THANK YOU
FOR YOUR ORDER.

D:IBM or IBM subsidiary
Philippe Laborie
rue de Verdun
Gentilly, 94253
France

IBM KnowledgeGate Operations
TB-17C, M/D T-08 One North Castle Drive
Armonk, NY 10504
United States
Phone or
Email itorders@us.ibm.com

ORDER INFORMATION

INFO #:	19799214
SHIP VIA:	Ariel (eMail TIFF)
ORDER #:	
CUSTOMER #:	123283 / 2285751
BILLING REF:	
CCD:	4590
ORDERED:	01/25/2011
NEED BY:	

ARTICLE INFORMATION

ACCESSION #:	02545330
PUBLICATION:	ANNALS OF OPERATIONS RESEARCH
DETAILS:	180(1): 2010
AUTHOR:	
TITLE:	Discovering implied constraints in precedence graphs with alternatives

CUSTOMER INFORMATION

SPECIAL INSTRUCTIONS:	
COMPANY:	D:IBM or IBM subsidiary
PATRON:	Philippe Laborie
PHONE:	0139083567
FAX:	
EMAIL:	laborie@fr.ibm.com
ARIEL:	

The document you requested from KnowledgeGate is being sent to you via Ariel electronic delivery. Ariel sends a .tif file attachment to your email address. Opening the attachment requires a multipage TIFF viewer included with most operating systems (or download at http://www.alternatiff.com).

To access your document, click on the right arrow button on the tools bar. Use the right and left arrow buttons to navigate through your document.

Only the attachment called "Document.TIF" holds your article. The other two hold information about delivery. There is no need to open them unless you need more information about your delivery.

This document is protected by U.S. and International copyright laws. No additional

REGULAR

Ann Oper Res (2010) 180: 233–263
DOI 10.1007/s10479-008-0492-1

Discovering implied constraints in precedence graphs with alternatives

Roman Barták · Ondřej Čepek · Pavel Surynek

Published online: 13 December 2008

Abstract During automated problem solving it may happen that some knowledge that is known at the user level is lost in the formal model. As this knowledge might be important for efficient problem solving, it seems useful to re-discover it in order to improve the efficiency of the solving procedure. This paper compares three methods for discovering certain implied constraints in the constraint models describing manufacturing (and other) processes with serial, parallel, and alternative operations. In particular, we focus on identifying equivalent nodes in the precedence graph with parallel and alternative branches. Equivalent nodes correspond to operations that either must be all simultaneously present or none of them can be present in the schedule. Such information is frequently known at the user level, but it is lost in the formal model. The paper shows that identifying equivalent nodes is an NP-hard problem in general, but it is tractable if the graph has a nested structure. As the nested structure is typical for real-life processes and workflows, we use the nested graphs to experimentally compare the proposed methods.

Keywords Constraint satisfaction · Modelling · Implied constraints · Temporal networks

Scheduling problems deal with allocation of known activities to scarce resources and time. The activities in scheduling problems are frequently connected via the precedence constraints into processes describing the flow of products. These precedence constraints can be

R. Barták (✉) · O. Čepek · P. Surynek
Charles University, Faculty of Mathematics and Physics, Malostranské nám. 2/25, 118 00 Prague 1, Czech Republic
e-mail: roman.bartak@mff.cuni.cz

O. Čepek
e-mail: ondrej.cepek@mff.cuni.cz

P. Surynek
e-mail: pavel.surynek@mff.cuni.cz

O. Čepek
Institute of Finance and Administration, Estonská 500, 101 00 Prague 10, Czech Republic

modelled using a *precedence graph* which is a directed acyclic graph where the nodes correspond to the activities and the arcs describe the precedence relations. Real-life problems are usually more complex than existing theoretical models such as job-shop or flow-shop and, for example, they also require a selection among alternative process routes or alternative resources in complex manufacturing enterprises. These decisions are usually done in advance at a planning stage, so the traditional scheduling deals primarily with the sequencing of the pre-selected activities on resources. To increase flexibility of scheduling systems, it is important to do the decisions on alternatives at the same level as sequencing decisions because both types of decisions are clearly interrelated (for example, a fully occupied resource may force an alternative process route that does not use that resource). Reasoning on alternative processes introduces logical constraints into the model—these constraints describe relations such as exclusive alternatives (at most one alternative can be used). This paper deals with these logical constraints; in particular, we study methods inferring additional logical constraints that increase efficiency of problem solving. We assume that the studied methods are used together with the traditional resource constraints in the context of constraint-based scheduling.

To model alternative process routes we use an extension of precedence graphs called a *P/A graph* (Barták et al. 2007) that models both parallel and alternative routes. To model the selection of alternatives, a validity variable is assigned to each node in the precedence graph. This validity variable indicates whether the node/activity is selected to be or not to be in the final solution plan. The decision about the validity/invalidity of the node is done by the solver and it can be influenced by other constraints, such as resource or temporal constraints. The precedence graph is also augmented by the description of splitting and joining operations that implicitly define logical dependencies between the nodes in the network. The dependency relations specify which nodes must/cannot be valid in relation to the validity status of other nodes. In some nodes the manufacturing process can split into two or more parallel sub-processes that can join back to a single process. These sub-processes are part of the whole process and they can run in parallel. For example, a wood board is cut in parts that are processed in parallel and then assembled together to a final product. This is called *parallel branching*. Another form of branching is *alternative branching* when the process also splits in sub-processes, but these sub-processes are treated as exclusive alternatives, so exactly one of them is used. For example, to cut a wood board we can use a fully automated cutting machine or the cutting can be done manually in several steps. The alternative sub-processes can also join back to a single process. Figure 1 gives an example of a P/A graph with alternative and parallel (marked by a semicircle) branching, where one process is selected.

The logic behind the above described precedence graph with alternative routes—a *P/A graph*—can be easily described as a constraint satisfaction problem. Each validity variable has a domain $\{0, 1\}$ specifying possible values that can be assigned to the variable: 0 means that the node is invalid, 1 means that the node is valid. Each alternative branching from a to a set of b_i's is described using a constraint $a = \sum_i b_i$, where a and b_i's are the validity variables of corresponding nodes in the branching. Notice that the arithmetic relation naturally describes the logical dependency between the nodes: a is valid if and only if exactly one of nodes b_i is valid. Finally, each parallel branching from a to a set of b_i's is described using the set of constraints $\forall i\, a = b_i$. Again, these constraints describe the logic behind parallel branching which says that all nodes in the branching are valid or neither node is valid (all nodes are invalid). Clearly, there exists an assignment of values to variables satisfying all the constraints if and if only there exists a selection of nodes of the P/A graph satisfying the branching relations. We call the above described model a *basic constraint model* for the P/A graph.

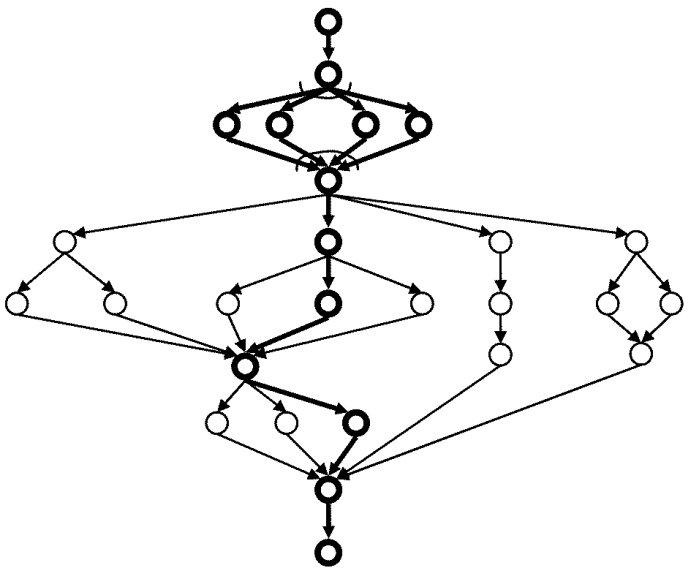


Fig. 1 Graph of parallel (*top*) and alternative (*bottom*) sub-processes with a selected process

Constraint satisfaction problems are usually solved using a combination of depth-first search and an inference technique known as (generalized) arc consistency (Dechter 2003). Each time a decision is made during search, for example a value is assigned to a variable, this decision is propagated to other variables by making the problem arc consistent. This propagation goes through individual constraints and removes values from variables' domains that do not satisfy the constraint. The problem of the above constraint model is that it does not propagate well. For example, if we make valid the bottom node in Fig. 1, the constraint model is not able to deduce (via arc consistency) that the top node must be valid as well. This weak propagation may be explained by a disjunctive character of constraints modeling alternative branching. Assume the following constraint satisfaction problem: $A = B + C$, $D = B + C$, where domains of all variables are $\{0, 1\}$. If we assign value 1 to A then no value is pruned from the domains of B and C simply because any value can still satisfy the constraint. Hence no information is propagated to the other constraint and the domain of D is not changed. Clearly, value 0 cannot be assigned to D in any solution. This weak filtering can be bridged by adding so called *implied constraints* that propagate the information directly between the variables. In the above example we can add an implied equality constraint $A = D$ which is similar to deducing that the validity status of the top and bottom nodes in Fig. 1 is identical.

Many implied constraints are known to the user specifying the problem, but they are lost in the formal constraint model simply because the implied constraints are redundant—they are not necessary to define the problem. Nevertheless, the implied constraints may improve efficiency of problem solving by helping the local inference techniques such as arc consistency to filter out more inconsistencies and hence to prune the search space. Therefore it is important to rediscover such implied constraints.

In this paper, we deal with rediscovering the implied constraints in P/A graphs, in particular, with finding pairs of nodes in the P/A graph whose validity variables must be assigned the same value in all solutions—we call such nodes *logically equivalent*. In the paper, we compare three methods for finding equivalent nodes. The first method was originally proposed for general P/A graphs in Barták et al. (2007) and it is based on applying graph-modification rules to the graph. The second method (Barták and Čepek 2007) exploits a nested structure of graphs that is typical for real-life P/A graphs. The last method (Barták

2007) has been proposed for finding implied logical constraints in any CSP using singleton arc consistency, but it was also motivated by the problem of finding equivalent nodes in P/A graphs.

The paper is organized as follows. We first introduce the necessary notions from the area of constraint satisfaction and we survey the related works. Then, we formally define the P/A graph and its feasible assignments of validity variables. We then show that the problem of deciding the existence of such a feasible assignment is NP-complete, and we also prove that finding all logically equivalent nodes in a P/A graph is an NP-hard problem. The main part of the paper will describe the three above-mentioned methods for identifying some equivalence classes in P/A graphs. We conclude the paper by experimental comparison of these three methods using nested P/A graphs that are the most typical representatives of P/A graphs in real-life problems.

1 Constraint satisfaction at a glance

A *constraint satisfaction problem* (CSP) P is a triple (X, D, C), where X is a finite set of decision variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for variable x_i (the domain), and C is a finite set of constraints (Dechter 2003). A constraint is a relation over a subset of variables that restricts possible combinations of values to be assigned to the variables. Formally, a constraint is a subset of the Cartesian product of the domains of the constrained variables. We call the variable Boolean if its domain consists of two values $\{0, 1\}$ (or similarly $\{false, true\}$). A *solution to a CSP* is a complete assignment of values to the variables such that the values are taken from respective domains and all the constraints are satisfied. We say that a constraint C is (*generalized*) *arc consistent* if for any value in the domain of any constrained variable, there exist values in the domains of the remaining constrained variables in such a way that the value tuple satisfies the constraint. This value tuple is called a support for the value. Note that the notion of arc consistency is usually used for binary constraints only, while generalized arc consistency is used for n-ary constraints. For simplicity reasons we will use the term arc consistency independently of constraint's arity. The CSP is *arc consistent* (AC) if all the constraints are arc consistent and no domain is empty. To make the problem arc consistent, it is enough to remove values that have no support (in some constraint) until only values with a support (in each constraint) remain in the domains. If any domain becomes empty then the problem has no solution. We say that a value a in the domain of some variable x_i is *singleton arc consistent* if the problem $\mathrm{P}|x_i = a$ can be made arc consistent, where $\mathrm{P}|x_i = a$ is a CSP derived from P by reducing the domain of variable x_i to $\{a\}$. The CSP is *singleton arc consistent* (SAC) if all values in variables' domains are singleton arc consistent. Again, the problem can be made SAC by removing all SAC inconsistent values from the domains. SAC is stronger than AC as it can remove more inconsistencies from the problem, but achieving SAC is also more time consuming than achieving AC which is the reason why SAC is rarely used. Figure 2 shows an example of a CSP and its AC and SAC forms. Nodes correspond to variables (they show variables' domains) and edges correspond to binary constraints between the variables.

Assume now the constraint satisfaction problem with Boolean variables A, B, C, and D and with constraints $A = B + C$ and $D = B + C$. This problem is both AC and SAC. Now assume that we assign value 1 to variable A. The problem remains AC but it is not SAC because value 0 cannot be assigned to variable D. This is an example of weak domain pruning in P/A graphs. If we now include constraint $A = D$ and make the extended problem arc consistent then value 0 is removed from the domain of D. Clearly, any assignment satisfying the original constraints also satisfies this added constraint. Hence we call this constraint

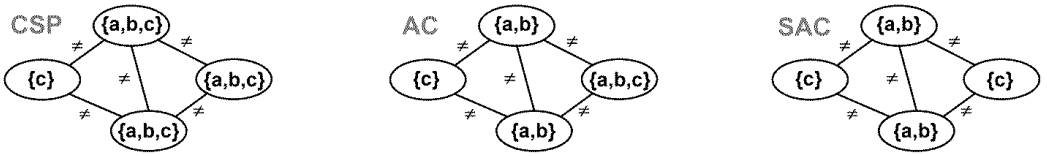


Fig. 2 A graph representation of a CSP, an arc consistent problem, and a singleton arc consistent problem (*from left to right*)

implied, because the constraint is logically implied by the original constraints (sometimes, these constraints are also called *redundant*). Our goal is to find such implied constraints that contribute to stronger domain filtering. As the above example shows, SAC could be one of the methods that can help us there.

2 Related works

There are two important areas of this paper that deserve a deeper study of related works. Those are: (1) modelling alternatives in scheduling problems and (2) automated generation of implied constraints.

An interest in modeling alternatives in planning and scheduling dates far back. As early as in 1960's efforts were made to integrate the decisions about alternative plans into the scheduling phase of project planning (Crowston and Thompson 1967). Proposed solution methods usually utilized heuristics or integer programming tools. Among the more recent papers (Focacci et al. 2000) describes a graph concept for modelling alternative processes, but it cannot be used for alternative routes because all activities must be present. The motivation for our proposal of P/A graphs is also very close to the work by Beck and Fox (2000) on modeling alternative activities in scheduling problems. They used so called PEX variable to describe a probability of existence of a node (activity) in the final solution and they introduced XorNodes and AndNodes to model branching. The PEX variables were used mainly for propagation of probability of existence (rather than logical reasoning) and for guiding the search procedure. The authors also attempted to define legal temporal networks which describe "valid" real-life networks. Our P/A graph (Barták et al. 2007) formalizes the idea of alternative nodes in precedence graphs. We abstracted from the type of node by focusing on the branching relations. Hence a node may be part of parallel input branching and alternative output branching at the same time (and vice versa) which is not possible in Beck and Fox's approach. Nevertheless, this is not a significant structural difference, because such combined node can be artificially split into two nodes in Beck and Fox's approach. The focus of our work is on logical reasoning about alternatives which was omitted in previous works. In Barták et al. (2007) we showed that logical reasoning, in particular solving the assignment problem, is NP-hard for general P/A graphs. This justifies the quest for valid networks where logical reasoning is easy. The legal networks in Beck and Fox (2000) are defined only informally and no algorithm for recognizing legal networks was given. We proposed nested P/A graphs (Barták and Čepek 2007) to describe the very same idea though there are some differences. First, we use a formal and constructive definition of nested networks. This definition is more general than legal networks, for example, the graph in Fig. 1 is illegal according to Beck and Fox (2000), but it is a nested graph. We believe that our approach is more transparent and more general in the specification of valid networks (Beck and Fox 2000). Moreover, we proved that logical reasoning is tractable in nested graphs and we provide an algorithm for recognizing nested graphs.

Nested P/A graphs (Barták and Čepek 2007) were proposed as a special version of P/A graphs by using the motivation from real-life manufacturing processes appearing by decomposition of meta-processes into operations. This is identical to obtaining temporal networks in HTN planning (hierarchical task networks; Erol et al. 1994) so it is not surprising that the structure of Nested TNA is identical to Temporal Planning Networks proposed in Kim et al. (2001) and also used in TAEMS formalism (Horling et al. 1999). TPN and TAEMS provide a language for specifying the nested networks and only the nested networks, hence they do not require any algorithm for recognizing nested networks. We adopted a more general approach for network description based on specifying the type of branching. Examples in Barták and Čepek (2007) show P/A graphs which cannot be expressed as TPNs. Hence we need a method for recognizing nested graphs among P/A graphs, which is given in this paper. Moreover, the recognition algorithm is a key to the formulation of our tractable constraint model. The authors of TPN and TAEMS never paid attention to complexity of selecting alternatives if some nodes must be present, in particular, selection of right alternative is influenced there by temporal and resource constraints only. However, choosing alternative operations to satisfy given goals is very common in practice. In paper (Barták and Čepek 2007) we showed that selection of alternatives with pre-selected nodes is tractable for nested graphs and the preliminary experiments in this paper confirmed that the basic model (that behaves similarly to TPN) is not always computationally efficient. Moreover, by using logical constraints, we put another dimension into the problem specification in addition to temporal and resource constraints.

Logical constraints were introduced to scheduling problems in MaScLib by ILOG (Nuijten et al. 2003) and Extented RCPSP (Kuster et al. 2007) where binary logical relation can be expressed between any two nodes (any pair of validity variables). Recall, that P/A graphs use n-ary logical relation only between the node and directly preceding (or succeeding) nodes which seems more restrictive. Though logical relations between distant nodes can be found in industrial practice (selection of one alternative influences selection of another alternative), we have found it less frequent in real-life problems and actually, the benchmarks provided with MaScLib do not use this feature. On the other hand, the binary logical constraints cannot express the alternative branching constraints which are much more frequent. Let us consider a ternary constraint $v_A = v_B + v_C$ given by an alternative branching from node A to nodes B and C. The satisfying assignments (models) of truth values to validity variables v_A, v_B, and v_C are $M_1 = (1, 0, 1)$, $M_2 = (1, 1, 0)$, and $M_3 = (0, 0, 0)$. Let us now assume by contradiction, that the same set of models for v_A, v_B, and v_C can be enforced by a set X of binary constraints (by a conjunction of all constraints in X). The set of all binary constraints involving variables $V(A)$ and $V(B)$ must have a model $v_A = 1$, $v_B = 0$ since otherwise, regardless of the remaining binary constraints in X, M_1 is not a model of X. Similarly, using M_2 we easily conclude that the set of all binary constraints involving variables v_A and v_C must have a model $v_A = 1$, $v_C = 0$. Finally, M_3 forces the set of all binary constraints involving variables v_B and v_C to have a model $v_B = 0$, $v_C = 0$. However, putting the above considerations together, it is obvious that $v_A = 1$, $v_B = 0$, $v_C = 0$ is a model of X. On the other hand, it is not a model of the original ternary constraint, which gives the desired contradiction.

There is a clear relation of our framework to temporal networks (Dechter et al. 1991) but no existing extension of temporal networks can handle alternative paths in the same generality as P/A graphs. For instance in Conditional Temporal Planning (Tsamardinos et al. 2003) the existence of a node in the network depends on a certain external condition. Though there is some similarity to the modelling of alternative processes/plans, a satisfaction of a

condition in CTP depends on external forces—Nature—rather than being an internal relation between the nodes. In our approach, the decision about the validity of a node is done internally based on logical relations between the nodes.

There is also some visual similarity of P/A graphs and AND/OR graphs which are widely used in Artificial Intelligence. However, there are two major differences between our approach and AND/OR graphs. First, the alternative branching corresponds more to XOR relation than to OR relation. Moreover, alternative branching is not used in the same way as OR relation in AND/OR graphs. We allow only two types of feasible assignment for alternative branching: either all nodes in the branching are invalid or the principal node is valid and exactly one branching node is valid (XOR relation would allow the principal node to be invalid if two or more branching nodes are valid). The second difference is even more significant—opposite to AND/OR graphs, where only fan-out branching is marked by AND/OR label, P/A graphs specify both fan-in and fan-out branching for nodes.

Finally, we should mention that there is also a considerable body of literature on how implied constraints can be discovered in an automatic manner (Frisch et al. 2001; Charnley et al. 2006; Bessiere et al. 2007). However, these papers treat the automatic discovering of implied constraints from a more general perspective (proof planning, global constraints) and do not have a direct connection to the present paper.

3 P/A graph and its solution

Let G be a directed acyclic graph. A subgraph of G is called a *fan-out subgraph* if it consists of nodes $x, y_1, \ldots, y_k$ (for some k) such that each (x, y_i), $1 \leq i \leq k$, is an arc in G. Similarly, a subgraph of G is called a *fan-in subgraph* if it consists of nodes $x, y_1, \ldots, y_k$ (for some k) such that each (y_i, x), $1 \leq i \leq k$, is an arc in G. In both cases x is called a *principal node* and all $y_1, \ldots, y_k$ are called *branching nodes*.

Definition 1 A directed acyclic graph together with a set of its pairwise edge-disjoint fan-out and fan-in subgraphs, where each subgraph in the set is marked either as a *parallel* subgraph or an *alternative* subgraph, is called a *P/A graph*. An *assignment* of 0/1 (false/true) values to nodes of a given P/A graph is called *feasible* if

- in every parallel subgraph all nodes are assigned the same value (both the principal node and all branching nodes are either all 0 or all 1),
- in every alternative subgraph either all nodes (both the principal node and all branching nodes) are 0 or the principal node and exactly one branching node are 1 while all other branching nodes are 0.

It can be easily noticed that given an arbitrary P/A graph the assignment of value 0 to all nodes is always feasible. On the other hand, if some of the nodes are required to take value 1 (this requirement is a very natural one if the P/A graph is used to model a real-life problem where the nodes to be valid correspond to custom orders that must be fulfilled), then the existence of a feasible assignment is by no means obvious. Let us now formulate this decision problem formally.

Definition 2 *P/A graph assignment problem* is given by a P/A graph G and a list of nodes of G which are assigned value 1. The question is whether there exist a feasible assignment of 0/1 values to all nodes of G which extends the prescribed partial assignment.

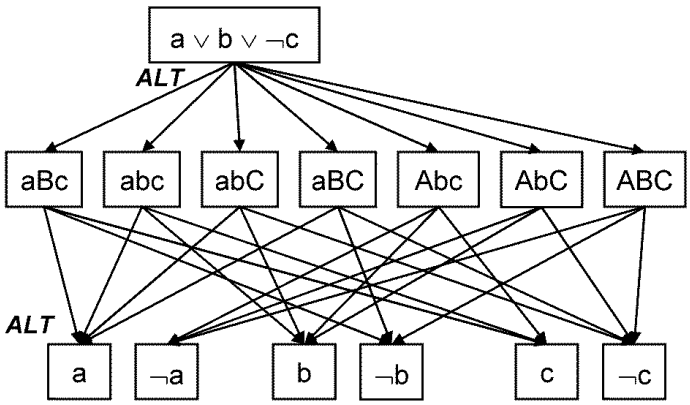


Fig. 3 Representation of a clause as a P/A graph

Remark The P/A graph assignment problem remains the same if we allow forcing value 1 for just a single vertex. To see this, observe that the general case can be reduced to this special one by adding an extra vertex, forcing it to 1 and connecting it by a fan-in (or fan-out) parallel subgraph to all nodes that were forced to 1 originally. Moreover, if the original graph was acyclic, then so is the new one.

Proposition 1 *The P/A graph assignment problem is NP-complete.*

Proof The problem is obviously in NP, because it suffices to guess the assignment and test its feasibility, which can be done in linear time in the number of parallel and alternative subgraphs (and hence in the number of edges). For the NP-hardness, we shall show that the 3SAT problem, which is known to be NP-complete (Garey and Johnson 1979), can be reduced (in a polynomial time) to the P/A graph assignment problem. Recall that the 3SAT problem is a problem of deciding whether there exists a model (a satisfying assignment of truth values to propositional variables) for a given formula in a conjunctive normal form (CNF), where each clause in the formula consists of exactly three literals. Moreover we may assume that no variable appears twice in a single clause, that is, each clause consists of literals of three distinct variables.

Now we shall describe how to construct, for a given CNF (an instance of 3SAT), an instance of the P/A graph assignment problem. Consider for example a clause $(a \vee b \vee \neg c)$. There exist seven mutually exclusive assignments of truth values to variables a, b, and c satisfying this clause (each assignment except of $a = \mathit{false}$, $b = \mathit{false}$, $c = \mathit{true}$ is a satisfying one). We can model this clause using a "clause subgraph" which consists of a node for the clause, seven nodes for the mutually exclusive satisfying assignments, and six nodes for the values of propositional variables (three for positive values and three for negative values, that is, one for each literal). The clause node is connected to all assignment nodes by a fan-out alternative subgraph and each value node is connected to appropriate assignment nodes (those assignment nodes containing the literal which corresponds to the given value node) by a fan-in alternative subgraph. Figure 3 shows the clause subgraph for the clause $(a \vee b \vee \neg c)$, where capital letters in the assignment nodes represent value *false* (so for example aBc corresponds to $a = \mathit{true}$, $b = \mathit{false}$, $c = \mathit{true}$).

Each clause from the input CNF will be modelled using a clause graph with the above-described structure. To connect the clause graphs, we introduce a formula node and connect it with all clause nodes by a fan-out parallel subgraph. The formula node is forced to take value 1 (because we need the formula to be satisfied). A variable which is used in more than one clause will have value nodes in all clause graphs where it appears. To interconnect these

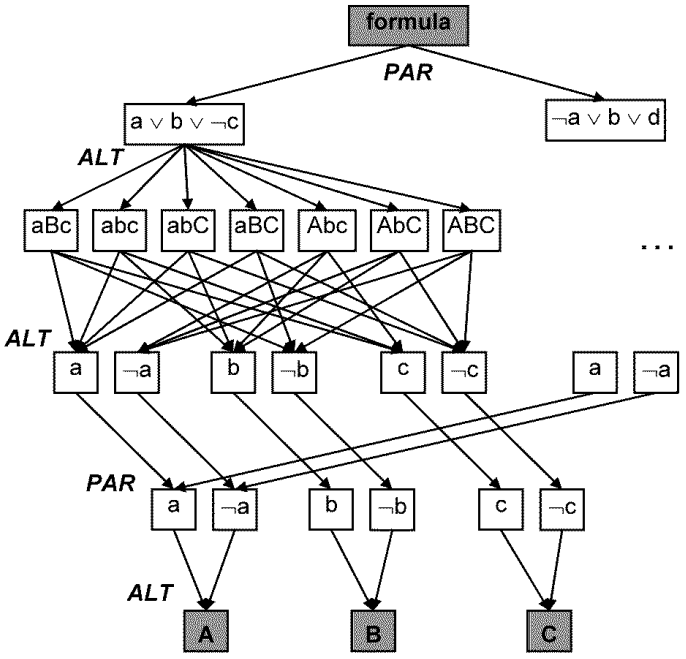


Fig. 4 Representation of a 3SAT formula as a P/A graph

value nodes we introduce for each variable in the formula a variable node, which is forced to take value 1, and two literal nodes connected to the variable node by a fan-in alternative subgraph. Finally, each literal node is connected by a fan-in parallel subgraph to all value nodes in clause graphs which correspond to the given literal. Figure 4 shows these additional nodes and connections (the shaded nodes are the nodes that are forced to take value 1).

First let us observe that the number of nodes in the constructed P/A graph is linear in the size of the input CNF formula. Namely, if there are M clauses and N variables (and hence $L = 3M$ literals) in the input CNF, then we get a graph with $(14M + 3N + 1)$ nodes. Because $14M + 1 \leq 5L$ and $N \leq L$ (assuming each variable appears at least once in the formula) we get that there are at most $8L$ nodes in the constructed P/A graph.

Now let us assume that the input CNF has a satisfying assignment. We shall construct a feasible assignment of the constructed P/A graph as follows. All clause nodes will get value 1 to satisfy the parallel fan-out from the formula node. The literal nodes of each variable will get the 0 and 1 values as defined by the satisfying assignment of the input CNF (for example if variable b is false in the satisfying assignment, then the node b gets the value 0 and the node $\neg b$ gets the value 1). This satisfies the alternative fan-in into the variable nodes, and moreover it defines the 0 and 1 values for all value nodes via the parallel fan-ins that replicate the literal values into all clause subgraphs. Finally, for each clause exactly one assignment node is made valid, namely the one in which all three literals are valid, which satisfies the alternative fan-out from the clause nodes. It remains to show that also all alternative fan-ins into value nodes are satisfied. So let us consider an arbitrary value node. If it corresponds to a valid literal then it is connected to exactly one valid assignment node (the one where also the other two literals are valid), and if it corresponds to an invalid literal then it is connected only to invalid assignment nodes (see Fig. 5). In both cases this is exactly what we need and hence the constructed assignment of 0/1 values to all nodes is feasible. Figure 5 shows how the satisfying assignment looks in the fraction of the P/A graph.

To complete the proof let us assume that there exists a feasible assignment of 0/1 values to all nodes of the constructed P/A graph. In this assignment:

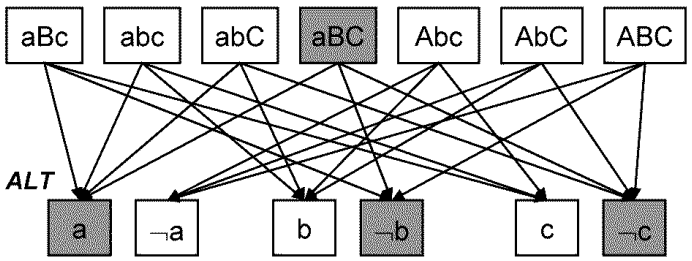


Fig. 5 A satisfying assignment of nodes

- All clause nodes have value 1 to satisfy the parallel fan-out from the formula node.
- For each clause exactly one assignment node has value 1 to satisfy the alternative fan-out from the clause nodes.
- For each variable one literal node has value 1 and the other has value 0 to satisfy the alternative fan-ins into the variable nodes. The literal values are replicated into the value nodes by the parallel fan-ins into the literal nodes.

Now let us check that the truth assignment defined by the values assigned to the literal nodes satisfies the input CNF. To this end let us pick an arbitrary clause and assume by contradiction that it is falsified. That means that the three valid value nodes correspond to the only missing combination among the assignment nodes, or in other words that the only valid assignment node must be connected to an invalid value node. However, this is a contradiction, because the corresponding fan-in subgraph into this value node spoils the feasibility of the assignment (the principal node is 0 while one of its branching nodes is 1). Hence, the input CNF has a satisfying assignment if and only if the corresponding P/A graph has a feasible assignment. □

4 Equivalence classes in P/A graphs

Solving the P/A graph assignment problem is hard so we will focus now on inferring some information from the graph that can be used later to improve efficiency of the constraint model. In particular, we will describe techniques for identification of logically equivalent nodes. We call a set of nodes of the P/A graph equivalent if and only if the nodes are assigned the same value in all feasible assignments of 0/1 values to nodes. Unfortunately, as we shall show later, the problem of finding the largest possible sets of equivalent nodes is also hard. Let us formalize the problem and prove its hardness.

Definition 3 Let G be a P/A graph, S_1 be a set of nodes of G which are fixed to value 1, and S_0 be a set of nodes of G which are fixed to value 0. Let u and v be arbitrary two nodes of G. Then u and v are called *equivalent* with respect to the partial assignment given by S_1 and S_0 if and only if there is no feasible assignment of G extending the given partial assignment in which u and v are assigned different values.

Definition 4 The *P/A graph equivalence class problem* can be stated as follows: given a P/A graph G, a set S_1 of nodes of G which are assigned value 1, and a set S_0 of nodes of G which are assigned value 0, output the partition of all nodes of G into equivalence classes, that is, into such sets that every two nodes from the same set are equivalent and no two nodes from distinct sets are equivalent.

Proposition 2 *The P/A graph equivalence class problem is NP-hard.*

Proof It is enough to prove the following: if there exists an algorithm A which solves the P/A graph equivalence class problem in polynomial time, then such an algorithm can be used to solve the P/A graph assignment problem (which is, as we have proved in the previous section, NP-complete) in polynomial time. So let us assume that G is a P/A graph, S_1 is a nonempty set of nodes of G which are fixed to value 1, and the question is whether there exists a feasible assignment of values to the remaining nodes.

Before we proceed further let us first decide, what output do we expect from algorithm A in the case when no feasible assignment exists. Strictly speaking, in that case the algorithm should output a single equivalence class, as every two vertices are equivalent according to the definition (there exists no feasible assignment in which they are assigned different values). If we adhere to this strict interpretation, a single run of A solves the assignment problem. Indeed, if all vertices end up in a single class then it suffices to test whether the all-one assignment is feasible (a feasible assignment exists) or not (a feasible assignment does not exist). If algorithm A returns at least two equivalence classes then a feasible assignment exists. However, it may be reasonable to look at the requirements on A in the following way: if no feasible assignment exists then there is an empty set of restrictions on equivalence classes and A may output arbitrary sets. So let us now consider this less strict version of A. After running A, three situations can happen:

- The set S_1 is split among several (at least two) equivalence classes. Recall, that vertices in S_1 are pre-assigned to 1 so they are equivalent. If algorithm A puts them in different equivalence classes then it corresponds to non-existence of feasible assignment as described above (algorithm A outputs arbitrary sets).
- All vertices end up in a single class. As above, in this case it suffices to test whether the all-one assignment is feasible (a feasible assignment exists) or not (a feasible assignment does not exist).
- The set S_1 is contained in a single equivalence class (let us denote it by C) but there exist at least one additional nonempty equivalence class.

In the last case we shall proceed as follows. Let u be an arbitrary node not in C, that is, u is not equivalent with nodes in S_1. If there exists a feasible assignment, then there exists at least one in which u gets value 0 (otherwise u is equivalent with nodes in S_1). So we may proceed by setting $S_1 \leftarrow C$ and $S_0 \leftarrow S_0 \cup \{u\}$, and running A on the new data. Again, the above three cases may happen, and we iterate the process, until either it terminates by arriving to the first or second case, or it subsequently inserts all nodes in $S_0 \cup S_1$ (in which case there is a single assignment to test for feasibility). Since every run of A adds at least one node to the set $S_0 \cup S_1$, the maximum number of runs of A is bounded by the number of nodes in G. Therefore, if A runs in polynomial time, then so does the above described algorithm for testing the existence of a feasible assignment. Notice that the algorithm is constructive in the sense, that it not only decides about the existence, but it in fact constructs a feasible assignment if one exists. □

5 Identifying (some) equivalence classes in P/A graphs

Since the P/A graph equivalence class problem is hard, we now focus on discovering certain typical equivalence classes that appear frequently in real-world problems. The most important situation we want to recognize consists of a process which splits in several subprocesses in alternative branching and all these sub-processes join afterwards (an example is given in Fig. 1). Principal nodes where the production process splits and sub-processes join

back again are equivalent. We are looking for the heuristic techniques that can discover at least such equivalence classes. In the subsequent sections, we will describe three techniques for identifying such equivalence nodes.

5.1 Graph modification by rules

The first method consists of application specific rules that modify the graph. Briefly speaking, the method expands and contracts some branchings and during this process it can identify equivalent nodes that are either in a parallel branching, where all the nodes are logically equivalent, or that are at the beginning and at the end of some alternative branching. The proposed algorithm has two phases. In the initial phase an undirected hyper-graph is constructed from the input P/A graph. The constructed hyper-graph has almost the same structure and represents almost the same information about the production processes as the original P/A graph. Only the directions of arcs and hence precedence relations are omitted, which is not a problem because the input P/A graph is acyclic so the precedence relations trivially hold (actually, the precedence relations are used only to define fan-in and fan-out subgraphs in acyclic P/A graphs).

The second and major phase of the algorithm repeatedly transforms the given hyper-graph using certain transformation rules into a simpler and more explicit hypergraph. Sets of equivalent nodes of the input P/A graph are built along these transformation steps. This phase terminates when no transformation rule can be applied or when a conflict in the hyper-graph is detected. The correctness of the transformation rules is proved in the appendix.

5.1.1 Initial phase of the algorithm

Let $G = (V, E)$ be a P/A graph represented as sets of marked fan-in and fan-out sub-graphs (Fig. 6). Some nodes of G may be forced to take the value 1 (filled by gray in Fig. 6) and some nodes may be fixed to value 0. We construct a hyper-graph $H = (U, F)$ over a set of nodes U obtained by adding two extra nodes 0 and 1 to the original set V. The extra nodes 0 and 1 represent equivalence classes of nodes which are always 0 and 1 respectively in all feasible assignments. The construction of the set of hyper-edges F is done according to the input graph G in the following way. Let set F be empty at the beginning.

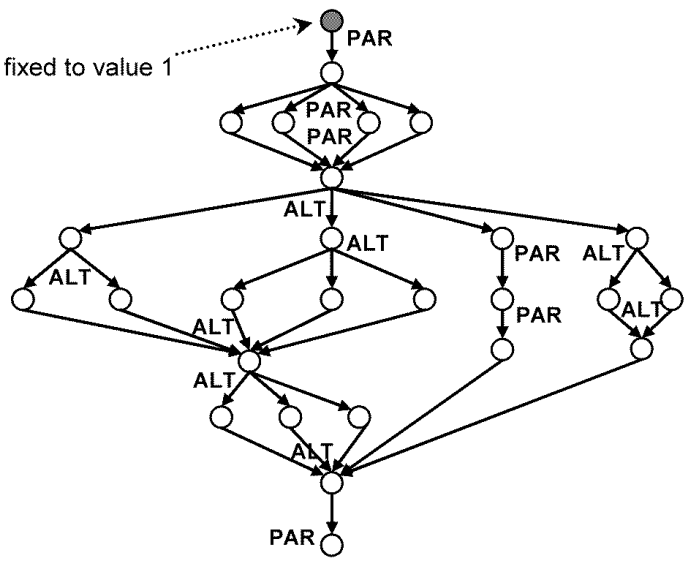


Fig. 6 Example of P/A graph with PAR/ALT annotation

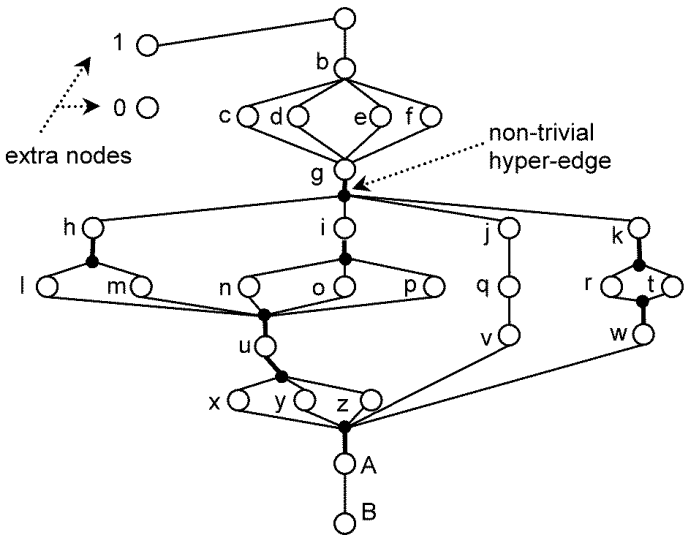


Fig. 7 Hyper-graph corresponding to the P/A graph

- For each *fan-in parallel* sub-graph of G over nodes $x, y_1, y_2, \ldots, y_k$, where x is the principal node, insert edges $\{\{x\}, \{y_i\}\}$ for $1 \leq i \leq k$ into F.
- *Fan-out* sub-graphs of G marked as *parallel* are treated in the same way as fan-in parallel sub-graphs.
- For each *fan-in alternative* sub-graph of G over nodes $x, y_1, y_2, \ldots, y_k$, where x is the principal node, insert non-trivial hyper-edge $\{\{x\}, \{y_1, y_2, \ldots, y_k\}\}$ into F.
- As in the case of parallel branching, *fan-out* sub-graphs of G marked as *alternative* are treated in the same way as fan-in alternative sub-graphs.
- For each node x of G which is *forced* to take the value 1 we insert edge $\{\{x\}, \{1\}\}$ into F. An analogical edge addition is done for nodes which are fixed to value 0. Edge $\{\{y\}, \{0\}\}$ is inserted into F for each node y of G which is forced to take the value 0.

Informally speaking, we use the same nodes in the hyper-graph as in the P/A graph. For each arc that is a part of parallel branching in the P/A graph we add an edge between the same nodes in the hyper-graph. For a set of arcs that form alternative branching in the P/A graph, we add a non-trivial hyper-edge connecting the same nodes in the hyper-graph (a small black dot in Fig. 7). We use the convention that an edge with the same structure is added only once (we do not allow multi-edges). Figure 7 shows a hyper-graph constructed for the P/A graph from Fig. 6.

The last step of the initial phase is constructing the initial equivalence classes. Let us denote Q_u the equivalence class for $u \in U$. Initially we set $Q_u = \{u\}$ for every $u \in U$. The constructed hyper-graph $H = (U, F)$ with associated equivalence classes is used as the input for the transformation phase of the algorithm.

5.1.2 Transformation phase of the algorithm

The goal of the transformation phase is to modify the hyper-graph while preserving the equivalence classes. This is realized by several transformation rules that update the hyper-graph by adding derived hyper-arcs. During these updates, the initial equivalence classes are being merged.

Edge contraction rule The first transformation rule contracts an edge. An edge $\{\{u\}, \{v\}\} \in F$ can be contracted if for any non-trivial hyper-edge $\{\{x\}, Y\} \in F |\{u, v\} \cap (\{x\} \cup Y)| \leq 1$ and

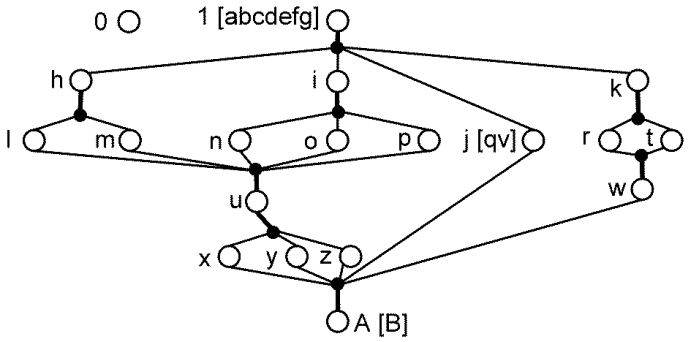


Fig. 8 Hyper-graph after edge contractions

$|\{0,1\} \cap (Q_u \cup Q_v)| \leq 1$. The case when an edge cannot be contracted is treated separately (see the rules below). The edge contraction rule represents a standard operation from the graph theory.

Let $\{\{u\},\{v\}\} \in F$ be the edge that can be contracted. Then the following steps are carried out. Erase vertex v by assigning: $U \leftarrow U - \{v\}$ and $Q_u \leftarrow Q_u \cup Q_v$. Edges and hyper-edges that contain v need to be modified to form a correct hyper-graph without v. If there is an edge $\{\{x\},\{v\}\} \in F$, where $x \neq u$, replace it by edge $\{\{x\},\{u\}\}$. If there is a non-trivial hyper-edge $\{\{v\}, Y\} \in F$ then replace it by $\{\{u\}, Y\}$. If there is a non-trivial hyper-edge $\{\{x\}, Y\} \in F$, where $v \in Y$, $u \notin Y$, and $x \neq u$ then replace it by hyper-edge $\{\{x\}, (Y - \{v\}) \cup \{u\}\}$.

Figure 8 shows a hyper-graph after applying the edge contraction rule to the hyper-graph in Fig. 7. Namely, we contracted all edges between nodes a and g (and 1) to obtain a single node for equivalence class $\{1, a, b, c, d, e, f, g\}$. We also contracted edges $\{\{j\},\{q\}\}$ and $\{\{q\},\{v\}\}$ and got a node with equivalence class $\{j, q, v\}$. Finally, we contracted edge $\{\{A\},\{B\}\}$ to obtain a node for equivalence class $\{A, B\}$.

Hyper-edge extension rule If there are non-trivial hyper-edges $\{\{x\}, Y\} \in F$ and $\{\{y\}, Z\} \in F$, where $y \in Y$ and $(\{x\} \cup Y) \cap (\{y\} \cup Z) = \{y\}$ then add a new hyper-edge $\{\{x\}, Y \cup Z - \{y\}\}$ into F. It may happen that the new edge generated by this rule is already present in the hyper-graph. Then the rule does not change the hyper-graph.

Hyper-edge meet rule If there are non-trivial hyper-edges $\{\{x\}, Y\} \in F$ and $\{Z, \{w\}\} \in F$, where $x \neq w$ and $Z \subseteq Y$ then add a new hyper-edge $\{\{x\}, (Y - Z) \cup \{w\}\}$ into F. Again the rule has no effect if the edge generated by this rule is already present in the hyper-graph. A special case of this rule when $Y = Z$ results in addition of a new edge $\{\{x\},\{w\}\}$ that can be contracted subsequently. This transformation rule in cooperation with the previous rule can discover the situation when a production chain splits into several alternatives and all these alternatives join again.

The hyper-graph in Fig. 9 was obtained by applying the hyper-edge meet rule to hyper edges $\{\{k\},\{r,t\}\}$ and $\{\{w\},\{r,t\}\}$ that lead to adding a new edge $\{\{k\},\{w\}\}$. This edge was then contracted to find a new equivalence class $\{k, w\}$. The new bottom hyper-edge $\{\{A\},\{u, j, k\}\}$ was then obtained by applying the hyper-edge meet rule using hyper-edge $\{\{u\},\{x, y, z\}\}$. The new top hyper-edge was obtained by applying hyper-edge extension rule.

Never-valid activity detection rule (case A) If there is a non-trivial hyper-edge $\{\{x\}, Y\} \in F$ and an edge $\{\{y_1\},\{y_2\}\} \in F$, where $\{y_1, y_2\} \subseteq Y$ (notice that the edge $\{\{y_1\},\{y_2\}\}$ cannot be contracted in this situation since contraction precondition is not satisfied), then replace the hyper-edge $\{\{x\}, Y\}$ by a new hyper-edge $\{\{x\}, Y - \{y_1, y_2\}\}$ and add new edges $\{\{0\},\{y_1\}\}$

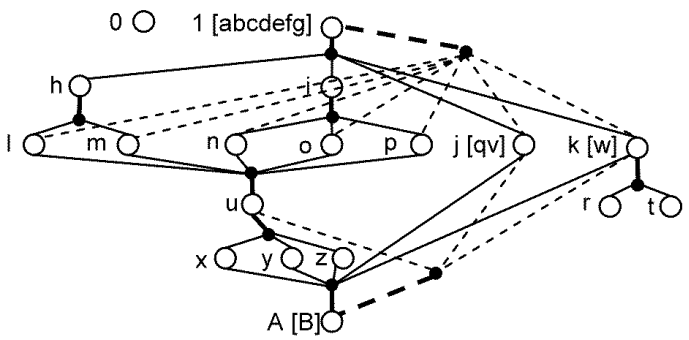


Fig. 9 Hyper-graph after extension and meet rule application

and $\{\{0\}, \{y_2\}\}$. This case describes the situation when two branching nodes of alternative branching are connected by an arc. As these two nodes must be equivalent and mutually exclusive at the same time, the only feasible assignment is having both nodes invalid.

Never-valid activity detection rule (case B) If there is a non-trivial hyper-edge $\{\{x\}, Y\} \in F$ and an edge $\{\{x\}, \{y\}\} \in F$, where $y \in Y$ (again notice that the edge $\{\{x\}, \{y\}\}$ cannot be contracted in this situation), then erase the hyper-edge $\{\{x\}, Y\}$ from F and add new edges $\{\{0\}, \{z\}\} \in F$ for all $z \in Y - \{y\}$. This case describes the situation when an arc is a part of alternative branching and hence one branching node is equivalent to the principal node. Consequently, all other branching nodes must be invalid.

Conflict detection rule The case when an edge $\{\{u\}, \{v\}\}$ that cannot be contracted due to the violated inequality $|\{0, 1\} \cap (Q_u \cup Q_v)| \leq 1$ (i.e. $|\{0, 1\} \cap (Q_u \cup Q_v)| = 2$ holds) occurs in the hyper-graph indicates a conflict. This conflict means that a set of nodes which are always assigned 0 is identical to the set of nodes which are always assigned 1 in every feasible assignment of 0/1 values to nodes. In other words, this indicates the situation that there is no feasible assignment for the given input.

If any of the above-defined transformation rules cannot be applied, the algorithm terminates with success. After successful termination, the node equivalence classes associated with nodes remaining in the final hyper-graph represent the sets of equivalent nodes.

5.2 Detection of nests

When we analyzed how the P/A graphs modeling real-life processes look, we noticed several typical features. First, the process has usually one start point and one end point. Second, the graph is built by decomposing meta-processes into more specific processes until non-decomposable processes (operations) are obtained. There are basically three types of decomposition.

- The meta-process can split into two or more processes that run in a *sequence*, that is, after one process is finished, the subsequent process can start.
- The meta-process can split into two or more sub-processes that run in *parallel*, that is, all sub-processes start at the same time and the meta-process is finished when all sub-processes are finished.
- Finally, the meta-process may consists of several *alternative* sub-processes, that is, exactly one of these sub-processes is selected to do the job of the meta-process.

Notice, that the last two decompositions have the same topology of the network (Fig. 10), they only differ in the meaning of the branches in the network.

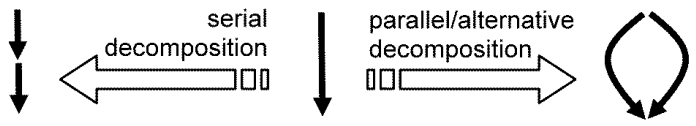


Fig. 10 Possible decomposition of meta-process into more specific processes

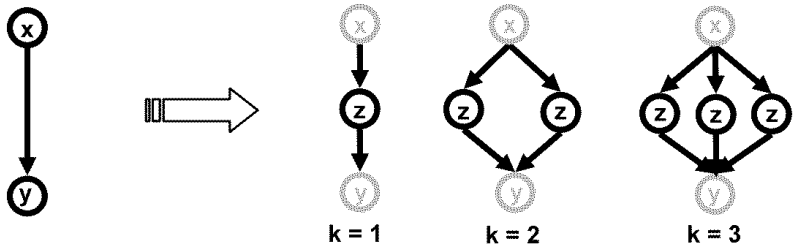


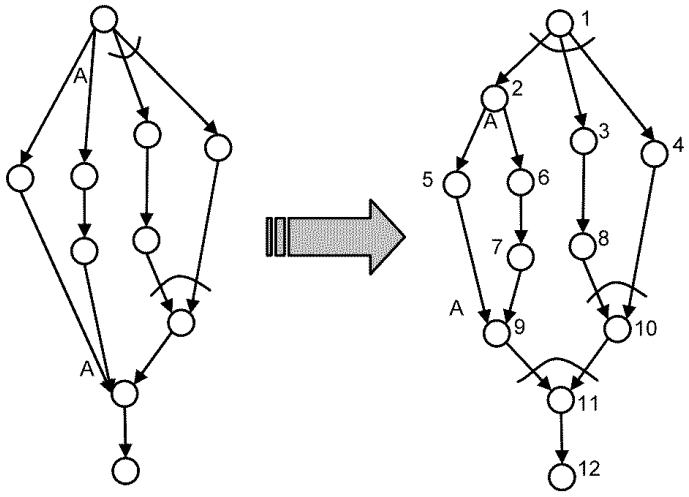
Fig. 11 Arc decomposition to form a nest

The above decompositions can be modeled by a single graph operation of splitting an arc into a so called nest as the following definition formally describes. Figure 11 shows this decomposition operation.

Definition 5 Let (x, y) be an arc in the graph, and $z_1, \ldots, z_k$ $(k > 0)$ be nodes such that neither z_i is in the graph. The *nest* is obtained by substituting the arc (x, y) by the graph consisting of nodes $x, y, z_1, \ldots, z_k$ and arcs $\{(x, z_i), (z_i, y) | i \in \{1, \ldots, k\}\}$.

Notice that if $k = 1$ then the arc decomposition describes a serial decomposition and if $k > 1$ then the arc decomposition describes a parallel or an alternative decomposition. To distinguish between the parallel and alternative decompositions, one needs to specify the type of fan-out graph with the principal node x and the type of fan-in graph with the principal node y. Clearly, both types should be identical, for example, if we split the process in x to parallel processes, we should join back these parallel processes in y. If the user specifies different types of fan-in and fan-out subgraphs then all nodes in the nest are forced to be invalid to satisfy the constraints. Assume, for example, that the fan-out subgraph with the principal node x is parallel and the fan-in subgraph with the principal node y is alternative. Then we get constraints $\forall i \in \{1, \ldots, k\} x = z_i$ and $y = \sum_{i=1,\ldots,k} z_i$ to model this situation. Clearly, the only solution to such constraints for Boolean variables is assigning value 0 to all variables (we assume the case $k > 1$). Though this is theoretically possible and P/A graphs allow such situation, practically it means that there is some mistake in the description of the process. In the algorithm described later in this section we can detect such situations and define the constraints (assignment to 0) accordingly.

The above decomposition operation has been proposed in Barták and Čepek (2007) to construct P/A graphs with a specific structure, so called *nested P/A graphs*. The paper also shows that the P/A graph assignment problem is tractable for nested P/A graphs and the algorithm to detect whether the input P/A graph is nested or not is presented there. We shall now modify this algorithm to construct an enhanced constraint model that propagates better than the basic model. Basically, this method attempts to reconstruct the process of building the P/A graph by above described decompositions or in other words, the algorithm identifies the nests and then it collapses each nest into a single arc while defining the constraints between the nodes in the nest. The algorithm has a single assumption about the input P/A graph. If there is a fan-in (fan-out) subgraph specified for a given node then this subgraph contains



node	*pred*	*succ*	*inLab*	*outLab*
1		2,3,4	–	PAR
2	1	5,6	–	ALT
3	1	8	–	–
4	1	10	–	–
5	2	9	–	–
6	2	7	–	–
7	6	9	PAR	–
8	3	10	PAR	–
9	5,7	11	ALT	–
10	4,8	11	PAR	–
11	9,10	12	PAR	–
12	11		PAR	–

Fig. 12 Representation of a (nested) P/A graph

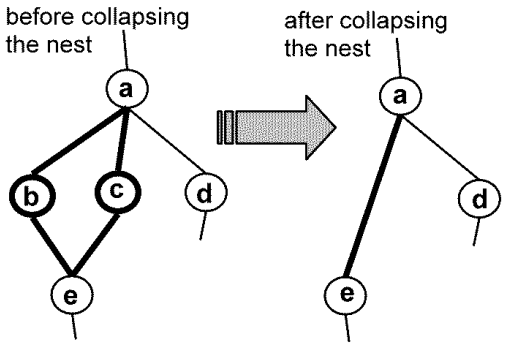
all incoming (outgoing) arcs for this node. As a consequence, we can put the subgraph type as an input or output label of the principal node. Figure 12 (right) shows a representation of the P/A graph satisfying this assumption. Note that this assumption is not restrictive in any way because if there are more subgraphs with the identical principal node (even of different types) then it is possible to add new auxiliary nodes modeling these sub-graphs (see Fig. 12 left).

As we already mentioned, the algorithm is based on, first, finding the nest and, second, collapsing the nest into a single arc and defining constraints for the nest. This process is repeated until no nest is found and then the basic constraint model is used for the rest.

The nest can be identified by finding all nodes z_i from Definition 5. A common feature of these nodes is having exactly one incoming and one outgoing arc and there must be identical predecessor (x) and identical successor (y). Moreover, for a complete nest, it is necessary that nodes z_i are either the only predecessors of y (no other predecessor of y different from z_i) or the only successors of x. If the only predecessors of y are nodes z_i then the *leading node* of the nest is y, otherwise the leading node is x.

Now we shall collapse the nest into arc (x, y) and define the constraints modeling the nest. Let k be the number of nodes z_i in the nest (Definition 5). If $k = 1$ then we include constraint $z_1 = leading\_node$. Let us assume $k > 1$ for the rest of the paragraph. If the output label of x is different from the input label of y then we have the situation discussed before—validity of all nodes participating in the parallel subgraph is forced to be 0. For example, if there is a parallel fan-in subgraph with the principal node y then we add constraints $y = 0$ and $\forall i z_i = 0$. Let us now assume that the output label of x equals to the input label of y. If the subgraphs are parallel ($\mathrm{InLab}(y) = \mathrm{OutLab}(x) = \mathrm{PAR}$) then we add constraints $\forall i\, leading\_node = z_i$. If the subgraphs are alternative ($\mathrm{InLab}(y) = \mathrm{OutLab}(x) = \mathrm{ALT}$) then we add constraint $leading\_node = \sum_i z_i$. Figure 13 shows original constraints from the basic model and new constraints from the enhanced model.

Notice that when collapsing the nest, we did not define a constraint between x and y. This is because this constraint will be added later when arc (x, y) will be removed from the graph, either by collapsing another nest or as a part of the basic constraint model. The consequence is that for nested P/A graphs we obtain a Berge acyclic constraint network which means that arc consistency implies global consistency (Beeri et al. 1983). If we obtain a P/A



outLab(a)	*inLab(e)*	*basic*	*enhanced*
PAR	PAR	a = b = c = d	a = e = d
		e = b = c	e = b = c
PAR	ALT	a = b = c = d	a = e = d
		e = b + c	a = b = c = 0
ALT	PAR	a = b + c + d	a = e + d
		e = b = c	e = b = c = 0
ALT	ALT	a = b + c + d	a = e + d
		e = b + c	e = b + c

Fig. 13 Original (basic) and new (enhanced) constrains modeling a nest

graph which does not contain any other nest, we can use the basic constraint model for this graph (see Introduction). In such a case Berge acyclicity is not guaranteed and arc consistent constraint model does not need to be globally consistent (search may be necessary to solve the P/A graph assignment problem). Nevertheless, at least we have stronger propagation for nested sub-parts while still generating a constraint model for any P/A graph. Note finally that the equality constraints in the enhanced constraint model define (some) equivalent nodes. In particular, if constraint $x = y$ is added to the model then nodes x and y are equivalent according to Definition 3.

The procedure `DetectNested` describes the above process of collapsing nests and generating constraints. Its time complexity is $O(n^2)$, where n is a number of nodes in the graph (Barták and Čepek 2007). The soundness can be formally proved by showing that the generated constraints are equivalent to the original constraints, which is a simple generalization of the table in Fig. 13.

procedure *DetectNested*(input: graph G, output: constraint model for G)
1. C ← {}
2. select all nodes *x* in G such that |*pred*(*x*)| = |*succ*(*x*)| = 1
3. sort selected nodes lexicographically according to index (*pred*(*x*), *succ*(*x*)) to form queue Q
4. **while** non-empty Q **do**
5. select and delete a sub-sequence L of size *k* in Q such that
   all nodes in L have an identical index ({*x*}, {*y*}) and either |*succ*(*x*)| = *k* or |*pred*(*y*)| = *k*
6. **if** no such L exists **then** exit the loop
7. **if** |*succ*(*x*)| = *k* **then** lead ← *x* **else** lead ← *y*
8. **case**
9. *k* = 1: C ← C ∪ { lead = *z* | *z* ∈ L }
10. *k* > 1 & *outLab*(*x*) ≠ *inLab*(*y*): C ← C ∪ { *a* = 0 } ∪ { *z* = 0 | *z* ∈ L }
    where *a* ← *x* if outLab(*x*) = PAR, *a* ← *y* otherwise
11. *k* > 1 & *outLab*(*x*) = *inLab*(*y*) = PAR: C ← C ∪ { lead = *z* | *z* ∈ L }
12. *k* > 1 & *outLab*(*x*) = *inLab*(*y*) = ALT: C ← C ∪ { lead = $\Sigma_{z \in L}$ *z* }
13. remove nodes z∈L from the graph G
14. remove nodes *x*, *y* from Q (if they are there)
15. add arc (*x*,*y*) to the graph G (an update *succ*(x) and *pred*(y))
16. **if** |*pred*(*x*)| = |*succ*(*x*)| = 1 **then** insert *x* to Q
17. **if** |*pred*(*y*)| = |*succ*(*y*)| = 1 **then** insert *y* to Q
18. **end while**
19. C ← C ∪ basic constraint model for graph G
20. return C

5.3 Singleton consistencies

While in the previous section we proposed a method that exploits a specific structure of P/A graphs called nesting, in this section we will describe a more general method that can be used to find implied constraints in any constraint satisfaction problem (Barták 2007).

As we mentioned in introduction, the problem of the basic constraint model for the P/A graph assignment problem is weak propagation via arc consistency. The propagation can be easily strengthen by a trial-and-error method similar to shallow backtracking or SAC. Recall the example from Introduction with constraints $A = B + C$, $D = B + C$, $A = 1$ over Boolean variables. AC does not prune the domain of D, but if we assign value 0 to D the AC generates a failure. Basically, we will try to assign values to pairs of variables (such as A and D in the above example) and if we find (via arc consistency) that only identical values can be assigned to the variables then we deduce that the variables are equivalent (they must be assigned to the same value in any solution). As a side effect, we can also discover other binary logical constraints, namely dependencies between the variables (if 1 is assigned to B then 1 must be assigned to A) and exclusions between the variables (either B or C must be assigned to 0 or in other words it is not possible to assign 1 to both variables B and C).

We will now present the learning method for an arbitrary constraint satisfaction problem P. Recall, that we will only learn specific logical relations between the Boolean variables of P. We will assign a value to some variable with non singleton domain and then we will gradually try to assign values to the other variables and each time we try the assignment, this assignment is propagated to other variables via AC. If the assignment leads to a failure then we know that the other value in the domain must be assigned to the variable (recall that we are working with Boolean variables). The whole learning process consists of two stages.

First, we collect information about which variables are instantiated after assigning value 1 to some variable A. We distinguish between *directly instantiated variables*, that is, those variables that are instantiated by making the problem $\mathrm{P}|_{A=1}$ arc consistent (one value in the variable domain is refuted by AC so the other value is used), and *indirectly instantiated variables*, that is, those variables where we found their value by refuting the other value in a SAC-like style (AC did not prune the domain, but when we try to assign a particular value to the variable it leads to a failure so the other value is used). Informally speaking, if we assign value 1 to variable A and make the problem arc consistent then all variables that are newly instantiated are directly instantiated variables. Indirectly instantiated variables are those variables B that are not instantiated by AC in $\mathrm{P}|_{A=1}$ but for which only one value is compatible with $A = 1$ because if the other value is assigned to B, it leads to a failure after making the problem AC (see procedure `Learn` below). More formally, let B be a non-instantiated (free) Boolean variable in $\mathrm{AC}(\mathrm{P}|_{A=1})$, where AC(P) is the arc consistent form of problem P (inconsistent values are removed from the domains of variables). If $\mathrm{P}|_{A=1,B=0}$ is not arc consistent then value 0 cannot be assigned to B, hence value 1 must be used for B. Symmetrically, we can deduce that value 0 must be assigned to B if $\mathrm{P}|_{A=1,B=1}$ is not arc consistent. Together, we can deduce which value must be used for B if value 1 is assigned to A. If both values for B are feasible then no information is deduced. If no value for B is feasible then value 1 cannot be used for A and hence A must be instantiated to 0. Note that information about indirectly instantiated variables is very important because it will help us to deduce implied constraints that improve propagation of the original constraint model. More formally, we are looking for implied constraints C such that $\mathrm{AC}(\mathrm{P}|_C) \subset \mathrm{AC}(\mathrm{P})$, where $\mathrm{P}|_C$ is a problem P with added constraint C and the subset relation means that all domains in $\mathrm{AC}(\mathrm{P}|_C)$ are subsets of relevant domains in AC(P) and at least one domain in $\mathrm{AC}(\mathrm{P}|_C)$ is a strict subset of the relevant domain in AC(P). In other words, constraint C helps in removing more inconsistencies from problem P.

The learning stage deduces three types of implied constraints. If $B = 0$ is indirectly deduced from the assignment $A = 1$ and $A = 0$ is indirectly deduced from the assignment $B = 1$ then the pair $\{A, B\}$ forms an exclusion, which is an implied *exclusion constraint* $(A = 0 \vee B = 0)$. Notice that this constraint really improves propagation because for example if 1 is assigned to A then the constraint immediately deduces $B = 0$, while the original set of constraints deduced no pruning for B. Similarly, if $B = 1$ is indirectly deduced from the assignment $A = 1$ then B depends on A, which is an implied *dependency constraint* $(A = 1 \Rightarrow B = 1)$. Again, this constraint improves propagation. Note that we introduce this constraint only if variables A and B are not found to be equivalent. The equivalent variables are found using the following procedure. We construct a directed acyclic graph where the nodes correspond to the variables and the arcs correspond to the dependencies between the variables. These dependencies are found in the first stage, we assume both direct dependencies discovered by the AC propagation and indirect dependencies discovered by the SAC-like propagation. Strongly connected components of this graph form *equivalence classes of variables*. Note that if A and B are in a strongly connected component then $(A = 1 \Rightarrow^* B = 1)$ and $(B = 1 \Rightarrow^* A = 1)$, where $\Rightarrow^*$ is a transitive closure of relation $\Rightarrow$. All equivalent variables must be assigned to the same value in any solution so we can put equality constraint between these variables.

The following code of procedure `Learn` shows formally both the data collecting stage and the learning stage of our method. BoolVars(P) is a set of not-yet instantiated Boolean variables in P, doms(P) are domains of P, $D_X = \{V\}$ means that the domain of variable X consists of one element V, and AC(P) is the arc consistent form of problem P (AC(P) = fail if problem P cannot be made arc consistent).

```
procedure Learn (input: CSP P, output: learned constraints)
// data collecting stage
1.   for each A in BoolVars(P) do
2.       Q ← AC(P|A=1)
3.       Direct(A) ← { X/V | DX = {V} in doms(Q) }
4.       for each B in BoolVars(Q) s.t. A ≠ B & Q ≠ fail do
5.           if AC(Q|B=0) = fail then
6.               Q ← AC(Q|B=1)
7.           else if AC(Q|B=1) = fail then
8.               Q ← AC(Q|B=0)
9.       end for
10.      Indirect(A) ← { X/V | DX = { V } in doms(Q) } – Direct(A)
11.      if Q = fail then
12.          P ← AC(P|A=0)
13.          if P = fail then stop with failure
14.  end for
// learning stage
15.  G ← (BoolVars(P), {(A,B) | B/1 ∈ Direct(A) ∪ Indirect(A)})
16.  Equiv ← StronglyConnectedComponents(G)
17.  Excl ← { {A,B} | B/0 ∈ Indirect(A) & A/0 ∈ Indirect(B)}
18.  Deps ← { (A,B) | B/1 ∈ Indirect(A) & ¬ {A,B} ⊆ X ∈ Equiv}
19.  return (Equiv, Excl, Deps)
```

The main advantage of the proposed method is simplicity and generality. Thanks to the meta-nature of singleton consistency it can be implemented easily on top of any constraint solver and it works with any constraint satisfaction problem (even if global constraints and non-Boolean variables are included). The time complexity of the data collection stage is $O(n^2.|\mathrm{AC}|)$, where n is the number of Boolean variables and |ACl is the complexity to make the problem arc consistent. Strongly connected components of the dependency graph can be

found in time not greater than $O(n^2)$ and exclusions and dependencies are generated in time $O(n^2)$. Clearly, majority of time to learn implied constraints by the above method is spent by collection information using the SAC-like method.

6 Experiments

To compare the three methods of enhancing the basic constraint model for the P/A graph assignment problem by implied constraints, we implemented all methods in SICStus Prolog 4.0.1 and performed several preliminary experiments. The tests run under Windows XP Professional on 1.1 GHz Pentium M with 1.2 GB memory. Runtime is always measured in milliseconds using predicate `statistics(runtime,[_,T])`, the number of backtracks when solving a CSP is measured using `fd_statistics(backtracks, BT)`.

In this section, we will first describe the features of the random problem generator used in the experiments. Then we will give evidence why one should look for enhancements of the basic constraint model for P/A graphs, and finally we will compare the three enhancements proposed in this paper.

6.1 Problems

In the experiments, we focused on P/A graphs with the nested structure as described in Barták and Čepek (2007) and also in Kim et al. (2001) and in Horling et al. (1999). Briefly speaking these graphs are obtained from a single arc by repetitive decompositions of arcs into nests as specified by Definition 5. The reasons for choosing this type of problems are twofold. First, it is a lack of real-life benchmark problems containing alternatives. In fact, we are not aware about any such benchmark set that is publicly available. Second, we believe that this type of networks with alternatives prevails in real-life applications, which is also supported by studies of temporal planning networks (Kim et al. 2001) and TAEMS formalism (Horling et al. 1999).

We used a "home-brewed" generator of random nested P/A graphs that is parameterised by the following arguments:

- *the number of decomposition steps*

  This is the number of times some randomly selected arc is decomposed according to Definition 5; the generator starts with a single arc. Together with the widths of decompositions (see below) it determines the number of nodes in the graph (Fig. 14).

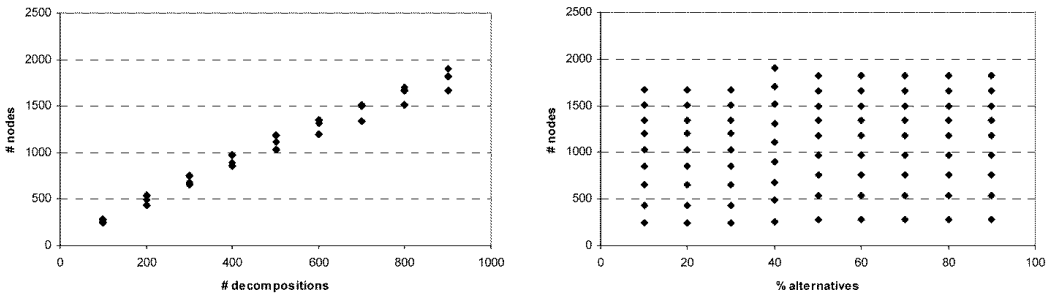


Fig. 14 The number of nodes in random nested P/A graphs depending on the generator parameters

- *the mean value of the width of decompositions based on the level*
  This describes the parameter k from Definition 5 that depends on the level of the decomposed arc. The initial arc has level 1 and each time we decompose an arc, the level of the newly introduced arcs increases by one in respect to the level of the decomposed arc. The actual k is chosen randomly using a normal distribution with the mean value given for the level of the decomposed arc and with standard deviation 0.2. Throughout our experiments, we used the following mean values of widths ordered by the level: 1, 2, 2, 2, 2, 3, 3, 3, 3, 3.
- *the probability of using an alternative branching*
  When a new nest with $k > 1$ is introduced, the type of branching, either alternative or parallel, is chosen according to given probability. Sometimes the type of branching is forced by the decomposed arc, see Barták and Čepek (2007) for details.

We generated random problems where we varied the number of decomposition steps from 100 to 900 and the probability of alternative branching from 10% to 90%. Figure 14 shows dependence of the graph size on the number of decomposition steps (left) and on the probability of alternative branching (right). The reader can see that the problem size depends mainly on the number of decomposition steps while the probability of using alternative branching does not influence it a lot (which is clear because this probability just affect the type of branching, not the graph structure). Figure 15 shows an example of a randomly generated nested P/A graph after 20 decomposition steps.

We used the experiments to validate two hypotheses. First, the basic constraint model presented in introduction does not propagate well which leads to significant increase of runtime with the increasing size and complexity of the problem. If this hypothesis is valid then it is useful to look for enhancements of that model as presented in this paper. Second, we wanted to compare efficiency of finding the enhancements of the basic constraint model. Our expectation was that the specific techniques designed for P/A graphs are more efficient than a generic approach based on singleton consistency and that the technique exploiting the nested structure outperforms the rule based technique. As we will see later, not all these expectations were fulfilled.

6.2 Problem hardness

We already know that the assignment problem for nested P/A graphs is tractable (Barták and Čepek 2007) which, however, does not imply that the problem can be solved fast using the basic constraint model. In the introduction we mentioned weak propagation of the basic constraint model which may lead to longer runtime, because this weakness must be resolved by the search algorithm. Our experiments indeed confirmed this expectation. In all experiments we used a default search strategy which first instantiates variables participating in the larger number of constraints—we use Boolean variables so this variable ordering corresponds to well-known Brélaz (1979) or dom + deg heuristic—and which first tries to assign value 1 (the node is valid).

Figure 16 shows the number of backtracks (left) and runtime (right) as a function of the number of nodes in the graph and the probability of alternative branching. We used the time limit of ten minutes for runtime and within this limit we cannot solve the largest problems where the probability of alternative branching is 50% and more.

The experiment confirms that the basic constraint model indeed does not propagate well because some backtracks are necessary to solve the problems. The experiment also shows an interesting feature of the problems. If less than half of all branchings are alternative, then the

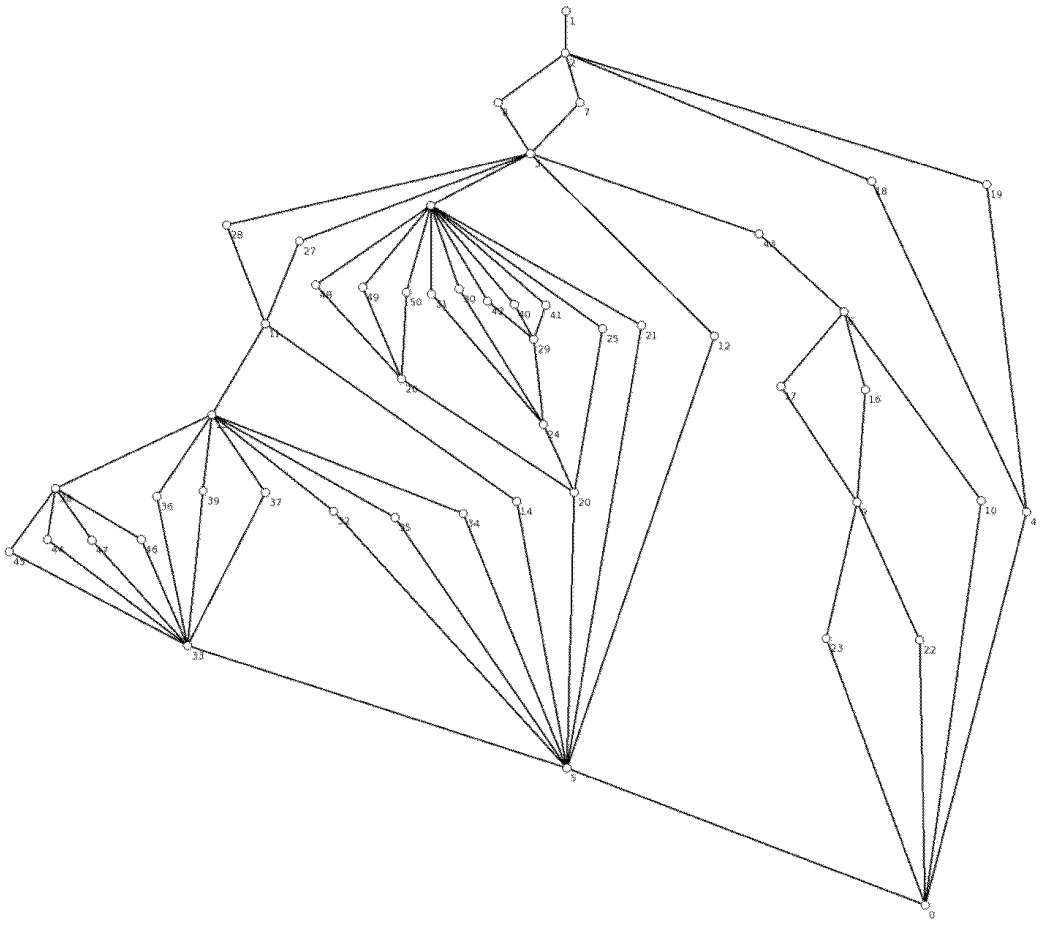


Fig. 15 Example of a randomly generated nested P/A graph

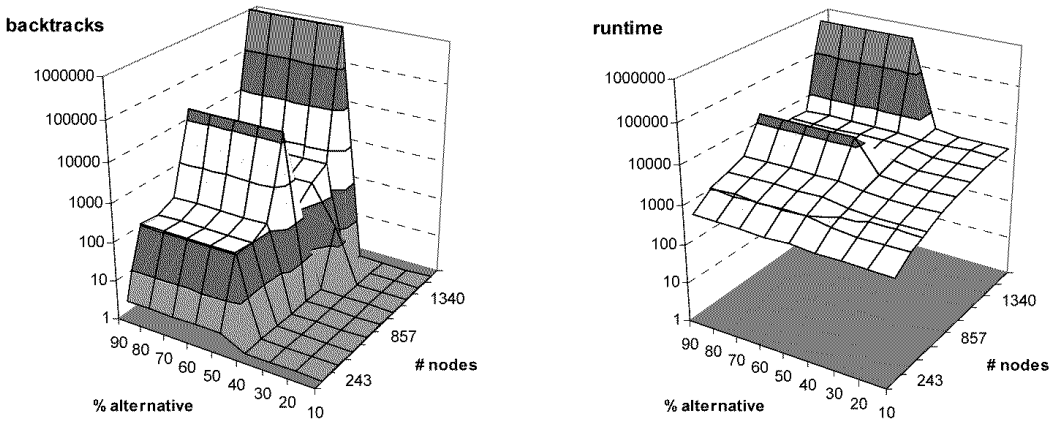


Fig. 16 The number of backtracks and runtime to solve the nested P/A graph assignment problem using the basic constraint model (logarithmic scale)

problem is easy and in most cases it can be solved without any backtracks even when the basic constraint model is used. This is because in such problems, the majority of constraints are

Table 1 Solved P/A graph assignment problems using the basic constraint model for problems with 80% of alternative branchings

Decompositions	Solved	Backtracks	Runtime (ms)
200	100%	1035	770
400	95%	38381	16415
600	75%	120143	65598
800	45%	2613	3065

equalities which propagate very well (if a value is assigned to a variable then it is propagated to many other variables), while the weak propagation is caused by the constraints modelling alternatives. When the percentage of alternative branchings reaches approximately 50 percent, the number of backtracks and runtime both jump up quite dramatically. A surprising feature is that if the probability of alternative branchings keeps increasing further above the point where the jump occurs, the number of backtracks and runtime stay level or do not increase significantly. This may require further investigation in the future.

In the previous experiment we used a single random problem for each set of parameters (81 problems in total) and the results showed an interesting complexity pattern. To explore further this complexity pattern we performed a second experiment where we fixed the probability of alternative branching to 0.8 (80%), which is in the "hard" region, we varied the number of decomposition steps from 200 to 800, and for each set of parameters we generated 20 problems. Table 1 shows the number of problems solved within the time limit of 10 minutes, the average runtime and the average number of backtracks for solved problems. Clearly, as the problem size increases, the problems are harder to solve for the basic constraint model. The decrease of runtime and the number of backtracks when going from 600 to 800 decompositions can be explained by using a time limit of 10 minutes to solve the problem. Notice that for 800 decompositions we were able to solve less than half of problems within the time limit so only the "easy" problems are evaluated regarding the runtime and the number of backtracks. This behaviour is typical for random problems with a so called heavy-tail distribution—if a problem is generated randomly using a given set of parameters of the generator, it may happen that an easy problem is generated but there is also a big chance to obtain a very hard problem.

The experiments presented on this section justify that the basic constraint model is not satisfactory and some enhancements are necessary. In particular, the runtime of the basic model is not stable due to involvement of search in the solving procedure (the basic constraint model does not guarantee global consistency via polynomial arc consistency).

6.3 Comparison of modelling methods

The second group of experiments focused on comparison of efficiency of finding the equivalence classes using the three methods presented in this paper—rules, nested method, and singleton method. In particular, we compared the runtime to generate the enhanced constraint models. The methods based on rules and singleton consistency add implied constraints to the basic constraint model, while the method identifying nests generates specific constraints modelling nests. It should be highlighted that all three enhanced models solved the P/A graph assignment problems from our experiments without any search (without backtracks) and the runtime to solve the problem was negligible in comparison to the runtime to generate the enhanced model. Backtrack-free search is theoretically guaranteed for the nested model (Barták and Čepek 2007) when solving the nested P/A graph assignment problem

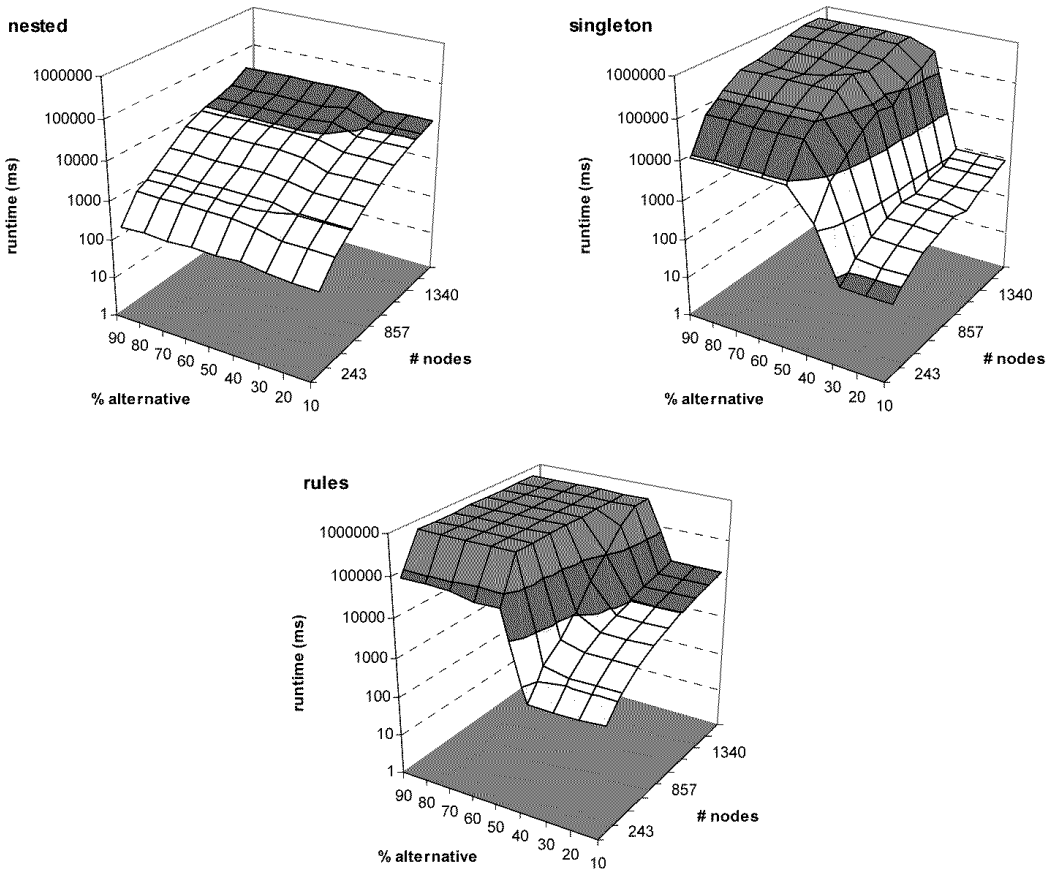


Fig. 17 Runtime to generate the enhanced model by studied methods for nested P/A graphs (logarithmic scale)

and the experiments showed that the implied constraints added by the other two methods also considerably contribute to pruning the search space.

We compared all three methods using the same problems as before where we varied the number of decompositions steps from 100 to 900 and the probability of alternative branching from 0.1 (10%) to 0.9 (90%). The runtimes (in milliseconds) of all algorithms are shown in Fig. 17; we again used a time limit of ten minutes to run each algorithm on particular data. Note that the graphs show only the time to generate the enhanced model (to find the implied constraints). The time to solve the P/A graph assignment problem is not included because it is negligible; it varied from 0 to few milliseconds for the nested model, from 0 to 200 milliseconds for the rules, and from 0 to 600 milliseconds for the singleton model (larger time is typically for the larger proportion of alternative branchings).

As we can see the nested method was the fastest among the three methods; only for problems with limited alternatives the method based on singleton consistency was faster (again, thanks to fast propagation of equality constraints). For the rule-based method and for the method based on singleton consistency we can identify the significant increase of runtime when going from problems with limited alternatives to problems with a lot of alternatives. There is only a small increase of runtime for the nested method. Again, the increase seems to be "step"-like, the runtime remains constant if we further increase the ratio between alternative and parallel branchings. Figure 18 presents 2D cuts of the 3D graphs from Fig. 17 providing an easier comparison of runtimes for the presented methods.

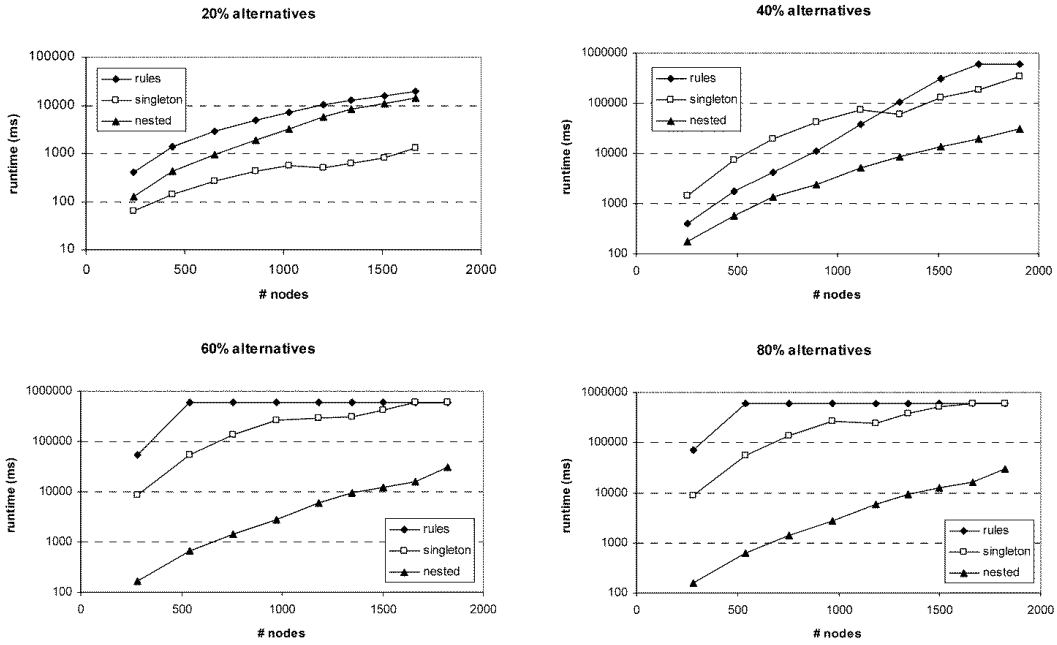


Fig. 18 Runtime to generate the enhanced model by the studied methods depending on the size of graphs (logarithmic scale)

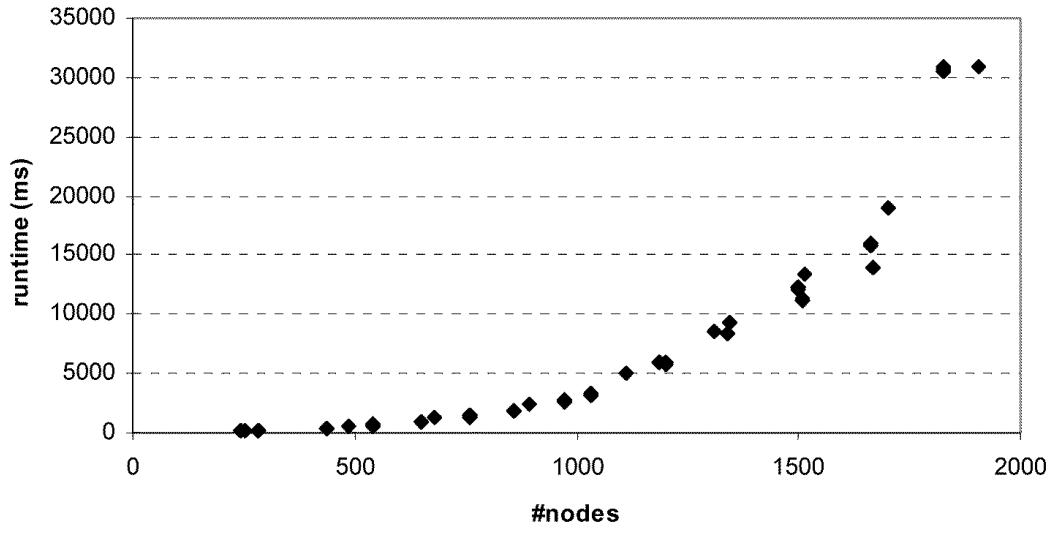


Fig. 19 Runtime of the nested detector for nested P/A graphs

Let us analyse the winning nested method further. Figure 19 shows the runtime to generate the enhanced constraint model (to identify all the nests in the P/A graph) as a function of the size of the P/A graph for all problems from the previous experiment. The graph corresponds to the quadratic complexity of the algorithm. Notice also that the runtimes are not scattered for a given size of the graph (the number of nodes) so the runtime is independent of the number of alternative branchings.

To evaluate the total efficiency of the nested method, we compared it with the basic model using the experiment where we fixed the probability of alternative branching to 80%,

Table 2 Solved P/A graph assignment problems for graphs with 80% of alternative branchings using the nested approach (the runtime includes the time to enhance the model as well as to solve it)

Decompositions	Solved	Backtracks	Runtime (ms)
200	100%	0	1007
400	100%	0	3039
600	100%	0	6809
800	100%	0	12308

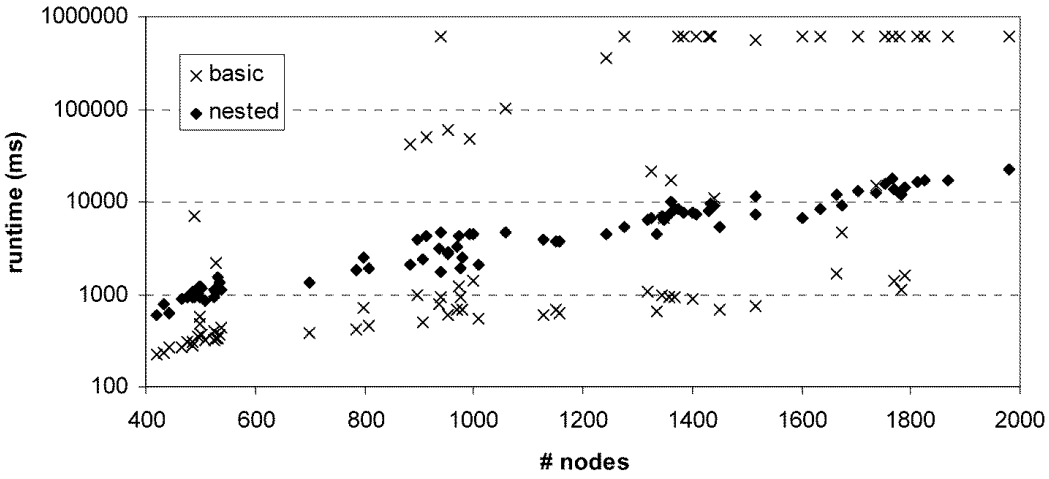


Fig. 20 Comparison of total runtimes for solving the nested P/A graph assignment problems using the nested method and the basic model (logarithmic scale)

we varied the number of decomposition steps from 200 to 800, and for each set of parameters we generated 20 problems. Table 2 shows the result; note that now the runtime includes both the runtime to enhance the constraint model plus the time to actually solve the problem. In comparison with Table 1, we can see that the nested method is much more stable than the basic constraint model—we can solve all the problems and the runtime is not significantly increasing with the increased size of the problem.

Finally, Fig. 20 directly compares the nested method with the basic constraint model. The graph shows a distribution of total runtime to solve the problem as a function of the problem size. The nested method is not always faster due to overheads with generating the enhanced constraint model, but it has some nice properties. Its runtime is stable, that is, the runtime is not scattered for a given problem size. Moreover, as the problem size enlarges, the nested method becomes faster for a larger portion of problems. This behavior is important because the real-life manufacturing scheduling problems are expected to be in the scale of thousands of operations. The stability of logical reasoning is also quite important especially, when temporal and resource constraints are added, where the reasoning is already known to be time consuming.

7 Summary

The paper studied three methods to improve constraint reasoning in P/A graphs modelling alternative processes. These improvements are based on the idea of identifying nodes in the graph whose validity status is identical in all feasible assignments—logically equivalent

nodes. We showed that the problems of finding a feasible assignment as well as identifying all logically equivalent nodes are NP-hard. Therefore the studied techniques are heuristic so we used empirical evaluation to compare their efficiency.

The rule-based method introduced in Barták et al. (2007) uses graph modification rules to find out logically equivalent nodes. It is a method specifically designed for P/A graphs but in some sense this method is similar to relational consistencies (Van Beek and Dechter 1994) or the resolution principle from theorem proving.

A general method using singleton consistency to find out logical equivalences and dependencies between Boolean variables was introduced in Barták (2007). In this paper, we applied this method to P/A graphs to find out logically equivalent nodes as well as implied dependencies between the nodes.

Finally, we modified the algorithm for detecting nested P/A graphs from Barták and Čepek (2007) to generate a constraint model that achieves global consistency for nested P/A graphs. This last method is the least general among the studied approaches because it generates specific constraints only for nested sub-graphs while for the rest of the P/A graph it uses the constraints of the basic constraint model.

We experimentally compared all three methods using randomly generated nested P/A graphs. The nested structure of P/A graphs is typical for many real-life problems as described in Horling et al. (1999) and Kim et al. (2001) so these experiments show an expected behaviour of the studied methods when applied to realistic problems. All methods proved themselves to significantly improve the basic constraint model. In particular, the models enhanced by any of the methods were solved without backtracks. However, there is an overhead to generate the enhanced models. The nested method was the most efficient one, which is not that surprising thanks to its specificity to nested graphs. Moreover, we used a straightforward implementation in Prolog so perhaps a more careful implementation may further increase its practical efficiency. It was surprising that the general method based on singleton consistency outperformed a more specific rule-based method. Similarity of the rule-based method to relational consistency and resolution principle may explain its poor performance, but this similarity requires further investigation. Finally, when we compared the runtimes of the basic model and the time to generate the enhanced models, it may seem that in most problems the basic model is still faster even if it requires search (Figs. 16 and 17). Nevertheless, the experiments showed "instability" of the basic constraint model (see Table 1 and Fig. 20) which means that there is a significant portion of problems for which the runtime is much higher. Opposite to this instability, the time complexity of the nested method is quadratic in the number of nodes and it generates constraint models achieving global consistency (Table 2). This advantage will be even more significant when temporal and resource constraints will be added to model complex scheduling problems, which is the topic of our future research.

Acknowledgements The research is supported by the Czech Science Foundation under the contract no. 201/07/0205 and by EU FP6 CRAFT Programme under the project No. 018071 EMPOSME. We would like thank the reviewers for valuable comments.

Appendix: Correctness of graph modification rules

We shall show now that the graph-modification rules introduced in Sect. 5.1 are correct, that is, the algorithm based on repeated application of rules returns groups of equivalent nodes. We can easily define a feasible assignment of 0/1 values to nodes of the hyper-graph. This definition should ensure that an assignment of 0/1 values to nodes of the original P/A graph

is feasible if and only if it is feasible for the corresponding hyper-graph (the correspondence between the P/A graph and the hyper-graph is given by the initial phase of the algorithm).

Let us denote $Q = \bigcup_{v \in V} Q_v$. An assignment $val : Q \to \{0, 1\}$ in hyper-graph $H = (U, F)$ is feasible if and only if $val(0) = 0$, $val(1) = 1$, for every edge $\{\{u\}, \{v\}\} \in F$ $val(u) = val(v)$, and for every non-trivial hyper-edge $\{\{x\}, Y\} \in F$ $\sum_{y \in Y} val(y) = val(x)$. Finally, the nodes in the same equivalence class associated with a given node have assigned the same value.

To prove the correctness of the algorithm it is sufficient to show that hyper-graph transformation rules preserve feasible assignments. The proofs are based on equivalence of arithmetic expressions describing the branching before and after applying the particular transformation rule.

Proposition 3 (Correctness of edge contraction) *An assignment $val : Q \to \{0, 1\}$ in the hyper-graph $H = (U, F)$ is feasible if and only if it is feasible for the hyper-graph after application of the edge contraction rule.*

Proof Let $\{\{u\}, \{v\}\} \in F$ be the contracted edge and $\{\{x\}, \{v\}\} \in F$, where $x \neq u$, be another edge (if any). Then feasibility for hyper-graph H enforces $val(x) = val(v)$ and $val(u) = val(v)$. Feasibility for hyper-graph after edge contraction enforces $val(x) = val(u)$ since $\{\{x\}, \{v\}\}$ is replaced by $\{\{x\}, \{u\}\}$ and $val(u) = val(v)$ since $v \in Q_u$. These two sets of formulas are clearly equivalent.

For hyper-edge $\{\{v\}, Y\} \in F$ feasibility for H enforces $\sum_{y \in Y} val(y) = val(v)$ and $val(u) = val(v)$. Feasibility for hyper-graph after edge contraction enforces $\sum_{y \in Y} val(y) = val(u)$ since $\{\{v\}, Y\}$ is replaced by $\{\{u\}, Y\}$ and $val(u) = val(v)$ since $v \in Q_u$. These two sets of formulas are equivalent.

Finally, for hyper-edge $\{\{x\}, Y\} \in F$, where $u \notin Y$, $v \in Y$ and $x \neq u$, feasibility for H enforces $\sum_{y \in Y} val(y) = val(x)$ and $val(u) = val(v)$. Feasibility for hyper-graph after edge contraction enforces $\sum_{y \in Y} val(y) - val(v) + val(u) = val(x)$ and $val(u) = val(v)$ since $v \in Q_u$. These two sets of formulas are again equivalent. □

Proposition 4 (Correctness of hyper-edge extension) *An assignment $val : Q \to \{0, 1\}$ in the hyper-graph $H = (U, F)$ is feasible if and only if it is feasible for the hyper-graph after application of the hyper-edge extension rule.*

Proof Let $\{\{x\}, Y\} \in F$ and $\{\{y\}, Z\} \in F$, where $y \in Y$ and $(\{x\} \cup Y) \cap (\{y\} \cup Z) = \{y\}$, be non-trivial hyper-edges that are selected to form a new hyper-edge. Feasibility for hyper-graph H enforces $\sum_{z \in Y} val(z) = val(x)$, $\sum_{z \in Z} val(z) = val(y)$, which implies $\sum_{z \in Y} val(z) + \sum_{z \in Z} val(z) - val(y) = val(x)$. This is exactly the constraint enforced by the new hyper-edge. □

Proposition 5 (Correctness of hyper-edge meet) *An assignment $val : Q \to \{0, 1\}$ in the hyper-graph $H = (U, F)$ is feasible if and only if it is feasible for the hyper-graph after application of the hyper-edge meet rule.*

Proof Let $\{\{x\}, Y\} \in F$ and $\{Z, \{w\}\} \in F$, where $x \neq w$ and $Z \subseteq Y$, be non-trivial hyper-edges on which the rule is applied. Feasibility for hyper-graph H enforces $\sum_{y \in Y} val(y) = val(x)$ and $\sum_{z \in Z} val(z) = val(w)$, which implies $\sum_{y \in Y} val(y) - \sum_{z \in Z} val(z) + val(w) = val(x)$. This is exactly the constraint enforced by the added hyper-edge. □

Proposition 6 (Correctness of never-valid activity detection) *An assignment* $val : Q \to \{0, 1\}$ *in the hyper-graph* $H = (U, F)$ *is feasible if and only if it is feasible for the hyper-graph after application of the never-valid activity detection rule.*

Proof Let us prove the case A of the rule first. Let $\{\{x\}, Y\} \in F$ and $\{\{y_1\}, \{y_2\}\} \in F$, where $\{y_1, y_2\} \subseteq Y$ be (hyper-)edges in H. The feasible assignment val in H must satisfy $val(y_1) = val(y_2) = 0$, since the remaining combinations of assignments of 0/1 values to y_1 and y_2 would violate the constraint $\sum_{y \in Y} val(y) = val(x)$. Hence, for any feasible assignment the constraint $\sum_{y \in Y} val(y) = val(x)$ holds if and only if $\sum_{y \in Y - \{y1, y2\}} val(y) = val(x)$ holds. Thus we did not change the set of feasible assignments for the hyper-graph by replacing the hyper-edge $\{\{x\}, Y\}$ by $\{\{x\}, Y - \{y_1, y_2\}\}$ and by adding new edges $\{\{0\}, \{y_1\}\}$ and $\{\{0\}, \{y_2\}\}$ that imply $val(y_1) = 0$ and $val(y_2) = 0$.

The proof of case B of the rule is similar. Let $\{\{x\}, Y\} \in F$ and $\{\{x\}, \{y\}\} \in F$, where $y \in Y$, be (hyper-)edges in H. Clearly, every feasible assignment val must satisfy $val(x) = val(y)$ and for all $z \in Y - \{y\}$ $val(z) = 0$. Any other assignment would violate the constraint $\sum_{y \in X} val(y) = val(x)$. Hence, addition of edges $\{\{0\}, \{z\}\}$ for all $z \in Y - \{y\}$ to F does not change the set of feasible assignments and the constraint on feasible assignments induced by the removed hyper-edge $\{\{x\}, Y\}$ is subsumed by constraints induced by newly added edges and by the edge $\{\{x\}, \{y\}\}$. □

References

Barták, R. (2007). Generating implied Boolean constraints via singleton consistency. In *LNAI: Vol. 4612. Abstraction, reformulation, and approximation (SARA 2007)* (pp. 50–64). New York: Springer.

Barták, R., & Čepek, O. (2007). Nested temporal networks with alternatives. In *Technical Report WS-07-12. Papers from the 2007 AAAI workshop on spatial and temporal reasoning* (pp. 1–8). Menlo Park: AAAI Press.

Barták, R., Čepek, O., & Surynek, P. (2007). Modelling alternatives in temporal networks. In *Proceedings of the 2007 IEEE symposium on computational intelligence in scheduling (CI-Sched 2007)* (pp. 129–136). New York: IEEE Press.

Beck, J. Ch., & Fox, M. S. (2000). Constraint-directed techniques for scheduling alternative activities. *Artificial Intelligence, 121*, 211–250.

Beeri, C., Fagin, R., Maier, D., & Yannakakis, M. (1983). On the desirability of acyclic database schemes. *Journal of the ACM, 30*, 479–513.

Bessiere, C., Coletta, R., & Petit, T. (2007) Learning implied global constraints, In *Proceedings of twentieth international conference on artificial intelligence (IJCAI 07)* (pp. 44–49).

Brélaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM, 22*, 251–256.

Charnley, J., Colton, S., & Miguel, I. (2006) Automatic generation of implied constraints, In *Proceedings of 17th European conference on artificial intelligence (ECAI 06)* (pp. 73–77).

Crowston, W., & Thompson, G. L. (1967). Decision CPM: A method for simultaneous planning, scheduling, and control of projects. *Operations Research, 15*, 407–426.

Dechter, R. (2003). *Constraint processing*. Los Altos: Kaufmann.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence, 49*, 61–95.

Erol, K., Nau, D., & Hendler, J. (1994). HTN planning: Complexity and expressivity. In *Proceedings of AAAI-94* (pp. 1123–1128). Menlo Park: AAAI Press.

Focacci, F., Laborie, P., & Nuijten, W. (2000) Solving scheduling problems with setup times and alternative resources, In *Proceedings of AIPS 2000*.

Frisch, A. M., Miguel, I., & Walsh, T. (2001) Generating implied constraints via proof planning, In *Proceedings of the IJCAR-01 workshop on future directions in automated reasoning* (pp. 48–55).

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. San Francisco: Freeman.

Horling, B., Leader, V., Vincent, R., Wagner, T., Raja, A., Zhang, S., Decker, K., & Harvey, A. (1999) *The taems white paper*. University of Massachusetts, http://mas.cs.umass.edu/research/taems/white/taemswhite.pdf.

Kim, P., Williams, B., & Abrahmson, M. (2001) Executing reactive, model-based programs through graph-based temporal planning, In *Proceedings of IJCAI-2001* (pp. 487–493).

Kuster, J., Jannach, D., & Friedrich, G. (2007) Handling alternative activities in resource-constrained project scheduling problems, In *Proceedings of twentieth international joint conference on artificial intelligence (IJCAI-07)* (pp. 1960–1965).

Nuijten, W., Bousonville, T., Focacci, F., Godard, D., & Le Pape, C. (2003) MaScLib: Problem description and test bed design. http://www2.ilog.com/masclib.

Tsamardinos, I., Vidal, T., & Pollack, M. E. (2003). CTP: A new constraint-based formalism for conditional temporal planning. *Constraints*, *8*(4), 365–388.

Van Beek, P., & Dechter, R. (1994) Constraint tightness versus global consistency, In *Proceedings of knowledge representation (KR-94)* (pp. 572–582).