on no circular argument. The main work of the TMS occurs when it receives a new justification. It then has to integrate the justification in the graph and, if this changes the validity of the formula, it must propagate this validity: all the nodes which could be IN-ed because of the justified node and all those which could be OUT-ed are examined and updated. If an inconsistency occurs following the addition of a justification, the system backtracks on the justifications in order to invalidate a hypothesis — a formula inferred non monotonically — which supports the inconsistency.

First, the notions of validity of a justification and INness of nodes are introduced. They ground the local basis for a semantics of RMS action.

Definition 14. A justification is *valid* if and only if each node of its IN-list is IN and each node of its OUT-list is OUT: $\forall j \in \mathcal{J}$, $\text{valid}(j) \Leftrightarrow \forall n \in \text{InList}(j)\ \text{label}(n) = \text{IN} \;\&\; \forall n \in \text{OutList}(j)\ \text{label}(n) = \text{OUT}$.

Definition 15. A node is *IN* if and only if there is a justification in its justification-list which is valid: $\forall n \in \mathcal{N}$, $\text{label}(n) = \text{IN} \Leftrightarrow \exists j \in \text{L-Just}(n);\ \text{valid}(j)$.

Definition 16. A node which is not IN is OUT: $\forall n \in \mathcal{N}$, $\text{label}(n) = \text{OUT} \Leftrightarrow \forall j \in \text{L-Just}(n), \neg \text{valid}(j)$.

This recurring definition is grounded since a justification without antecedent is valid and a node without justification is OUT. Validity and INness allow to introduce the local properties of labellings which are closedness and groundedness. They ensure that the labelling respects the natural constraints of logical circuits.

Definition 17. A set L of nodes represents the *labelling* of the dependency graph in which the nodes of L are IN and the others are OUT.

Notation. label(n)=IN can be noted as $n \in L$ and label(n)=OUT as $n \notin L$.

Definition 18. A labelling is *closed* if and only if every node which has a valid justification is IN.

Definition 19. A labelling is *locally grounded* if and only if every node which is IN has a valid justification.

Definition 20. An *admissible labelling* of the dependency graph is an assignation of IN or OUT tags to all the nodes of the graph which respect the two previous definitions.

After the definition of admissible labelling, notations referring to it can be introduced.

Definition 21. A node n is *accorded*[1] to a justification j if and only if the node is IN and belongs to the IN-list of j or is OUT and belongs to the OUT-list of j:

$$\text{accorded}(n,j) \Leftrightarrow (\text{label}(n)=\text{IN} \wedge n \in \text{InList}(j)) \vee (\text{label}(n)=\text{OUT} \wedge n \in \text{OutList}(j))$$

Definition 22. A node n is *unaccorded* to a justification j if and only if it is OUT and belongs to the IN-list of j or is IN and belongs to the OUT-list of j: $\text{unaccorded}(n,j) \Leftrightarrow (\text{label}(n)=\text{OUT} \wedge n \in \text{InList}(j)) \vee (\text{label}(n)=\text{IN} \wedge n \in \text{OutList}(j))$

Consequence. A node n is unaccorded to a justification j if and only if it is not accorded to j: $\text{unaccorded}(n,j) \Leftrightarrow n \in \text{InList}(j) \cup \text{OutList}(j) \wedge \neg \text{accorded}(n,j)$

The basic notions of interesting labellings (for the user and the RMS) can be introduced. First, there are strongly grounded labellings which are found with restrictive conditions, and then there are weakly grounded labellings which are the aim to achieve for the RMS in the general case.

Definition 23. A *strongly grounded* (or *well founded*) *labelling* is an admissible labelling in which a node's INness or OUTness does not depend on its own IN or OUTness. So, an admissible labelling is strongly grounded if the set of the nodes $\mathcal{N}$ can be ordered with a partial order $\sqsubseteq$ such that each n_i which is IN has at least one valid justification $<I\ O>:n_i$ in which $\forall n' \in I \cup O$, $n' \sqsubset n$ ($n' \sqsubseteq n$ but not $n \sqsubseteq n'$) and each n_i which is OUT has for every (invalid) justification $<I\ O>:n_i$ a node $n \in I \cup O$ unaccorded with j such that $n' \sqsubseteq n$.

Theorem 2. If a particular dependency graph admits a strongly grounded labelling, this is the only admissible labelling.

[1] *Aligned* and *misaligned* can also be used following [Mac Dermott 91].

proof. Assume there is another labelling. This means that in this labelling two cases are possible:

1 an actual IN node is labelled OUT. There are two new cases:
   a) if the node is supported (by others) so one of its antecedents (supporting ones) is in the first case (case 1),
   b) otherwise (the node is not supported) it is a premise node and cannot admit another labelling (closedness property).

2 an actual OUT node becomes IN, there are two new cases:
   a) if the node is supported, one of its antecedents is in the same case (case 2),
   b) the node is in a cycle of mutually supporting nodes, they must remain OUT until one of them be supported (i.e. be in the case 2a).
   c) otherwise (the node is not supported, i.e. has no justification), this is impossible since the node being IN does not respects the local groundedness property.

This proof is valid because the strongly grounded labelling definition ensures that the tree of supporting nodes is explored and that the leaves of that tree are reached (without looping; except for an OUT node which must then remain OUT). The order used in that tree is really constraining, since, at the end of the process, it is grounded on premise nodes. Therefore there is no other possible grounded labelling. ◊

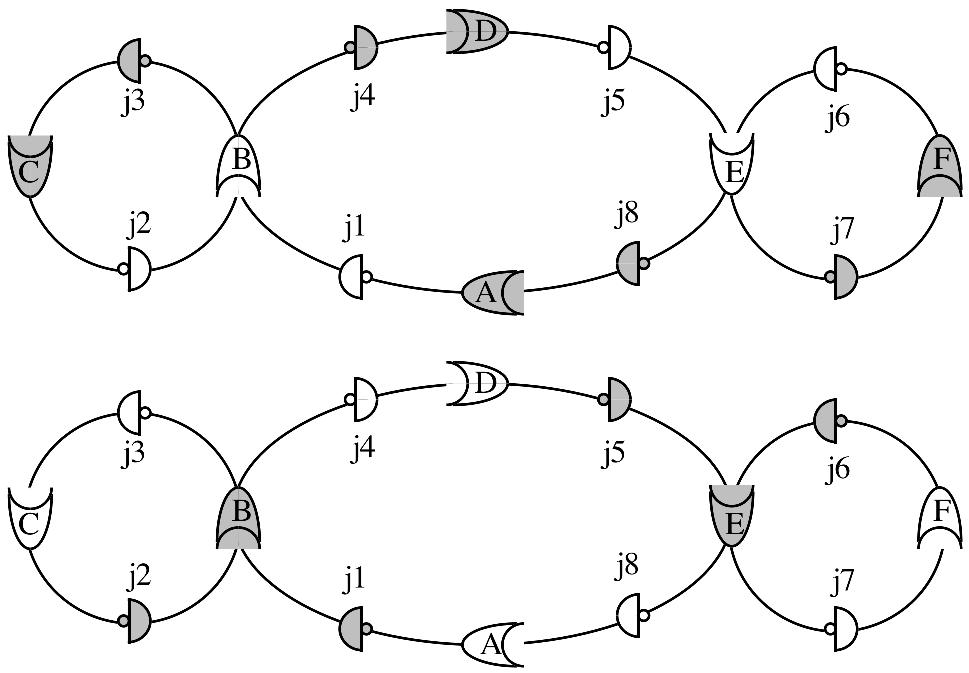


Figure 2. Two weakly grounded labellings of the same graph.

This theorem shows that strongly grounded labellings are indeed strong. But, due to the inherent incompleteness of knowledge which characterises systems using non monotonic reasoning, some graphs do not admit strongly grounded labellings. As a consequence, RMS look for a weakly grounded labelling.

Definition 24. A *weakly grounded* (or *not well founded*) *labelling* is an admissible labelling in which a node INness does not depend on its own INness. So, an admissible labelling is weakly grounded if the set of the nodes labelled IN can be ordered with a partial order $\sqsubseteq$ such that each n_i (which is IN) has at least one valid justification $<I\ O>:n_i$ in which $\forall n' \in I \cup O$, $n' \sqsubset n$ ($n' \sqsubseteq n$ but not $n \sqsubseteq n'$).

Consequence. Strongly grounded labellings are weakly grounded labellings.

This distinction between strongly and weakly grounded labelling emerges from the initial TMS algorithm of Jon Doyle. It will be better understood in the remainder. The interest of strongly grounded labellings is in their uniqueness; this is not always the case for weakly grounded labellings.

Definition 25. A *consistent labelling* of the dependency graph is a labelling which labels all nodes of $\mathcal{C}$ with OUT labels ($\mathcal{C} \cap L = \emptyset$).

The TMS task consists of computing a consistent and weakly grounded labelling of the graph. It is noteworthy that this labelling is possibly not unique. So the actual RMS choose among possible labellings when they exist. In order to ensure existence of such labellings, working hypothesis 1 is assumed in the remainder. The task of eliminating non consistent labellings is let apart as well.

Working hypothesis 1. From that point, only graphs without odd cycles are taken into account.

This restriction is also motivated by the will to avoid insane graphs (odd cycles are often paradoxical situations) and to avoid NP-completeness of the truth maintenance task [Úrbanski 87, Elkan 90]. Both reasons have always led truth maintenance system designers [Doyle 79, Goodwin 87, Petrie 87] to take this working hypothesis into account. Moreover, it ensures that the graph has, at least, a weakly grounded labelling.

Theorem 3. A dependency graph without odd-cycle admits at least one weakly grounded labelling.

proof. see [Charniak& 80]. It is noteworthy that every author [Goodwin 87, Reinfrank 90] refers to that proof. In fact, this theorem is cumbersome to prove because an algorithm has to be exhibited which stops with an admissible labelling in case of graphs without odd-cycles. So, it is difficult to exhibit the property without the algorithm one does not want to promote. ◊

The task of a RMS is thus to find a weakly grounded labelling of the current graph. Given a particular graph, this task is now fully characterised. Some more tools about RMS graphs are introduced in order to support further developments. Then, §2 isolates the parts of the dependency graphs which do not raise problems with their labellings. A longer treatment will concern the problematic parts.

1.3. Complete support graph

A TMS algorithm incrementally finds a weakly grounded labelling. This means that it reacts upon each modification in the graph by updating the current labelling. So it works by taking into account a current labelling. Since there are several possible labellings, it is useful to localise the parts of the graph which generate several labellings. They are called the multiple labelling generators but this does not judge in advance the characteristics of such generators. What their nature is will be progressively circumscribed.

An alternative labelling depends not only on the graph topology but also on the current labelling. Indeed, some multiple labelling generators in a context are not be generators in another one. In order to take this into account, Doyle's TMS uses a sub-graph of the dependency graph called the minimal support graph. After some work on the minimal support graph [Quintero 89], it does not seem suited for detecting multiple labelling generators, so the complete support graph is used here. Such a graph is just a more powerful tool for understanding incremental updating than the minimal support graph. It is used here as nothing more. This does not account for whether it is useful for RMS implementation in the form in which it is presented below. It is only useful in the first step of a new algorithm. Its power comes from the fact that it takes into account each constraint which concerns a node and not only a minimal set of constraints.

Intuitively, the complete support graph is the graph of all nodes really supporting others. In short all nodes accorded to valid justifications and all nodes unaccorded with invalid justifications of OUT nodes.

Definition 26. The *complete support graph* is an oriented graph $G_c=(V,E_c)$ where V is the set of nodes considered by the reason maintenance system and $E_c=E_c^+\cup E_c^-$ is a set of edges such that

$$(n_1, n_2)\in E_c^+ \Leftrightarrow \exists j\in\mathfrak{J};\ \text{Cons}(j)=n_2\ \&\ n_1\in\text{InList}(j)$$
$$\&\ \text{accorded}(n_1,j)\ \&\ \text{label}(n_2)=\text{IN}\ \&\ \text{valid}(j)$$
$$\vee\ \neg\text{valid}(j)\ \&\ \text{label}(n_2)=\text{OUT}\ \&\ \text{unaccorded}(n_1,j)$$

$$(n_1, n_2)\in E_c^- \Leftrightarrow \exists j\in\mathfrak{J};\ \text{Cons}(j)=n_2\ \&\ n_1\in\text{OutList}(j)$$
$$\&\ \text{valid}(j)\ \&\ \text{label}(n_2)=\text{IN}\ \&\ \text{accorded}(n_1,j)$$
$$\vee\ \neg\text{valid}(j)\ \&\ \text{label}(n_2)=\text{OUT}\ \&\ \text{unaccorded}(n_1,j)$$

This can be stated otherwise with the help of some rewriting:

Lemma 4. $\text{label}(n_2)=\text{OUT}\ \&\ \text{Cons}(j)=n_2 \Rightarrow \neg\text{valid}(j)$

Lemma 5. $n_1\in\text{InList}(j)\cup\text{OutList}(j)\ \&\ \text{valid}(j) \Rightarrow \text{accorded}(n_1,j)$

Consequence. The complete support graph is an oriented graph $G_C=(V,E_C)$ where V is the set of nodes considered by the reason maintenance system and $E_C=E_C^+\cup E_C^-$ is a set of edges such that

$(n_1, n_2)\in E_C^+ \Leftrightarrow \exists j\in\mathfrak{J}$; Cons($j$)=$n_2$ & $n_1\in$InList(j)

& (valid(j) $\vee$ (label(n_2)=OUT & unaccorded(n_1,j)))

$(n_1, n_2)\in E_C^- \Leftrightarrow \exists j\in\mathfrak{J}$; Cons($j$)=$n_2$ & $n_1\in$OutList(j)

& (valid(j) $\vee$ (label(n_2)=OUT & unaccorded(n_1,j)))

Here, some super-structures are provided in order to facilitate further proofs; they can be omitted in a first reading.

Definition 27. The *real supporting justifications* of a node n (R-J-Sup(n)) are defined such that:

Label(n)=IN $\Rightarrow$ R-J-Sup(n)={$j\in$L-Just(n); valid(j)}

Label(n)=OUT $\Rightarrow$ R-J-Sup(n)={$j\in$L-Just(n); $\neg$valid(j)}=L-Just(n)

Definition 28. The *real supporting nodes* of a node n (R-N-Sup(n)) are defined such that:

Label(n)=IN $\Rightarrow$ R-N-Sup(n)={$n'\in$Ant(n); accorded(n',j) &$j\in$R-J-Sup(n)}

Label(n)=OUT $\Rightarrow$ R-N-Sup(n)={$n'\in$Ant(n); unaccorded(n', j) & $j\in$R-J-Sup(n)}

Consequence.

$(n_1, n_2)\in E_C^+ \Leftrightarrow \exists j\in\mathfrak{J}$; Cons($j$)=$n_2$ & $n_1\in$InList(j) & $n_1\in$R-N-Sup(n_2)

$(n_1, n_2)\in E_C^- \Leftrightarrow \exists j\in\mathfrak{J}$; Cons($j$)=$n_2$ & $n_1\in$OutList(j) & $n_1\in$R-N-Sup(n_2)

The complete support graph constitutes a tool for later work.

2. Configurations excluding alternative labellings

The purpose of this section is the characterisation of a wide set of graph configurations which exclude multiple labelling generation. The complete support graph of a *labelled* graph and its partition in strongly connected components (SCC) is considered. Parts of the graphs which do not contain multiple labelling generators are isolated. Then, this labelling will be progressively extended to parts in which it does not allow for alternative labelling.

James Goodwin proposed a TMS algorithm which uses strongly connected components of the minimal support graph in order to speed up the algorithm. He showed that if SCC are examined following the precedence order, it is possible to label completely the graph, applying Doyle's algorithm (two steps) inside each SCC before labelling the rest of the graph. A conclusion to be drawn from this algorithm is that multiple labelling generators can only lie in SCC (the non determinism of Doyle's algorithm lies in the second step of the algorithm and nowhere else).

From now, strongly connected components of a graph are considered at length. Given a graph $G=<V,E>$ where V is a set of vertices and E a set of edges, a SCC decomposition of the graph is a partition of maximal subsets of V in which there is a path from each vertex of the subset to each other vertex in the subset. Such a subset is called a strongly connected component. First, SCC are used on the complete support graph which only contains nodes as vertices. Thus, for each SCC C there is a corresponding set of nodes of the dependency graph and they are used that way. Consequently, it is correct to speak, for instance, of the justifications of a node in a SCC.

This section goes a step ahead by characterising special SCC which do not allow for multiple labellings. The result is interesting in two respects:

- Multiple labelling generators reside inside other SCC.
- These SCC are fully labelled after the first step of Doyle's algorithm. So the algorithm and its processing can be kept short.

2.1. Alternate and stratified even SCC

SCC of the complete support graph are first classified. For that purpose, some facts about SCC are introduced.

Lemma 6. Inside an SCC C of the complete support graph, a node cannot influence simultaneously positively and negatively on another node (i.e. if two nodes n and n' are in C, there is not two justifications j and j' in L-Just(n) such that $n\in$InList(j) and $n'\in$OutList(j')).

In order to prove that lemma, the following path notations are introduced.

Notations. A path p from n to n' which goes through x OUT-lists is noted $p{:}n\text{-}x{\rightarrow}^{*}n'$. If this path is reduced to njn' it can be noted $n{\rightarrow}n'$. If $p_1{:}n\text{-}x{\rightarrow}^{*}n'$ and $p_2{:}n'\text{-}y{\rightarrow}^{*}n''$ then $p_1p_2{:}n\text{-}(x+y){\rightarrow}^{*}n''$.

proof. Let $n, n'\in C$ such that there exists a path $p({:}n\text{-}x{\rightarrow}^{*}n')$ from n to n' through x negative edges

If $p_1{:}n\text{-}x_1{\rightarrow}^{*}n'$ (x_1 odd) and $p_2{:}n\text{-}x_2{\rightarrow}^{*}n'$ (x_2 even) then, there exists an odd cycle in the graph since: If x is odd, then pp_1 is an odd cycle, and if x is even, pp_2 is an odd cycle. ◊

Consequence. $\forall C$ SCC, $(E_c^{+}\cap E_c^{-})\cap C\times C=\{\}$

Definition 29. An *even SCC C* is a SCC such that the complete support graph contains no odd cycle going through a node of C.

Consequence. Every SCC is an even SCC.

proof. Assuming the opposite, there should be an odd cycle in the complete support graph. So, there also should be an odd cycle in the potential support graph. This is contradictory with working hypothesis 1. ◊

Definition 30. A *stratified even* SCC C is an even SCC such that the complete support graph contains no negative arc between two nodes in *C*.

Definition 31. An *alternate even* SCC *C* is an even SCC such that the complete support graph contains at least a negative edge between two nodes of *C*.

2.2. Status of nodes and justifications with regard to SCC

After partitioning SCC, justifications and nodes can also be partitioned with regard to a particular SCC. This is useful for immediate results but also for defining constraints in the next section.

Definition 32. Given a SCC *C*, an *internal node* is a node in *C* ($n \in C$), other nodes ($n \notin C$) are *external nodes*.

Definition 33. *External incident nodes* of an SCC *C* are nodes external to *C* which are antecedents of a justification (considered in the definition of the complete support graph) whose consequent is in *C* (i.e. $n; \exists n' \in C; n \in$ R-N-Sup(n')).

Definition 34. An *internal justification j* (with regard to a SCC *C*) is a justification such that its consequent is in *C* and all its antecedents are in *C* (j; InList(j)$\cup$OutList(j)$\subseteq C \wedge$ Cons(j)$\in C$).

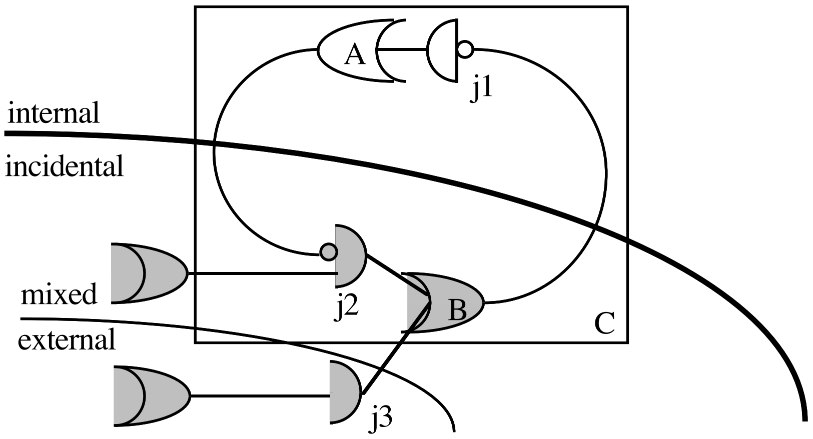


Figure 3. Examples of internal (*j1*), external (*j2*) and simply incidental (*j3*) justifications. Internal and incidental are exclusive while incidental and external are not.

Definition 35. An *incidental justification* (with regard to a SCC *C*) is a justification (considered in the definition of the complete support graph) such that its consequent is in *C* and (at least) one of its antecedents is external to the SCC: *j* is incidental to *C* if and only if $\exists n \in C$; $j \in$R-J-Sup(n) & $\exists n' \in$InList(j)$\cup$OutList(j); $n' \notin C$.

Definition 36. An incidental justification j is an *external justification* of C if no antecedent of j is in C ((InList(j)$\cup$OutList(j))$\cap C$={}), it is a *mixed justification* otherwise.

The following lemma is very important, it shows that the complete support graph is indeed complete with regard to supports.

Lemma 7. External incident nodes of C cannot have their labelling changed as a consequence of a change in C.

proof. Otherwise (there exists an external incident node n which changes its label) this means that either:
1° there exists $n'\in C$ such that $(n',n)\in E_c^+\cup E_c^-$ (if n passes from IN to OUT, n' accords with a valid justification, else, n' disaccords with an invalid justification).
2° there exists n'' such that $(n'',n)\in E_c^+\cup E_c^-$ and n'' changed (and recursively $n'\in C$ has been reached).

Thus, n would be in C, that which is opposed to the basic hypothesis. ◊

As a conclusion, whatever change occurs in C, it cannot come back to C through an external node. This is a first step in order to prove that multiple labellings of the graph arise from and only from multiple labellings inside some special SCC. Such SCC are characterised here together with the way they can be labelled.

Before examining the labellings of SCC, let see how to decompose the graph. A `FindSCC` procedure is used which takes a set of nodes as arguments and a tag (PSG, MSG, CSG, UG) indicating on which graph the SCC decomposition has to be processed (potential, minimal or complete support graph, unconstrained graph). The `FindSCC` procedure, is a slight variation of the SCC decomposition procedure given in [Aho& 74]. It is able, not only to return the set of SCC respecting their partial order, but also to provide the incident justifications and the parity of each SCC. The modifications take place in the second (backward) depth first search in which the incident justifications are recorded when encountered and the parity indicator changed when going through an OUT-list. As a consequence this algorithm still proceeds in O(n) where n is the number of nodes and justifications.

2.3. Labelling of stratified even SCC

It is now possible to prove that stratified even SCC of the complete support graph can only be labelled in one way and thus cannot generate multiple labellings.

Lemma 8. In a stratified even SCC, all nodes have the same label.

proof. This is obvious for unitary SCC. Otherwise, for every transition $(n_1,n_2)\in E_c^+$, such that $n_1,n_2\in C$, there is a justification $j\in$R-J-Sup(n_2) such that $n_1\in$InList(j) and Cons(j)=n_2.
1° If n_2 is IN then j is valid (because $j\in$R-J-Sup(n_2) and n_1 is accorded to j) so label(n_1)=IN.
2° If n_2 is OUT then j is invalid (same reason) and n_1 is unaccorded to j so its label is OUT.
◊

Lemma 9. Every SE-SCC has only one possible labelling.

proof. Two cases are possible:
1° Assuming that there exists an external valid justification or a premise justification, then, the node it reaches is IN and cannot be OUT, so every other node must be IN [lemma 8].
2° There is no external valid justification or premise justification. Assuming that every node is IN, then the labelling is not weakly grounded. Considering the definition, it is not possible to find a proper order $n_1,\ldots n_m$ between the nodes of the SCC: as a matter of fact, this order must have a least element which must be in C and have a valid justification independent of the validity of another node in C (they are all IN). Such a justification must be external: this is opposed to the basic assumption (2°), hence every node is labelled OUT. ◊

Theorem 10. Every SCC which is not an AE-SCC has only one possible labelling.

proof.

O-SCC are forbidden [working hypothesis 1].

SE-SCC only have one labelling [lemma 9]. ◊

As a consequence, multiple labelling generation can only take place inside an SCC but not inside unitary nor stratified SCC. Hence, only alternate even SCC are suited to be MLG.

2.4. Subsequent algorithms

It is already possible to compare the relative advantages of algorithms. Doyle's algorithm [Doyle 79] uses two steps, run on the whole dependency graph. It is as follows:

```
procedure Relabel( G: subgraph ) =              /* Doyle's version */
begin
   for all n∈G do LabelAndPropagate( n, G ) ;
   for all n∈G such that unlabelled( n ) do LabelAESCC( n ) ;
end
```

With

```
procedure LabelAndPropagate( n : node, S : SubGraph ) =
if Constrained( n )                              /* version 1 */
then
   Label( n ) ;
   S <- S \ { n } ;
   for all n' in Csq(n) ∩ S do LabelAndPropagate( n', S ) ;
endif
```

The first step is described later as the `UnConstrainedSubgraphs` algorithm. `LabelAESCC` is also presented later. This algorithm does not take into account the topology of the graph and acts almost blindly. As a consequence, its complexity on a linear graph (a connected graph with exactly one element without antecedent and one element without consequent and exactly one antecedent and one consequent for the others) is O(n^2). Goodwin, for his own, takes into account the properties presented above and propagates only once in the SCC graph, and for each SCC, applies algorithms for constraint propagation — labelling even cycles — and a backward algorithm (similar to Doyle's) which propagates an OUT label to the antecedents of a chosen node. So, his algorithm is a further step in the topological study of the graph.

```
procedure Relabel( G ) =                        /* Goodwin's version */
for all S∈FindSCC( G, MSG ) by order
do   for all n∈S do LabelAndPropagate( n, S ) ;
     LabelUnGroundedLoops( S ) ;
     with n∈S such that unlabelled( n )         /* Choice point */
     do   Partition( n, S, OUT, IN) ;
done
```

The procedures `LabelUnGroundedLoops` and `Partition` are presented below, but it can be noted that `LabelUnGroundedLoops` enables the labelling of stratified even cycles which are not supported (the others being labelled by `LabelAndPropagate`). [Euzenat 88] proposed an improvement of Goodwin's algorithm which tests, before examining a SCC, if the incident labelling changed, and, if not, does not change the labelling of the SCC. This algorithm allows to stop the propagation when it is not necessary any more.

A very similar algorithm to that of Goodwin can be proposed; in particular, it covers the graph in the SCC order and does not examine a SCC twice. Moreover, by distinguishing between alternate and stratified even SCC, it is able to trigger the adequate procedure in order to label the SCC. The unitary or stratified even SCC are labelled very quickly by applying the adequate procedure.

```
procedure Relabel( G ) =                        /* version 1 */
for all S∈FindSCC( G, CSG ) by order
do
   if stratifiedp(S)
   then
     if ∃J∈incidental-just(S) such that validep(j)
     then LabelStratified( S, IN );
     else LabelStratified( S, OUT );
     endif
   else
     with n∈S such that unlabelled( n )         /* Choice point */
     do   Partition( n, S, OUT, IN) ;
   endif
done
```

Again, the `LabelAESCC` procedure, dedicated to label alternate even SCC, is detailed below. Note that these algorithms do not assume that the SCC are recorded, they can calculate them or maintain them incrementally. The developments made so far are not very new in the development of RMS algorithms. They enable to partition the graph in SCC in order to propagate inside each SCC. But, it is now established that MLG are not in certain parts of the graph, and the other parts can be addressed.

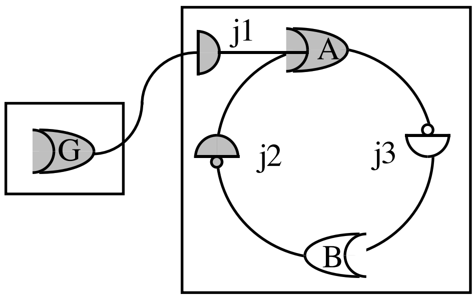
3. (Forward) constraints

The first decomposition of the dependency graph in SCC of the complete support graph provided a first circumscription of the MLG: they come from inside the AE-SCC. But some questions remain unanswered. The first question to ask is what in an AE-SCC is really involved in the generation of multiple labelling. The answer is that the MLG is not the whole AE-SCC. This is shown with the help of the (forward) constraint notion. A (forward) constraint is what constrains a node to have a certain label. Of course, in a SCC, (forwardly) constrained nodes cannot change their labelling. So, first, they can be labelled with a (forward) constraint propagation procedure, and, second, they are not involved in MLG.

The characterisation of multiple labellings of the graph is operational and is thus presented through algorithms. Moreover, these algorithms can be used in order to actually label the graph. The complete support graph is used as before and a particular SCC of that graph is considered taking into account the partial labelling which has been performed on the antecedent SCC in the SCC order. So, keep in mind that external incident nodes are already labelled but nodes in the SCC are not. The process of labelling the nodes of the SCC, letting apart its current labelling, is emphasised. Therefore each node in the SCC is considered as unlabelled (some other authors clearly consider three labels: IN, OUT and UNKNOWN).

The problem is the following: at a given moment, a static labelling has been considered together with its complete support graph. Now, a partial labelling with its influence on the remaining part of the graph is considered. There are less constraints on that part of the graph than before: the constraints internal to a precise SCC are not taken into account, this is depicted on figure 4.

Figure 4i. In the complete support graph, *A* is supported by *G* and *B*.



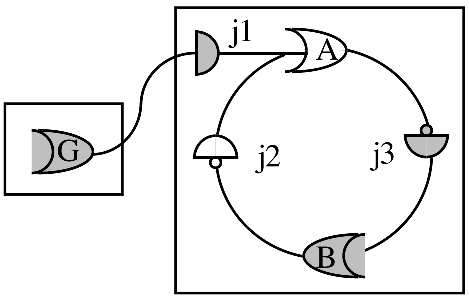
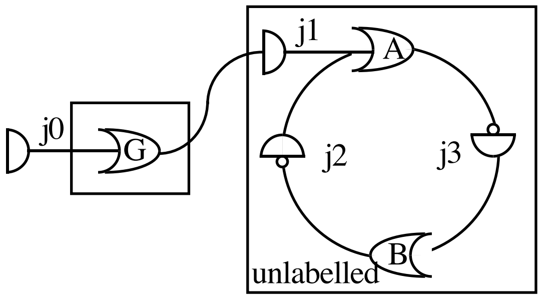


Figure 4ii. During the labelling process, *j1* is not constraining *A* so $\{A, B\}$ is an initial subSCC (see below) of itself and it can generate two labellings.

Figure 4iii. In this other labelling process (after adding *j0*), *j1* being valid is constraining *A* to be IN.



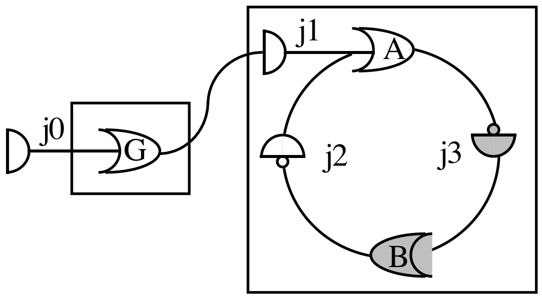


Figure 4iv. The constraint propagation algorithm will label {*A*, *B*} by only considering the constraints on each node.

Hence, the complete support graph is a constraining graph but it takes into account circular constraints inside a SCC. So it is useful for propagating from a SCC to another but not for the propagation inside a SCC which is loosely constrained. Thus, a notion of (forward) constraint which arises during the labelling process of a SCC is put forward and sets of nodes which are dependent of others and mutually constrained is isolated: the MLG.

The notion of (forward) constraint leads to partition the graph into smaller sub-graphs which are called subSCC. This second level of examination of the graphs, more precise than SCC, reveals multiple labelling generators.

As before, SCC decomposition is used. But it is used against the unconstrained part of the graph. Thus, the considered graph is a graph G=<V, E> in which the set of vertices V is a set of (unlabelled) nodes and the set of edges follows the (forward) constraint relation between nodes. So, again, for each SCC C corresponds unambiguously a set of nodes of the dependency graph and SCC is used that way.

The departure point of the section is that of the RMS labelling process, i.e. considering a part of the graph whose labelling is grounded and will not be changed, and a part of the graph which is currently unlabelled. (Forward) constraints on the graph are defined. They are used on the unlabelled graph with regard to the labelled graph. According to these (forward) constraints, the unlabelled nodes of the graph can be labelled, then, what remains is decomposed in SCC of the remaining graph.

3.1. Constraint definitions

The problem concerns the definition of sub-graphs which can change their labellings. This is not the case for every AE-SCC, so the (forward) constraints are defined which avoid the SCC from switching to another labelling. The question to be answered is what can (forwardly) constrain the validity of some nodes of a SCC and does not depend of the SCC itself. It is, of course, incidental justifications and external nodes.

Definition 37. A node is *(forwardly) constrained* if and only if each of its justifications has an unaccorded node or it has a valid justification.

Consequence. A node in an unlabelled SCC *C* is *(forwardly) constrained* if and only if each of its justifications has an unaccorded external node, or it has an external valid justification.

Definition 38. *(forward) Constraining justifications* of a SCC *C* are constraining justifications of the nodes in *C*.

Definition 39. *(forward) Constraining nodes* of a SCC *C* are:

- (external) (accorded) antecedents of valid external justifications,
- external unaccorded antecedents of invalid mixed justifications,

of a (forwardly) constrained node in *C*.

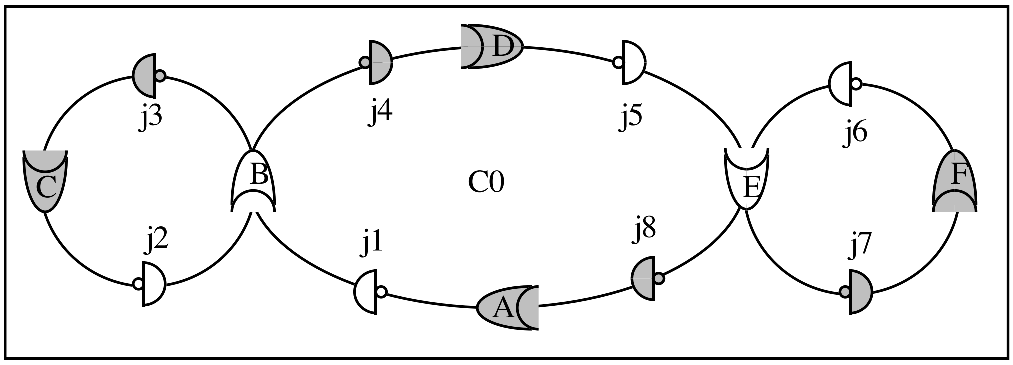


Figure 5. One of the previous labelling of figure 2, the whole graph is one SCC *C0*.

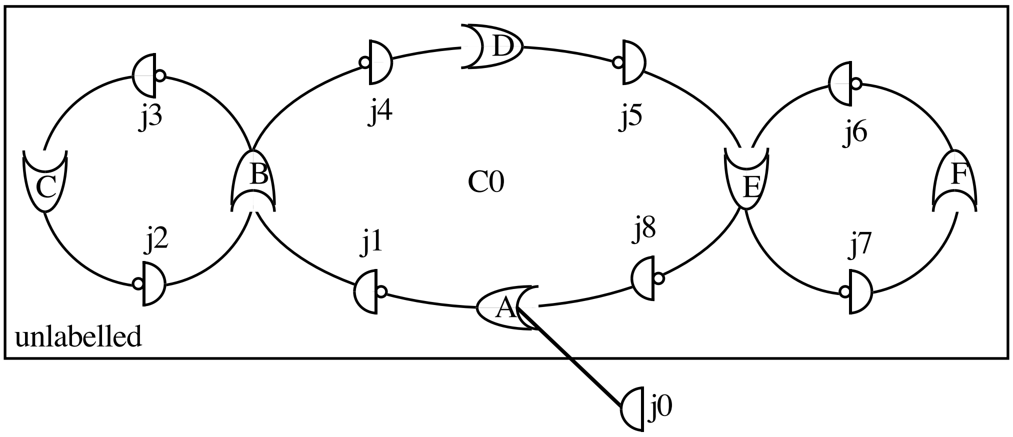


Figure 6. A new justification j0 requires label propagation through the graph. According to the former statements, the (alternate even) SCC has to be labelled at once.

This first part of Doyle's algorithm is enough in order to propagate each constraint on the SCC an let unlabelled the unconstrained parts of the SCC. It is noteworthy that its blindness can be corrected by the fact that:

- It is called on a fragment of the graph (the SCC).
- It can be run only on constraining nodes that which reduces the number of nodes examined in vain.

3.2. Unconstrained subSCC

After applying the (forward) constraint propagation step, some nodes of the SCC might remain unlabelled. These nodes can be isolated in a sub-graph.

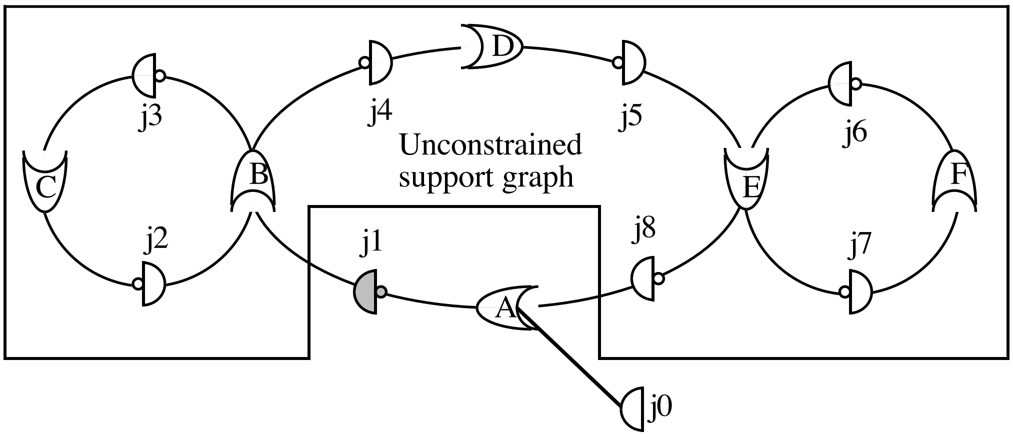


Figure 7. The unconstrained support graph of the SCC *C0* after propagating the only constraint (*j0*).

Definition 40. An unconstrained subSCC (called subSCC for short) is a SCC of the complete support graph minus (forwardly) constrained nodes (and justifications).

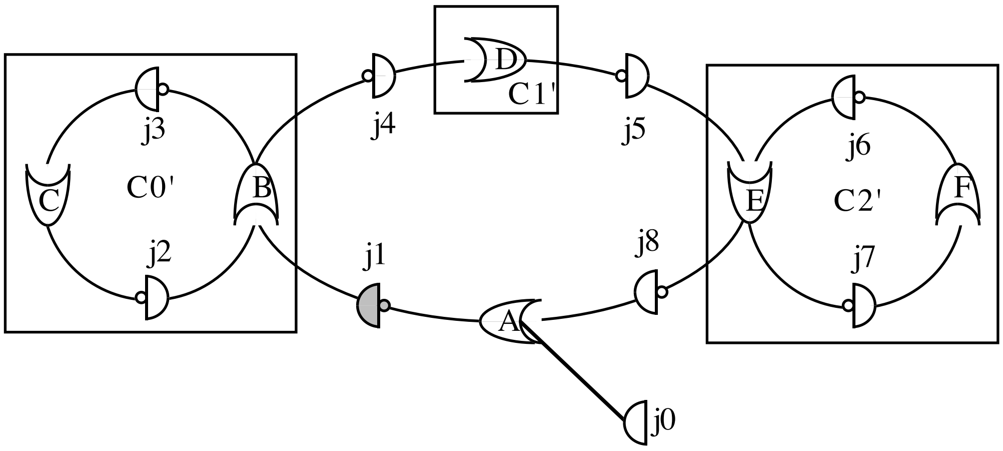


Figure 8. The three (*C0'*, *C1'*, *C2'*) subSCC of the unconstrained sub-graph of figure 7.

These subSCC are simply computed on an abstract graph, in which only unlabelled justifications whose consequent is unlabelled are considered. A justification is unlabelled when one of its antecedents is unlabelled and each labelled antecedent is accorded to it. This graph is called Unconstrained Support Graph (USG). Once again, apparent circularity stops with justifications without antecedents and nodes without justifications: nodes and justifications which are part of a circularity or consequences of such a circularity remain unlabelled.

The process of determining successive subSCC is analysed in the following. For that purpose, one procedure needs to be explained: the `Label` procedure which labels a (forwardly) constrained node according to its justification validity.

```
procedure Label ( n : node ) =
if ∃j∈L-Just[n] such that valid(j)
then   Label[n] <- IN ;
else   if ∀j∈L-Just[n], invalidep(j) then Label[n] <- OUT endif
endif
```

Lemma 11 (Local groundedness and closedness of `Label`). `Label` satisfies the properties of local groundedness and closedness.

proof. A node is labelled IN by `Label` if and only if there is a valid justification for it (local groundedness) and a node with a valid justification cannot be labelled OUT (closedness).

◊

`Label` also builds an implicit grounding order which can be used in proof of global groundedness. For a node labelled IN with the help of a justification j=<I O>:n, $\forall n' \in I \cup O$, $n' \sqsubset n$ and for a node labelled OUT and for each n' disaccording with a justification of n, $n' \sqsubseteq n$.

Now, an algorithm for determining unconstrained subSCC of a SCC can be given. It only determines cycles by eliminating non circular arguments (this is the aim of Doyle's first step procedure). Such an algorithm can recursively reduce the unconstrained sub-graph by first considering the whole SCC as a sub-graph and then eliminating (forwardly) constrained nodes of the sub-graph by labelling them. Two versions of the same algorithm are given below. The first one tries to label each node and if it is able to do it, labels the (forwardly) constrained consequents of the last labelled node.

```
function UnConstrainedSubgraphs ( S : SubGraph ): SubGraphSet =
begin                                        /* spreading */
  for all n in S do LabelAndPropagate( n, S ) done ;
  return( FindSCC( S, UG ) ) ;
end
```

The second algorithm tries to label each new (forwardly) constrained node but does not propagates (forward) constraints to other nodes. It iterates that process until there is no more nodes labelled.

```
function UnConstrainedSubgraphs ( S : SubGraph ): SubGraphSet =
begin                                               /* iterative */
  Sr <- {{}} ; S' <- {S} ;
  while S'≠ Sr
  do
    Sr <- S' ; S' <- {} ;
    for all s in Sr
    do  for all n in s such that Constrained( n )
        do Label( n ) ; s <- s\{n} done ;
        S' <- FindSCC( s, UG ) ∪  S';
    done
  done ;
  return( S' )
end
```

These two programs are equivalent (this is stated without proof). The first one is only the first step of Doyle's algorithm. The second one can be seen as computing a fix-point on a set of sub-graphs. The algorithm `UnConstrainedSubgraphs` can be applied when faced with a set of nodes to label, in order to determine which of these nodes can be labelled and which ones cannot, these last ones are returned decomposed in subSCC of the first set of nodes. What is returned by both algorithms is the set of SCC of the remaining unconstrained sub-graph.

Lemma 12 (Local correctness of `UnConstrainedSubgraphs`). Considering the initial labelling as given (as it is in this subsection), `UnConstrainedSubgraphs` provides a strongly grounded labelling of the nodes it labels.

proof.

1° The labelling is admissible because it is done according to `Label` which respects closedness and local groundedness properties [lemma 11].

2° If the labelling before the SCC is considered as strongly grounded this means that there exists an order between nodes preceding the SCC which respects strongly grounded constraints. On the other hand, for each node labelled by `UnConstrainedSubgraphs`, the node is (forwardly) constrained. So, its (forward) constraining nodes can be found and an order can be built such that these (forward) constraining nodes are preceding the currently labelled one. Since the `UnConstrainedSubgraphs` procedure never comes back to a labelled node, this is really an order between these nodes and while the (forward) constraints are expressing those of strongly grounded definition, this is a strongly grounded labelling. ◊

This theorem just means that `UnConstrainedSubgraphs` obeys a strongly grounded order as far as there exists one. Thus, if the external labelling is strongly grounded, then its concatenation[2] with the `UnConstrainedSubgraphs` order is strongly grounded while if it is weakly grounded, the concatenation is weakly grounded.

Consequence. The labelling given by `UnConstrainedSubgraphs` is unique.

[2] The concatenation R::S of two binary relations R and S such that the domain of R is disjoint from the co-domain of S corresponds to the transitive closure of the sum of the two relations R+S. The concatenation of two partial order is a partial order.

proof. by theorem 2. ◊

3.3. Ungrounded loops

James Goodwin clearly went further on by detecting ungrounded loops inside an alternate even SCC and labelling them as OUT. These ungrounded loops are the nodes which *cannot* be labelled IN. For that purpose, the `LabelUnGroundedLoops` algorithm finds the transitive closure of the nodes which *can* be labelled IN independently of themselves. Consequently, the others cannot and are labelled OUT. This is done as far as the algorithm labels some more nodes since the potential cause of groundedness of some nodes may have disappeared by labelling other nodes OUT.

```
procedure LabelUnGroundedLoops( S: SubGraph ) =
begin
   TOPROCEED <- S; FOUNDABLE <- {};
   while ( FOUNDABLE ≠ TOPROCEED )
   do   OLDFOUNDABLE <- TOPROCEED ;
        while ( FOUNDABLE ≠ OLDFOUNDABLE )
        do   OLDFOUNDABLE <- FOUNDABLE ; FOUNDABLE <- {};
             for all n∈TOPROCEED\OLDFOUNDABLE;
                                ∃j=<I O>:n∈J;
                                  ∀n'∈I, label[n']=IN ∨ n'∈OLDFOUNDABLE
             do FOUNDABLE <- FOUNDABLE ∪ {n'} ;
        done ;
        for all n∈TOPROCEED\FOUNDABLE do LabelAndPropagate( n, OUT ) ;
        TOPROCEED <- {n∈TOPROCEED; unlabelled( n )} ;
   done
end
```

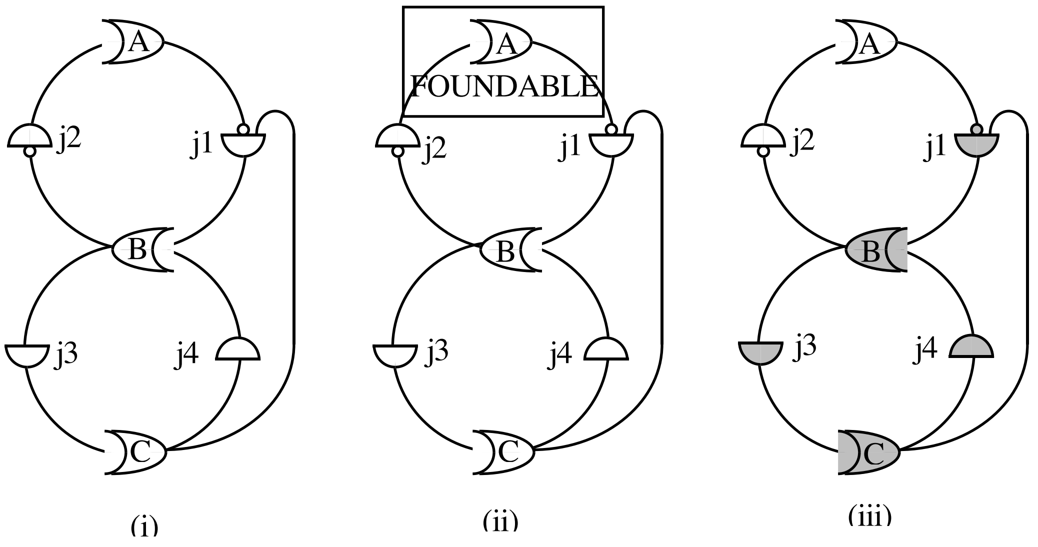


Figure 9. The graph (i) is an unconstrained SCC. After a first pass, the `LabelUnGroundedLoops` algorithm finds that *A* is the only node which can be IN (ii). So, it labels both *B* and *C* as OUT. As a consequence of this, *A* is be labelled IN due to *j2* being valid (iii).

The `FOUNDABLE` variable of `LabelUnGroundedLoops` contains the set of nodes for which it is possible to find a grounding order that support their (possible) INness. Hence, if some nodes are not in this set they can only be labelled OUT in a weakly grounded labelling. This algorithm does not only label SE-SCC but also some stratified even loops embedded in an alternate one such as those of figure 9.

It is noteworthy that the resulting set of unlabelled nodes is not necessarily connected. Since the initial SCC can have been broken by labelled nodes. This is the case for figure 10.

The consequence of that algorithm is that remaining nodes cannot be labelled in a unique way (otherwise they would be by the previous algorithms) and that it is possible to find a grounding order for which any node can be IN (otherwise they would be labelled OUT by that algorithm). This is ensured by the fact that what remain to label are still alternate even SCC without odd-cycles. This needs to be, because otherwise, unitary SCC would have been labelled by `LabelAndPropagate` for unitary SCC and by `LabelUnGroundedLoops` for stratified SCC.

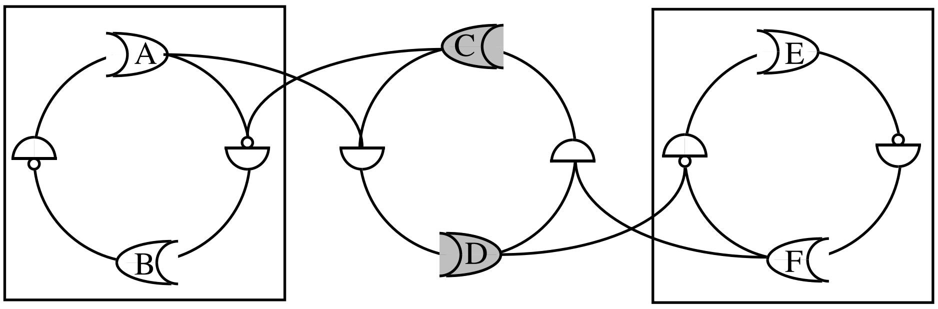


Figure 10. This graph is only one unconstrained SCC, but after the labelling produced by `LabelUnGroundedLoops`, it results in two disconnected components: $\{A, B\}$ and $\{E, F\}$. If a justification <{}{*F*}>:*B* is added, the graph is still connected (from {*E*,*F*} to {*A*,*B*}) but not strongly connected.

Therefore, after `LabelUnGroundedLoops`, any remaining node can either be labelled OUT or IN (but not independently).

Lemma 13. Considering the initial labelling as given, `LabelUnGroundedLoops` finds a strongly grounded labelling of the graph it labels.

proof. The inner while part of `LabelUnGroundedLoops` finds the transitive closure of the nodes which can be labelled IN with the help of IN-antecedents labelled IN independently of themselves. These nodes could, in some ways, be labelled IN in a weakly grounded labelling and the other cannot because all their possible IN-labelling relies, at least, on themselves being IN. So, the others are labelled OUT by the subsequent part of the algorithm. This labelling is strongly grounded since:

- groundedness: the nodes labelled OUT cannot be labelled IN independently of their own labelling.
- closedness: the only potential causes of INness of the nodes are dependent of the INness of themselves; since the nodes are labelled OUT (and since there is no odd cycles) closedness is ensured.

The order constructed by that inner while loop is such that for any two nodes *n* and *n'* labelled OUT, $n \sqsubseteq n'$ (and vice versa). This order can be concatenated to the initial order.

At the end of that first part, `LabelUnGroundedLoops` uses `LabelAndPropagate` which uses `Label` in turn which establishes a locally grounded and closed labelling. It also constitutes a strongly grounded labelling since the antecedent orders are strongly grounded independently of the still unlabelled nodes.

After the while loop, the TOPROCEED variable contains the remaining unlabelled nodes. When everything is labelable IN, `LabelUnGroundedLoops` stops its work on that set of nodes. The `LabelUnGroundedLoops` procedure runs that loop until it cannot find any more nodes to label. The resulting order is a strongly grounding order that can be concatenated with the preceding ones. ◊

3.4. Unlabelable SCC

It is possible to modify the previous algorithm in order to take a full advantage of the topology of the graph. This can be done with the help of more SCC decomposition. The unlabelled SCC are unconstrained SCC of the graph without ungrounded loops. Such SCC can be determined incrementally by the following algorithm.

```
function UnLabelableSubGraphs( S: SubGraph ): SubGraphSet =
begin
  RESULT <- {}; REMAINDER <- UnConstrainedSubgraphs:used( S );
  while ( REMAINDER ≠ {} )
  do
    TOPROCEED <- First( REMAINDER ):
    REMAINDER <- Rest( REMAINDER );
    FOUNDABLE <- {};
    while ( FOUNDABLE ≠ TOPROCEED )
    do   OLDFOUNDABLE <- TOPROCEED ;
         while ( FOUNDABLE ≠ OLDFOUNDABLE )
         do   OLDFOUNDABLE <- FOUNDABLE ; FOUNDABLE <- {};
              for all n∈TOPROCEED\OLDFOUNDABLE;
                                ∃j=<I O>:n∈J;
                                  ∀n'∈I, label[n']=IN ∨ n'∈OLDFOUNDABLE
              do FOUNDABLE <- FOUNDABLE ∪ {n'} ;
         done ;
         for all n∈TOPROCEED\FOUNDABLE do Label( n, OUT ) ;
         if ( TOPROCEED ≠ FOUNDABLE )
         then
           REMAINDER <- UnConstrainedSubgraphs( FOUNDABLE)
                          ∪ REMAINDER;
           TOPROCEED <- First( REMAINDER );
           REMAINDER <- Rest( REMAINDER );
         else
           RESULT <- RESULT ∪ { TOPROCEED };
           REMAINDER <- UnConstrainedSubgraphs( First( REMAINDER ) )
                          ∪ Rest( REMAINDER );
         endif
    done
  done;
  return( RESULT )
end
```

`UnLabelableSubGraphs` is just a juxtaposition of `LabelUnGroundedLoops` and `UnConstrainedSubgraphs`; from the former, it uses the detection of ungrounded loops and from the latter, it uses the successive SCC decomposition which help to restrict the number of nodes on which the procedure is run. Therefore, `UnLabelableSubGraphs` also build a weakly grounded labelling.

Lemma 14. Considering the initial labelling as given, `UnLabelableSubGraphs` finds a strongly grounded labelling of the graph it labels.

proof. `UnLabelableSubGraphs` is no more than `LabelUnGroundedLoops` plus `UnConstrainedSubgraphs`. Since they find a strongly grounded labelling of the graph they label [lemma 12 and 13], so does `UnLabelableSubGraphs`. ◊

Corollary 15. There is no other labellings in which the initial labelling is the same and the labelling given by `UnLabelableSubGraphs` is different.

proof. by [lemma 14 and theorem 1]. ◊

This algorithm processes a lot of SCC decomposition because after a first such decomposition, it labels some more nodes and this could lead to a new decomposition. This seems to be very heavy, but it is restricted since, the decomposition is run on smaller units and it is limited to the re-decomposition of the first SCC of the current `REMAINDER`. Anyway, it is not to be used in a real implementation, but is presented here for the sake of modularity of the presentation. A really efficient algorithm just needs the initial unlabelable subSCC and can compute them on the fly as soon as previous initial unlabelable subSCC are labelled.

After using `UnLabelableSubGraphs`, a partially ordered set of unlabelled SCC is provided. Some properties are expected from these sub-graphs.

Lemma 16. SubSCC which are unitary are labelable in one step when all their antecedents are labelled.

proof. If all the antecedents of the SCC are labelled, then the subSCC (and the node in it) is not any more unconstrained: every incident justification can be labelled and thus also the node. This is done by `UnConstrainedSubgraphs`. ◊

Definition 41. SubSCC with no unlabelled antecedents are called initial subSCC.

As for the analysis above, subSCC can be partitioned into stratified and alternate even subSCC.

Lemma 17. Initial subSCC which are stratified are labelled in one step when all their antecedents are labelled and they are globally labelled OUT.

proof. In order to prove this, it is enough to re-use the case 2 of the proof of lemma 9. As a matter of fact, this is the only case which applies (the first case considers that a node in the subSCC is (forwardly) constrained). This is done automatically by `LabelUnGroundedLoops`.
◊

Lemma 18. Initial unlabelable subSCC are alternate even SCC.

proof. They have no unlabelled antecedents and the lemmas above [lemma 16 and 17] show that if they were unitary or stratified, they would be labelled by the `UnLabelableSubGraphs` algorithm. ◊

As a consequence, the only part of the graph which needs a non deterministic algorithm in order to be labelled are these initial unlabelable subSCC. Thus, it is unquestionable that initial unlabelable subSCC are multiple labelling generators (MLG, they are referred that way in the remainder). Some remarks can be made concerning the generation and influence of labelling these MLG:

- Labelling a MLG can lead to a re-decomposition of parts of the MLG itself and thus generate new MLG.
- Non initial unlabelable SCC can either become MLG or being partially or totally labelled after the labelling of their antecedents MLG.
- Last, all the MLG have several possible labellings and the two former points are dependent of the chosen labelling of the MLG.

Thus the MLG are inherently dynamic entities dependent of the current partial labelling.

4. Labelling AE-SCC

The problem is now essentially in labelling the AE-SCC. The available algorithms is presented and discussed under the light of the three properties mentioned above (and worth remembering):

- *Correctness*: the program must respect the specifications of RMS (finding a weakly grounded labelling).
- *Potential completeness (*or *non deterministic completeness)*: the program must be able to find any of the weakly grounded labellings.
- *Efficiency*: the program must have the lower complexity as possible.

Since, the important (and unusual) property here is the potential completeness, it is worth noticing the advantages of complying with this property in the real use of RMS:

- Usually RMS are used in a wider environment which triggers them in order to find a labelling. But, it is often interested in directing the RMS toward the more “preferred” labelling. It is thus required that the algorithm be able to find these preferred labelling.
- RMS use backtracking algorithms in order to find a consistent weakly grounded labelling. Such an algorithm starts from a current weakly grounded labelling and, if not consistent, adds some justifications in the graph in order to obtain a consistent weakly grounded labelling minimising the difference between labellings. It would be better to try to find first another weakly grounded labelling of the graph which is consistent (so minimising first the difference in the *graphs*). If this fails, it could the be possible to use a classic

backtracking algorithm. Thus, once more, it is preferable to ensure that any of the weakly grounded labelling can be found.

- The three last reasons have to do with completeness more than potential completeness. First it would be useful to provide to the prover all the weakly grounded labellings one after another.
- A potentially complete algorithm is, in fact, a first step toward a complete algorithm. Such an algorithm had already been given by Ulrich Junker and Kurt Konolige [Junker& 90]. This algorithm is an enumerative algorithm which hardly use the topology of the graph.
- Last, but not least, this work is part of a larger project dedicated to incrementally maintain all the weakly grounded labellings of a graph [Euzenat 91]. It is difficult to deal with such a problem in an efficient way. One proposal that calls for examination is the propagation through the graph of labels representing the configurations of "multiple labelling generators" under which nodes must be labelled IN.

The term "potential completeness" is to be understand as a more practical and mechanism-oriented than "non-deterministic complete" which is, although equivalent, more theoretical.

Doyle and Goodwin's algorithms are first considered before turning to tentative solutions of the problems they raise.

4.1. Doyle's algorithm: ensuring correctness

Doyle's algorithm uses a `LabelAESCC` procedure which is called as follows:

```
for all n such that unlabelled( n ) do LabelAESCC( n ) ;
```

and reads as follows:

```
procedure LabelAESCC( n: node ) =
begin
   if ∃j∈L-Just[n] such that ∀n'∈InList[j] label[n']=IN
                             & ∀n'∈OutList[j] label[n']≠IN
   then
      Label( n ) ;
      for all n'∈Csq( n ) do label[n] <- nil;
   else
      Label[n] <- OUT ;
   endif ;
   for all n'∈Csq( n ) such that unlabelled( n' ) do LabelAESCC( n' )
end
```

The algorithm tries to label a node by considering that the current unlabelled nodes is labelled OUT. It can happen that a node which had been considered as OUT became IN: then, the graph must be re-labelled. It is noteworthy that, in the procedure, the consequences considered in the else part are the effective consequences, i.e. those whose current label is supported by the absence of label for the node.

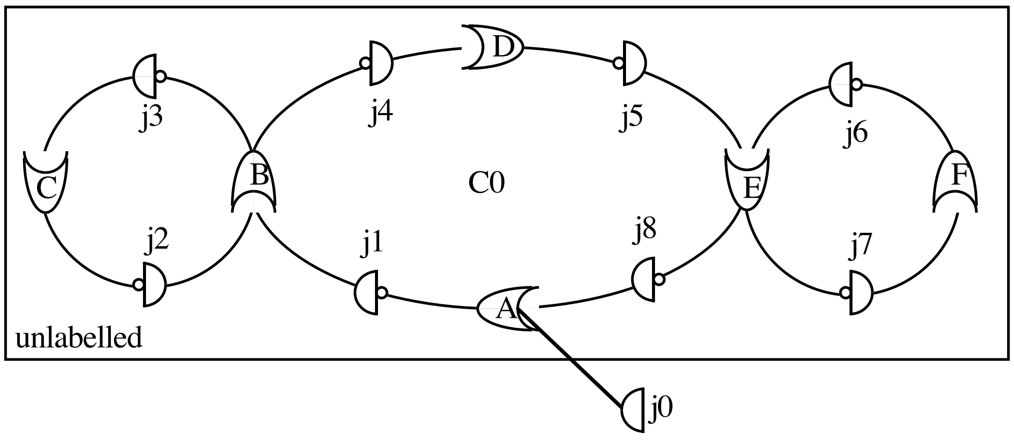


Figure 11. A dependency graph to be labelled after the addition of *j0*.

This algorithm is exemplified here on the example of figure 6 (of course, the choices are made in order to demonstrate the worst cases of the algorithm). The algorithm, after addition of *j0*, works as above (problems are signalled by ‡):

1- Every node is a consequent of *A* and, so, is unlabelled.

2- All nodes are in the same SCC.

3- Justifications are checked for validity in the following order:

- *j0* is chosen first: *j0* valid, *A*: IN
- *j1* is chosen because each of its antecedents is labelled: *j1* invalid.
- Now each justification (*j2-j8*) is candidate, let choose *j8*: *j8* valid, *F*: IN.
- *j6* is now constrained as its predecessor (*F*) is IN, it is chosen: *j6* invalid.
- At new each remaining justification (*j2-j5*, *j8*) can be chosen. Let choose *j5*: *j5* valid, *E*: IN. Here is the problem at first ‡: in order to ensure correctness it is necessary to propagate again to *j7* and *F*.
- *j8* is now constrained: *j8* invalid.
- Again, each justification (*j2-j4*) can be chosen. Let choose *j4*: *j4* valid, *D*: IN. Here comes the problem again ‡: it is necessary to propagate through *j6* and *E* and *j8* and *j7* and *F*.
- At last, each justification (*j2-j3*) is candidate. Let choose *j2*: *j2* valid, *B*: IN. Here comes the last problem ‡: the propagation spreads again through *j4* and *D* and *j6* and *E* and *j8* and *j7* and *F*.
- *j3* is constrained: *j3* invalid, *C*: OUT.

So, this algorithm is burdened by time expenses, because it does not use a proper order for labelling nodes.

4.2. Goodwin's algorithm: ensuring efficiency

The part of Goodwin's algorithm labelling AE-SCC is now explained with its problems and advantages. This algorithm recovers from the complexity of Doyle's by two means:

- First, by the graph decomposition into SCC which allows to explore it with a proper, although partial, order.
- Second, by finding the nodes which cannot be IN (with `LabelUnGroundedLoops`) and then by labelling other nodes in a fast step (the “Swedish colouring” algorithm).

Goodwin’s algorithm is as follows:

```
procedure Relabel( G: graph ) =              /* Goodwin’s version 2 */
for all S∈FindSCC( G, MSG ) by order
do   for all n∈S do LabelAndPropagate( n, S ) ;
     LabelUnGroundedLoops( S ) ;
     while ( ∃n∈S; unlabelled( n ) )
     do                                       /* Choice point */
        with n∈S such that unlabelled( n ) do Partition( n, S, OUT, IN) ;
     done
done
```

Afterwards, the algorithm chooses a node to label OUT and label the rest of the nodes with the help of the “Swedish colouring” algorithm. The “Swedish colouring” algorithm propagates backward constraints in order to label the whole graph:

```
procedure Partition( n: node, S: SubGraph, l1, l2: label ) =
if n∈S then
   if labelled( n )
     then if label( n ) = l2 then signal(‘odd-cycle’) endif
     else
        label[n] <- l1 ;
        for each <I O>:n∈𝒥
        do   for all n’∈I do Partition( n’, S, l1, l2);
             for all n’∈O do Partition( n’, S, l2, l1)
        done;
        for each <I O>:n’∈𝒥; n∈I do Partition( n’, S, l1, l2);
        for each <I O>:n’∈𝒥; n∈O do Partition( n’, S, l2, l1);
     endif
endif
```

The presented algorithm is James Goodwin’s concrete (i.e. implemented) algorithm. It is an improvement from Doyle’s one but does not correspond to its abstract algorithm on which he proved properties. Our work shares a lot with Goodwin’s abstract algorithm; in particular, there are similar concepts which do not share the same name:

- A constraint is called a relaxation (condition).
- A subSCC is called an unlabelled SCC, and an initial subSCC is called a minimal unlabelled SCC (MUSCC).

The backward propagation ensures closedness since it warrants that nodes labelled OUT do not have any support and it ensures groundedness since any node labelled IN has a support. Namely, for each node to be labelled IN it labels all of its unlabelled antecedent justification as valid, and thus all their antecedents have to comply the IN- or OUT-list to which they belong. For a node to be labelled OUT it labels all of its antecedent justifications as invalid and all their antecedents to mismatch the IN- or OUT-list to which they belong. Since, at the end of

`LabelUnGroundedLoops`, the SCC is not any more closed, it is necessary to propagate forwardly in order to reach all the nodes in the former SCC which can be labelled (since this was an SCC, the labelling is weakly grounded). For the same reason, the `Relabel` algorithm run `Partition` against each unlabelled node. If the SCC had remained connected, then one call to `Partition` would have been enough.

The idea fully used by James Goodwin is backward constraint propagation. On the example of figure 9, the `LabelUnGroundedLoops` procedure cannot label any node. Suppose that Goodwin's algorithm chooses the same option as the former algorithm. It works that way:

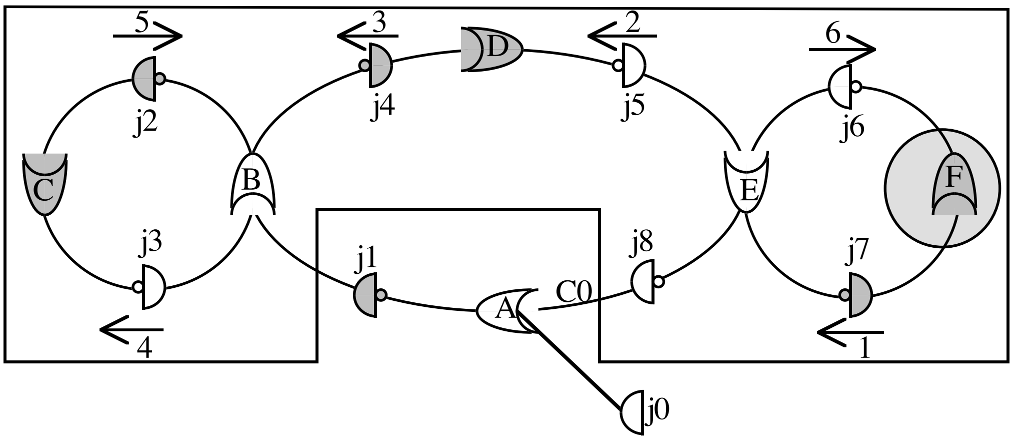


Figure 12. The labelling process of Goodwin's algorithm after *F* as been chosen to be OUT.

The behaviour of Goodwin's algorithm is traced below on the example of figure 6 again; the trace starts at step 3 of Doyle's algorithm since the former steps render the same results. The symbol (†) signals a problem to be discussed later on.

3- Justifications are checked for validity in the following order:

1 *F* is chosen to be OUT. *j7* must be invalid.

2 *E* must be IN. The algorithm (over-)ensures (†) it by declaring both *j6* and *j5* valid.

3 So *D* must be OUT and *j4* invalid.

4 *B* must be IN, so *j3* must be valid.

5 *C* must be OUT so *j2* must be OUT.

6 Back to *j6*, it (must) be valid and it is.

That way, i.e. by going backward, Goodwin's algorithm ensures that a weakly grounded labelling according with its choice of *F* to be OUT is found.

Theorem 19. Goodwin's algorithm finds a weakly grounded labelling of the SCC it labels `Relabel`.

sketch of proof (see [Goodwin 87]).

1° The whole SCC is labelled (there is a `Partition` call for each connected part of the SCC).

2° If there is no odd-cycle this labelling is weakly grounded:

. it is closed because each node labelled OUT has all of its justifications invalid (ensured by the recursion step): there is no INing and OUTing.

. it is weakly grounded because for each node labelled IN the propagation finds an ordered support since it only comes back to the node by encountering an OUT-list (this is true because, otherwise, it would have been labelled OUT by `LabelUnGroundedLoops`).

◊

Note that this theorem holds whichever be the node chosen to be labelled OUT in the first place. Therefore, the following corollary holds.

Corollary 20 (Goodwin's property). After the `LabelUnGroundedLoops` procedure, for any node of the remaining unlabelled initial subSCC, there exists a weakly grounded labelling in which the node is OUT.

Lemma 21 is introduced in order to facilitate the proofs of potential completeness; it can be omitted in first reading.

Lemma 21. A weakly grounded labelling of a MLG contains at least one node labelled OUT, and for each node of the MLG there exists a weakly grounded labelling in which it is OUT.

proof. First note that MLG are AE-SCC [lemma 18]. If each node is labelled IN and the labelling is weakly grounded, there is a strict ordering of the nodes $n_1 \sqsubset \ldots \sqsubset n_m$ with each a valid justification. Thus, n_1 would have a valid justification independently of every other node of the MLG and so, it is constrained and should not make part of this MLG (and should have previously been labelled by `UnConstrainedSubgraphs`). Thus, there is, at least, one node labelled OUT in each MLG.

Now it is possible to use Goodwin's `Partition` algorithm against that SCC. Since it is strongly connected, it is enough to propagate only backwardly and all nodes in the MLG will be reached. If `Partition` is run on any node with an OUT label it finds a weakly grounded labelling (otherwise there is an odd loop). ◊

But there is a remaining problem (noted †) with this algorithm: it always finds the same labelling if F is chosen OUT, but this is not the unique labelling with F OUT. The labelling of figure 13 is also valid:

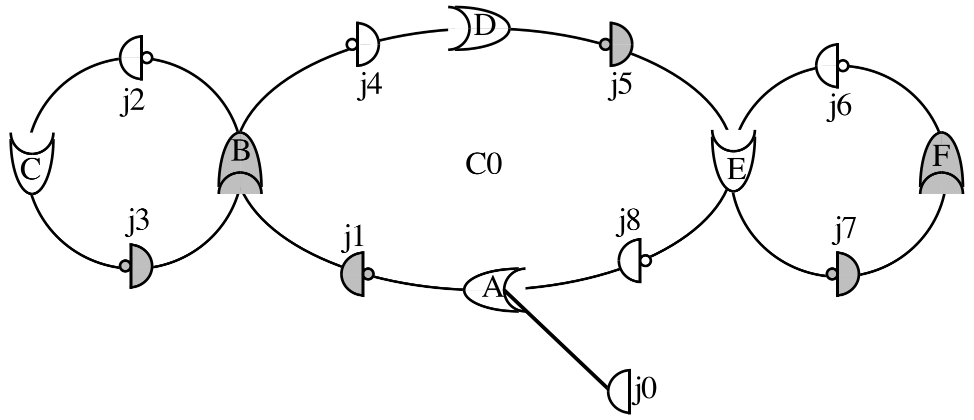


Figure 13. This labelling cannot be found with Goodwin's algorithm (if *F* has been chosen as OUT).

For each MUSCC, Goodwin's algorithm only uses one choice instruction. It chooses a node to label OUT and automatically propagates this choice toward the graph to other nodes in the SCC. This is achieved with the help of the "Swedish colouring" of the graph. But there is not only one way to do this, and Goodwin's algorithm is only able to do it in one way because it does not have another source of non determinism. In fact the "Swedish colouring" constraint is too coercive. See figure 14 and 15 for evidence of this.

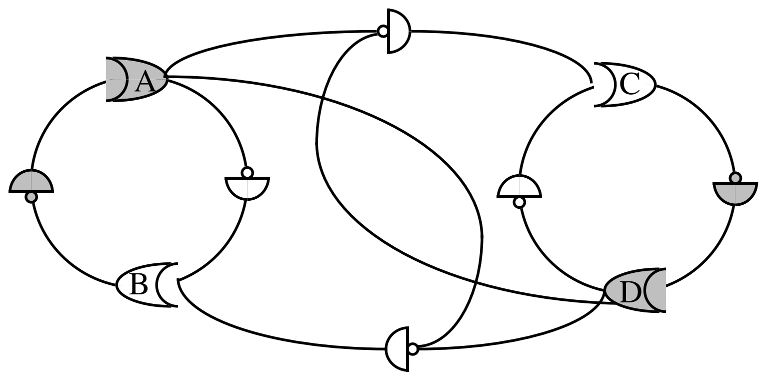


Figure 14. A graph and one of its "Swedish colouring". The other admissible colouring is exactly the opposite one ({*A*,*D*} instead of {*C*,*B*}).

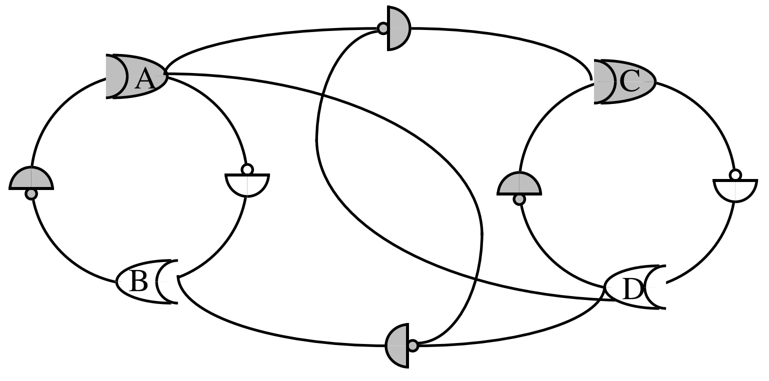


Figure 15. A labelling which is weakly grounded as well but is not a "Swedish colouring" of the graph. This is also true for labelling {*C*,*A*}.

4.3. Lokhorst's algorithm: ensuring potential completeness

In order to overcome this problem it is necessary to relax the "Swedish colouring" constraint. Partially for insuring potential completeness, Henk Lokhorst recently proposed the relaxation of the former algorithm [Lokhorst 92]. The algorithm avoid over-constraining by constraining only what is necessary. As a consequence, the MLG is not labelled in only one step like for Goodwin's algorithm, and it has to be re-iterated on the remaining unlabelled part of the MLG (which becomes a or several new MLG).

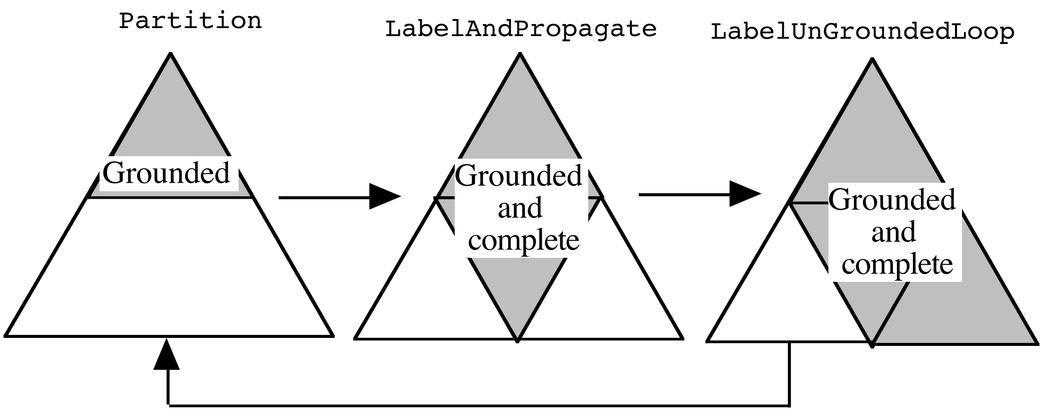


Figure 16. The process of Lokhorst's algorithm which grounds the choice that has been made in a MLG. Then, since some parts of the MLG can be unlabelled, it re-iterates the process by propagating (having then a grounded and complete part of the graph labelled) and labelling the remaining ungrounded loops. Note that, by opposition to what is suggested by the drawings, there can remain several new MLG.

```
procedure Relabel( G: graph) =                   /* Lokhorst */
for all S∈FindSCC(G, MSG ) by order
do   while (∃n∈S; unlabelled( n ) )
     do
       for all n∈S do LabelAndPropagate( n, S ) ;
       LabelUnGroundedLoops( S ) ;
       with n∈S such that unlabelled( n )        /* Choice point 1 */
       do Partition( n, S, OUT ) ;
     done
done
```

Thus, this algorithm differs in two ways from Goodwin's: first, the labelling is not carried out in only one step inside the MLG and its `Relabel` part is iterated until termination; second, the `Partition` algorithm only looks for the minimal grounding and thus tries to ground the chosen node with the first antecedent which allows to do it. the reason why this works is not obvious and the best argument are in the proof of Lemma 23.

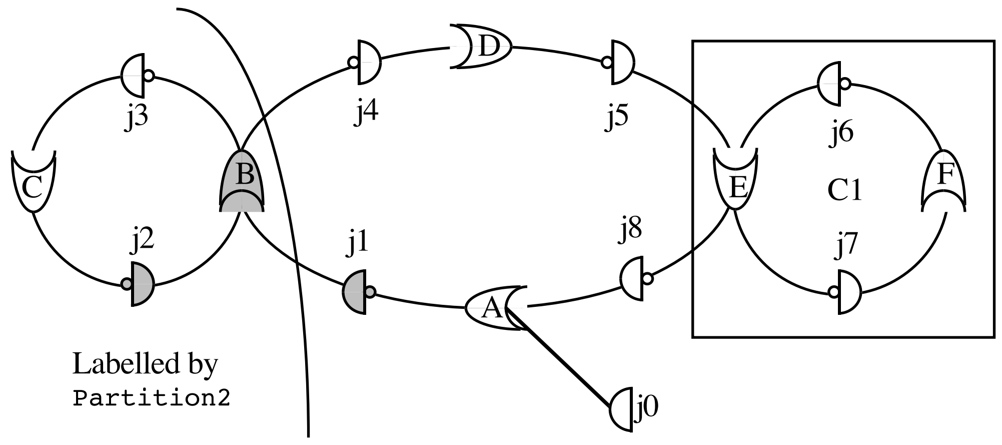


Figure 17. After choosing to label *B* as OUT, the `Partition` algorithm labels *C* as IN and stops. The subsequent call to `LabelAndPropagate` labels *D* as IN and stops. then, a new call to `LabelUnGroundedLoops` is run on *C1* and leaves it as one MLG. `Partition` can then be run on it.

```
procedure Partition( n: node, S: SubGraph, l: label ) =
if n∈S and unlabelled( n )
then
  if whichLabel( n ) = IN
  then
    label[n] <- IN ;
    with <I O>:n∈J                              /* Choice point 2 */
    do   for all n'∈I do Partition( n', S, IN) ;
         if ¬valid( j ) then signal('odd-cycle') endif
    done ;
  else
    label[n] <- OUT ;
    for each j=<I O>:n∈J
    do                                           /* Choice point 2 */
      TP <- I∪O ;
      while ¬invalid( j ) & TP≠{}
      do   if First(TP)∈I
           then Partition( First(TP), S, OUT) ;
           else Partition( First(TP), S, IN) ;
           endif ;
           TP <- Rest( TP ) ;
      done ;
      if ¬invalid( j ) the signal('odd-cycle') endif
    done
  endif
endif

function whichLabel( n: node, l: label ) =
if ∀<I O>:n∈J, invalid( j )
then <- OUT
else if ∃<I O>:n∈J such that valid( j ) then <- IN else <- l endif
endif
```

The `Partition` algorithm tries to find a grounded support for attributing label `l` to node `n`. It uses `whichLabel` which checks that the node can indeed have the label. If it is impossible, the algorithm terminates by giving the other label to `n`. In the opposite case, the algorithm looks for the support by trying each justification of a node to be `IN` or each antecedent of a justification to be invalid. It stops at the first which gives the expected result. The algorithm reports an odd cycle only when it is not possible to find the expected labelling.

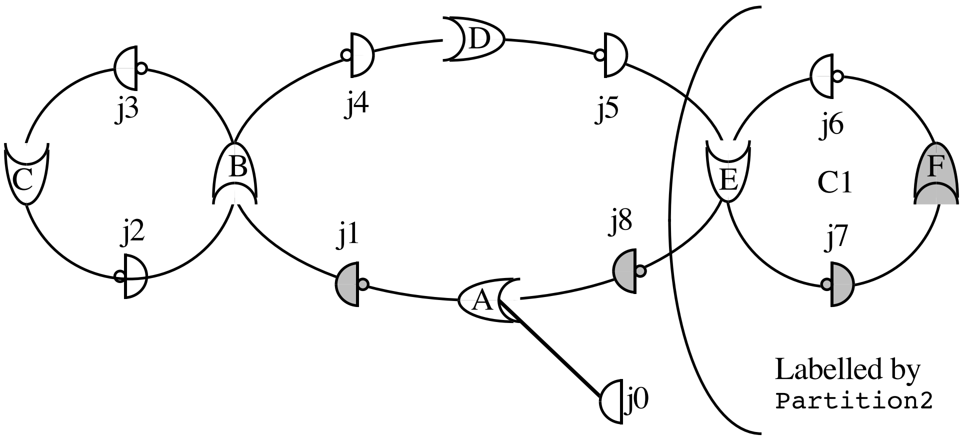


Figure 18. After choosing to label *F* as OUT, the `Partition` algorithm labels *E* as IN and stops. The subsequent call to `LabelAndPropagate` labels *j8* as invalid (that which does not matter) and stops. Then, a new call to `LabelUnGroundedLoops` can be carried out on the remainder and let it as one MLG. The `Partition` is then run on it and can choose to find the labelling of figure 13 for instance by labelling *B* as OUT.

Theorem 22 (termination of `Relabel`). `Relabel` terminates.

proof. Each time the remaining triangle is strictly smallest. Unlike the geometric metaphor, the space of nodes to label is finite and non dense. The process thus stops. ◊

Lemma 23. If there exist a labelling, `Relabel` finds one.

proof. This is the more difficult result about this algorithm. The main problem is to see why the algorithm is right when it notifies an odd cycle. This is due to the fact that odd cycles are signalled only if there is no possibility for labelling the graph. In such a case, the odd cycle exists because the actual labelling is due to the choice (1) otherwise the graph would have been labelled by previous call to `LabelAndPropagate`. If it were possible to backwardly label the current node IN (because it is impossible to invalidate any of its justification) and OUT because it is required by the actual backward search. This means that it is possible to reach the current node, from the node chosen in (1), by both odd and even paths. But lemma 6 tells that this is impossible without odd cycle. ◊

Theorem 24. (correctness of `Relabel`). `Relabel` finds a weakly grounded labelling of the graphs it labels.

proof. Just like the other algorithms, it does not apply too much constraint: it ensures grounding. It also ensures completeness by calling `LabelAndPropagate` (and all the labelling established by `Partition` is grounded). ◊

An algorithm which is potentially complete is an algorithm which can potentially find each one of the weakly grounded labellings of a considered graph. Potential completeness is practically important because it means that a particular application is enabled, through control on choice points, to determine which labelling the RMS achieves. Since non determinism only consists of a choice between alternative nodes of a MLG, it is sufficient to prove that these initial unlabelled subSCC are all that which can generate multiple labellings: the multiple labelling generators.

Corollary 25 (potential completeness of `Relabel`): `Relabel` can possibly find every weakly grounded labelling depending on the choices it does.

proof. First, any labelling of an MLG has an OUT node [lemma 21]. So, given a labelling *L* of the graph, it has a node labelled OUT. Let choose that node for initiating the call to `Partition`. The constraints which are propagated by the `Partition` algorithm are minimal. They apply to the labelling *L*. It is thus possible to choose the nodes and justifications which ensures groundedness in the same way as it is in *L*. Since the other procedures (`LabelAndPropagate` and `LabelUnGroundedLoops`) do a job which must also be accounted by *L*. The procedure as a whole can find the labelling *L*. ◊

At least, here is an algorithm which shares all the properties which are required. This algorithm is efficient (a bit less than Goodwin's which just uses one call to `Partition` on each MLG), correct and potentially complete. The behaviour of the algorithm, like that of Goodwin, is surprising because it labels the graph by alternating forward and backward constraint propagation. The key notion is the search for grounding when it can only be weak. However, labelling backwardly is surprising because the interpretation and production of justifications is achieved forwardly. This reveals one of the difficulties of non monotonic reasoning which does not progress from certitude to new certitude but must rather ensure that its assumptions can find some, at least weak, grounding[3].

The remaining sections are devoted to the search for a forward processing algorithm. It shows that the forward processing of re-labelling cannot be achieved deterministically.

4.4. Forward constraint propagation

A second algorithm only respects the closedness and groundedness constraints implied by the choice of one node to be OUT. As a consequence, this new algorithm labels the nodes only by propagating forwardly. It is not anymore able to label the whole MLG in one step. It thus re-iterate the SCC decomposition and the labelling of stratified SCC on the remaining unlabelled part of the MLG.

The algorithm proposed here is very similar to that of Goodwin, in particular, it covers the graph in the SCC order and does not examine a SCC twice.

[3] Not to speak of the will to preserve the set of assumptions from inconsistencies which is out of the scope here.

```
procedure Relabel( G: Graph ) =                    /* forward */
for all S∈FindSCC( G, CSG ) by order
do
  if stratifiedp(S)
  then
    if ∃J∈incidental-just(S) such that validep(j)
    then LabelStratified( S, IN );
    else LabelStratified( S, OUT );
    endif
  else
    ChooseLabel0( UnLabelableSubGraphs( S ) );
  endif
done
```

In order to label MLG, a non deterministic algorithm is provided. The new procedure also takes into account the fact that, after triggering `UnLabelableSubGraphs`, each node remaining to be labelled can, in some weakly grounded labelling, be labelled as OUT. This procedure chooses a node in a MLG to be labelled OUT and then uses the `UnConstrainedSubgraphs` procedure in order to propagate constraints and run `FindSCC` on the remaining unconstrained graph. It is called on a particular MLG *S* by:

`ChooseLabel0( UnLabelableSubGraphs( S ) ).`

It is noteworthy that `ChooseLabel` is able to label the initial stratified even subSCC.

```
procedure ChooseLabel0( S : SubGraphSets ) =
if S ≠ {} then
  with n in First( S )                         /* Choice point */
  do
    Label[n] <- OUT ;
    ChooseLabel0( UnLabelableSubGraphs( First(S)\{n} ) );
  done
  ChooseLabel0( Rest( S ) ) ;
endif
```

The processing of the algorithm, on the example of figure 2, can be seen on figure 19 and 20.

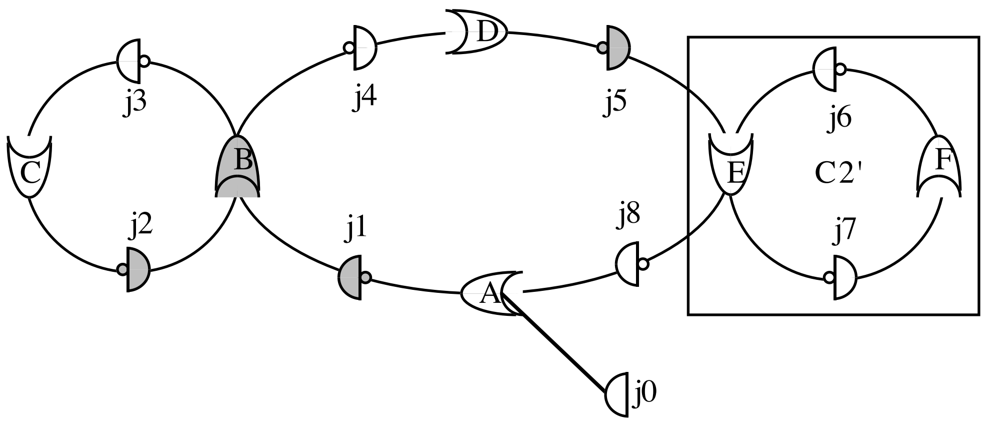


Figure 19. After choosing to label *B* as OUT, subSCC *C0’* and *C1’* are labelled. *C2’* remains unlabelled; it is thus, in turn, a MLG. Another choice has to be made.

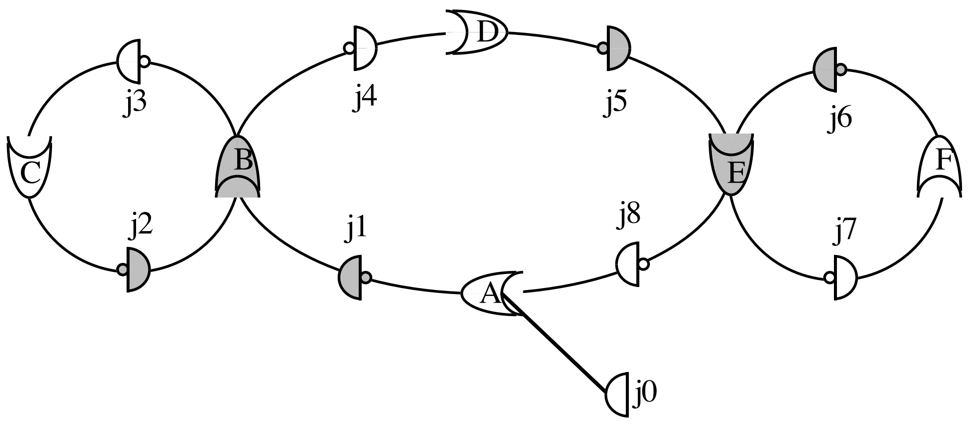


Figure 20. After choosing to label *E* as OUT, the whole MLG is labelled. Note that if in the previous choice (figure 19), it had been decided to label *C* OUT, then the whole MLG would have been labelled with only one choice.

The choice for the node to be labelled OUT does not only label a node but a set of dependent nodes. It is noteworthy that the algorithm has to choose twice: first for the first MLG (they are only partially ordered), second, for the node `n` to be labelled OUT. In fact, only the second choice is a source of non determinism of the labelling. The ordering of subSCC warrants that a subSCC is not taken into account before it becomes initial.

Some properties of that procedure can now be proven.

Lemma 26. The labelling process needs at least as many choices as the number of initial subSCC (MLG).

proof. This proposition just says that initial subSCC can only be labelled by choosing. This is obviously true since the labelling of an initial SCC cannot lead to constrain another initial SCC since they are fully constrained. ◊

Theorem 27 (termination of `ChooseLabel0`). `ChooseLabel0` terminates.

proof. `ChooseLabel0` stops when every node is labelled and otherwise chooses one and labels it before calling `UnLabelableSubGraphs`. `UnLabelableSubGraphs` terminates (by opposition to Doyle's initial algorithm, this one terminates more obviously) since at each iteration it labels an unlabelled node or stops and there is a finite number of nodes it can label. Moreover, at each call of `ChooseLabel0`, there is one more node which is labelled: the chosen one. Since the number of nodes is finite, the algorithm stops due to a finite number of iterations. ◊

The proof of theorem 27 gives the following corollary:

Corollary 28. `ChooseLabel0` stops with all nodes labelled.

The potential completeness of the algorithm can now be established. This algorithm is able to find each possible labelling of the graph. As a consequence, alternate labellings are generated at the choice points of this algorithm which, in turn, only rely on the choices among the MLG which are initial unlabelable subSCC.

In fact, the procedure above is the first incremental algorithm which labels the nodes exclusively in a *forward* fashion. It does it by determining dynamically the order in which MLG can be examined. The labelling process leads to the sequence given in figure 5 to 8, 19 and 20 (whose labelling is different from those of figure 13). So, the tree of possible choices in the labelling process is that of figure 21. This tree contains the three possible labellings of the graph of figure 6.

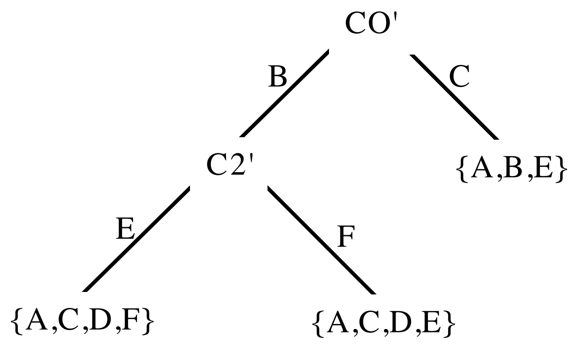


Figure 21. The different choices which can be made by `ChooseLabel0`. The nodes on the edges of the tree are those chosen to be OUT. The nodes are initial subSCC and the leaves are the set of nodes labelled IN.

Theorem 29. (potential completeness of `ChooseLabel0`). `ChooseLabel0` can possibly find every weakly grounded labelling depending on the choices it does.

proof. It has to be shown that this algorithm is able to find any weakly grounded labelling *L*. Let consider a partially labelled graph and assume that this graph is labelled as in *L* (otherwise, this reasoning as to be done during the former labelling process). It is clear [corollary 15] that every node which is labelled by `UnLabelableSubGraphs` is constrained (it cannot be labelled in any other way) and this is also true for *L*.

Now, let consider the remaining nodes in a MLG. Any such node (by virtue of corollary 20), can be labelled OUT. Let choose as the node to label a node labelled OUT in *L* (if it does not exist such a node then *L* violates some property). Since, `ChooseLabel0` does not over-constrain, the constraint it propagates are satisfied in *L* and thus the labelling during propagation is the same as that of *L*. Now, two cases are possible after backward propagation:

- The whole SCC is labelled. Then it is just like in *L* and the algorithm can consider consequent SCC and respects the initial condition.
- There is no more constraints to be satisfied, then it can be proven, like in the proof of lemma 21 that if all the remaining nodes to label are IN, then the labelling *L* is not weakly grounded (since the remaining nodes are decomposed into MLG by `UnLabelableSubGraphs`, they would be all IN on their own). Therefore there is an

unlabelled node which is labelled OUT in L, and which can be chosen to be labelled OUT. The forward propagation takes place here again in the same conditions as above.

It is noteworthy that no violation can occur during this process since the choices are exactly that of L which is a solution. As a consequence, the resulting labelling must be L. ◊

This process can be transformed into a decision procedure for the problem of testing weak groundedness of a given labelling.

Annex A shows that the complexity of this last algorithm is $O(|\mathcal{N}|^2)$, the same as that of Goodwin's algorithm. Unfortunately this algorithm is not correct. Like Goodwin's was over-constraining, this one is under-constraining. In order to understand it, consider the graph of figure 22 and its labelling.

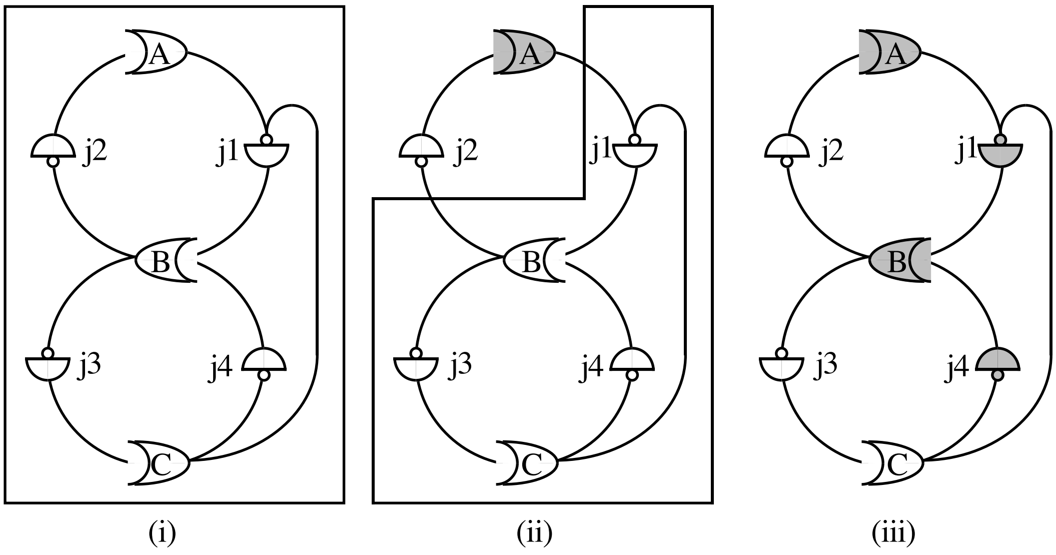


Figure 22. The graph (i) is an unconstrained SCC. So, the `ChooseLabel` algorithm can choose A to be labelled OUT. As a consequence, there is a subSCC $\{B, C\}$ in (ii). If B is then chosen to be OUT, the resulting labelling is (iii). The problem is that the labelling of (iii) is not closed!

This example shows that it is practically impossible to label the graph in a pure forward fashion, since the choices made at a moment lead to constraint the graph backwardly for closedness purpose. The choice of A as first OUT is certainly possible since the graph of figure 23 has an admissible labelling with A OUT.

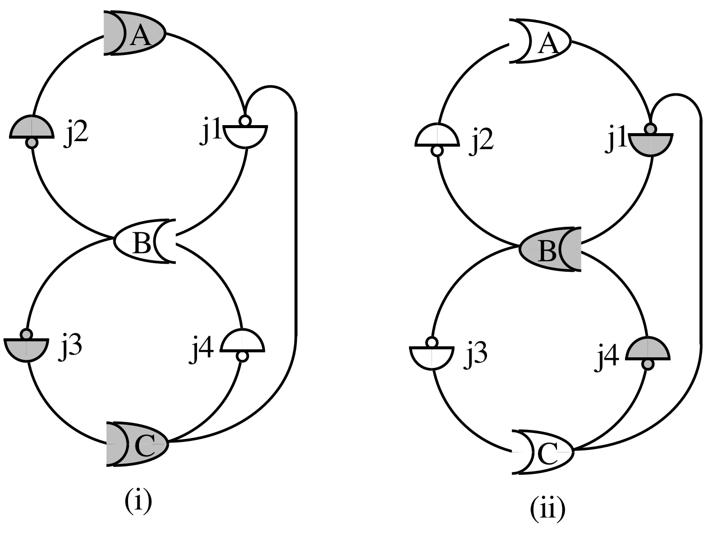


Figure 23. The two weakly grounded algorithms of the graph of figure 22.

4.5. Other algorithms

David Russinoff [Russinoff 85] designed a complete algorithm for a broader problem than the one addressed here: that of finding a weakly grounded labelling to any graph (without taking hypothesis 1 into account). The algorithm works in a strictly forward fashion recording each choice made and backtracks whenever it cannot label anymore without violating the weak groundedness requirement. Such an algorithm is, obviously, exponential.

Junker and Konolige's algorithm [Junker& 90], for its own, is quite different since it ensures that it finds every possible weakly grounded labelling. However, it is a non-incremental algorithm which does not maintain the set of labellings but deliver them as soon as it finds them. It propagates forwardly but does not consider MLG. It adopts the idea that cycles are "what remains unlabelled when classic constraint propagation labelled every other nodes" and then alternatively chooses every such node to be OUT. As a result, this algorithm generates several times the same labelling and this might cause important overhead. It certainly can be adapted in order to consider MLG and, so, to generate a minimal number of labellings.

Such an algorithm would consist of the algorithm `ChooseLabel0` used for generating each possible labelling and enhanced with a test for each newly labelled node in order to ensure that it does not support the INness of a previously OUT labelled node (a chosen one). As a matter of fact, although `ChooseLabel0` is not correct in the deterministic case, it is noteworthy that, in the perspective of an algorithm which looks for all weakly grounded labellings (or, if the last algorithm were non deterministic), Goodwin's would miss some labellings while `ChooseLabel0` would have too many ones. Anyway, in the deterministic case, Goodwin is better since it always finds a weakly grounded labelling.

So, in order to ensure closedness, it is possible to consider `ChooseLabel0` plus a test of closedness during label propagation. If that closedness is not satisfied, then, the choice is withdrawn and the system must choose another node. If there is no other choice available, then the program backtracks to the last call of `ChooseLabel` until it finds a new available choice. This closedness test of `LabelAndPropagate` is the basis for the `ChooseLabel1` algorithm.

```
procedure ChooseLabel1( S: SubGraphSets )=
if S≠{}
then
  ChooseAndContinue( First( S ) );
  ChooseLabel1( Rest( S ) );
endif

procedure ChooseAndContinue( S: SubGraph ) =
with n∈S                                        /* Choice point */
do
  push( CP, < S\{n}, G > );
  label[n] <- OUT;
  ChooseLabel1( UnLabelableSubGraphs( First(S)\{n} ));
done

procedure LabelAndPropagate( n: node ) =        /* version 2 */
if labelled(n)
then
  if Label[n]=OUT & ∃j∈L-Just[n]; valid( j )
  then Backtrack()
  endif
else
  if constrained( n )
  then
    Label( n );
    for all n'∈Csq[n] do LabelAndPropagate( n' );
  endif
endif

procedure Backtrack() =
begin
  < S, G > <- pop( CP );
  if S ≠ {}
  then ChooseAndContinue( S );
  else Backtrack();
  endif
end
```

Despite the fact that this procedure can be supposed correct and potentially complete, it is not efficient at all. This is the reason why a better algorithm can be looked for.

5. Conclusion

Two “topological” criteria have been defined in order to isolate multiple labelling generators. The first one is rather classic, it is the partition of the dependency graph in SCC. The originality stems from the partitioning of the complete support graph and not of the usual minimal support graph or potential support graph. With this criterion, the result obtained is that MLG are circumscribed inside AE-SCC. A corollary of this is that other SCC can be labelled in only one step. This, conjugated with previous results due to James Goodwin, allows to label very quickly an important part of the graph.

The second criterion relies on the current labelling process and partitions each AE-SCC in unlabelable subSCC. It has been shown that the initial unlabelable subSCC are the MLG and can be labelled in several ways.

Are these two criteria necessary? The whole graph could have been considered under the point of view of current unlabelled support graph which really accounts for MLG. But, in such a case, this does not lead to an incremental algorithm though the old labelling is not considered any more. The advantages of the first criterion are manifold:

- The SCC of the complete support graph are easy to obtain.
- They allow to circumscribe the part of the graph in which a propagation is needed.
- The notion of complete support (it was also the case for minimal support [Euzenat 88]) enables to stop the propagation if it is no further needed.
- The complete support graph SCC allows to quickly label unitary and stratified SCC.

So, both criteria are important as soon as the implementation is the aim to achieve.

At last this work did only systematically apply common-sense knowledge to the labelling process of reason maintenance. Topological criteria on the different graphs manipulated by the RMS have been provided with a constructive characterisation. This leads to some proposed algorithms but can also explain some previous works. The text is guided by the search for a trade-off between correctness, potential completeness and efficiency. Hence, the algorithms are more intelligible if not more efficient.

Acknowledgement

I am indebted to Henk Lokhorst who carefully read a previous version of this report and found a bright counter-example to the successor of `ChooseLabel1` which, consequently, does not appear anymore.

References

[Aho& 74]
Alfred Aho, John Hopcroft, Jeffrey Ullman
The design and analysis of computer algorithms
Addison-Wesley, Reading (MA US), (rep. Data structures and algorithms, 1983), 1974

[Charniak& 80]
Eugene Charniak, Christopher Riesbeck, Drew Mac Dermott
Artificial intelligence programming
Lawrence Erlbaum, Hillsdale (NJ US), 1980

[Doyle 79]
Jon Doyle
A truth maintenance system
Artificial intelligence 12(3):231-272, 1979

[Elkan 90]
Charles Elkan
A rational reconstruction of nonmonotonic truth maintenance systems
Artificial intelligence 43(2):219-234, 1990

[Euzenat 88]
Jérôme Euzenat
Un nouvel algorithme de maintenance de la vérité (in french)
Technical report, Cognitech, Paris (FR), 1988

[Euzenat 90]
Jérôme Euzenat
Un système de maintenance de la vérité à propagation de contextes (in french)
PhD Thesis, université Joseph-Fourier, Grenoble (FR), 1990

[Euzenat 91]
Jérôme Euzenat
Contexts for nonmonotonic RMS
Proc. 12th IJCAI, Sydney (AU), pp300-305, 1991

[Fujiwara& 89]
Yasushi Fujiwara, Shinichi Honiden
Relating the TMS to autoepistemic logic
Proc. 11th IJCAI, Detroit (MI US), pp1199-1205, 1989

[Goodwin 82]
James Goodwin
An improved algorithm for non-monotonic dependency net update
Research report MAT-R-82-23, Linköping university (SE), 1982

[Goodwin 87]
James Goodwin
A theory and system for non monotonic reasoning
Linköping studies in science and technology 165, 1987

[Junker& 90]
Ulrich Junker, Kurt Konolige
Computing the extensions of autoepistemic and default logics with a truth maintenance system
Proc. 8th AAAI, Boston (MA US), pp278-285, 1990

[Junker 91]
Ulrich Junker
Prioritized defaults: implementation by TMS and application to diagnosis
Proc. 12th IJCAI, Sydney (AU), pp310-315, 1991

[Lokhorst 92]
Henk Lokhorst
New algorithms for justification-based truth maintenance
Research report INF/SCR-92-25, Utrecht university, Utrecht (NL), 1992

[MacDermott 91]
Drew Mac Dermott
A general framework for reason maintenance
Artificial intelligence 50(3):289-329, 1991

[Quintero 89]
Jose Alejandro Quintero-Garcia
Parallélisation de la maintenance de la vérité tirant parti des composantes fortement connexes (in french)
Master report, Joseph-Fourier university/INPG, Grenoble (FR), 1989

[Petrie 87]
Charles Petrie
Revised dependency directed backtracking for default reasoning
Proc. 6th AAAI, Seatle (WA US), pp167-172, 1987

[Reinfrank& 88]
Michael Reinfrank, Oskar Dressler
On the relation between truth maintenance and non-monotonic logics
Research report INF2-ARM11-88, Siemens, München (DE), 1988

[Reinfrank& 89]
Michael Reinfrank, Oskar Dressler, Gerd Brewka
On the relation between truth maintenance and autoepistemic logics
Proc. 11th IJCAI, Detroit (MI US), pp1206-1212, 1989

[Reinfrank 90]
Michael Reinfrank
Fundamentals and logical foundations of truth maintenance
Linköping studies in science and technology 221, 1989

[Russinoff 85]
David Russinoff
An algorithm for truth maintenance
Research report AI-062-85, MCC, Austin (TX US), 1985

[Úrbanski 87]
Andrej Úrbanski
Truth maintenance systems and their algorithms
Proc. 2nd Cognitiva, Paris (FR), pp262-266, 1987

Annex A: A brief complexity analysis of `ChooseLabel0`

Before analysing the algorithm, two notices about the used sub-algorithms:

1) The `FindSCC` procedure, is a slight variation of the SCC decomposition procedure given in [Aho& 74]. It is able, not only to return the set of SCC respecting their partial order, but also to provide the incident justifications and the parity of the SCC. The modifications take place in the second (backward) depth first search in which the incident justifications are recorded when encountered and the parity indicator changed when going through an OUT-list. As a consequence this algorithm still proceeds in O(n) where n is the number of nodes and justifications.

2) In its first complexity analysis [Goodwin 82], James Goodwin considered that checking a justification for validity could be done in constant time. In his thesis [Goodwin 87], he came back on that proposition and considered that, if no limit is imposed on the number of antecedents of a justification, this must be done in linear time with regard to the number of antecedents, so, at worst, O($|\mathcal{N}|$). It will be considered that it is possible to improve validity checking time complexity, by sacrificing some space in order to maintain, in the justification structure, an indicator of the number of “not yet accorded” antecedents. Thus, the time for checking and updating this indicator is constant.

`UnLabelableSubGraphs` if in O(n^2) [Goodwin 82]. But this complexity assume that the n nodes are labelled (one after another). Thus, in such a case, `ChooseLabel0` does not need any more processing and its complexity is O($|\mathcal{N}|^2$). This is a first lower bound for the complexity of `ChooseLabel0` which needs to be checked because the `UnLabelableSubGraphs` call is itself included in a loop. In fact, the worst case of `ChooseLabel0`’s loop is characterised by two properties:

P1) At each iteration of the algorithm, only one node is labelled.

P2) At each iteration of the algorithm, everything is in only one MLG.

Let prove that this is indeed the worst case.

Assume ¬P1, then the propagation algorithm takes linear time in term of the number of labelled nodes, and there is less iterations of the whole algorithm. As a whole, it takes less time.

Assume ¬P2:

(a) SE-SCC (and U-SCC) are labelled in linear time. Since, these SCC are recognised during the decomposition phase at no cost, this takes less time than labelling an AE-SCC.

(b) multiplying SCC obviously liberates the “divide and conquer” action. This can be proved by induction on the size of the subSCC. The required time for labelling a SCC of size 2 is $t_2=2^2=4$ while that for labelling two SCC of size 1 is $t_1+t_1=1+1=2$. Now, assume an AE-

SCC with n elements which is divided into several subSCC $S_1,\ldots S_k$ with size $g_1,\ldots g_k$, they can be labelled in less than $t_n=n^2$ since they are labelled in:

$$\sum_{i=1}^{k} t_i = \sum_{i=1}^{k} g_i^2 < n^2 \text{ (because} \sum_{i=1}^{k} g_i = n).$$

thus, by induction, the worst case is when there is only one big AE-SCC.

The case P2b also applies for deducing that, labelling several nodes by `UnLabelableSubGraphs`, would enable to divide and conquer again.

`ChooseLabel0` alternates the call of SCC decomposition and propagation phases. In this worst case, it is called $|\mathcal{N}|$ times. Each time, it has to proceed the SCC decomposition (`UnLabelableSubGraphs` which, in that case (no labelling) is linear in term of the number of remaining nodes and justifications) and a constraint propagation which is also linear in term of the remaining nodes. As a consequence, `ChooseLabel0` is in $O(|\mathcal{N}|^2)$.

Annex B: Index of algorithms

Algorithms, their occurrence in other algorithms and the proof of properties are indexed here. Only one occurrence is referred.