A PROLOG TECHNIQUE OF IMPLEMENTING SEARCH OF A/O GRAPHS WITH CONSTRAINTS

Mária Bieliková
Pavol Návrat

Slovak Technical University, Dept. of Computer Science and Engineering,
Ilkovičova 3, 812 19 Bratislava, Slovakia
E-mail: {bielikova,navrat}@elf.stuba.sk
Tel.: (+ 42 7) 791 395
Fax: (+ 42 7) 720 415

Abstract. Our research has been motivated by the task of forming a solution subgraph which satisfies given constraints. The problem is represented by an A/O graph. Our approach is to apply a suitably modified technique of dependency-directed backtracking. We present our formulation of the standard chronological backtracking algorithm in Prolog. Based on it, we have developed an enhanced algorithm which makes use of special heuristic knowledge. It involves also the technique of node marking. We have gathered experience with the prototype Prolog implementation of the algorithm in applying it to (one step of) the problem of building a software configuration. Our experience shows that Prolog programming techniques offer a considerable flexibility in implementing the above outlined tasks.

Keywords. A/O-graph, non-chronological backtrack, Prolog

1 PROBLEM AREA AND GOAL

Many problems to which artificial intelligence techniques are often applied can be described as constraint satisfaction problems. We concentrate ourselves on a problem how to find a connected subgraph of an A/O-graph which is consistent, i.e. such that all the nodes in the connected subgraph satisfy the consistency requirements (constraints). It is assumed that an A/O-graph has one or more roots.

Motivation for our work originates in a research in software system configuration building. We have devised a new method for building a configuration [5, 16].

One of the crucial steps in our method is searching for a so called generic configuration in a model of software system represented as an A/O graph [3]. However, we have formulated our new search algorithm more generally. The main aim of this paper is to present programming techniques that have proven useful in implementing such algorithms.

Let us introduce some graph terminology first.
Let AN and ON be disjoint sets, i.e. $AN \cap ON = \emptyset$. Let N be their union, i.e. $N = AN \cup ON$. Let E be a binary relation $E \subseteq N \times N$. An oriented graph $G = (N, E)$ with the set of nodes N and set of arcs E is called an A/O-graph.

Nodes are called A-nodes (nodes from the set AN) or O-nodes (nodes from the set ON) according to the way sets of their successors are interpreted (logical AND or logical OR, respectively).

Next, we define a solution graph. Let $G = (N, E)$ be an A/O-graph with a root s_G. We call graph $SG_G = (P, H)$, with $P \subseteq N, H \subseteq E$ a solution graph of the graph G, if the following holds:

$s_G \in P \wedge$

$\forall n_1((n_1 \in P \wedge (n_1 \text{ is } O\text{-node })) \Rightarrow \exists! n_2(n_2 \in P \wedge (n_1, n_2) \in H)) \wedge$

$\forall n_1((n_1 \in P \wedge (n_1 \text{ is } A\text{-node })) \Rightarrow \forall n_2((n_2 \in N \wedge (n_1, n_2) \in E) \Rightarrow$
$(n_2 \in P \wedge (n_1, n_2) \in H))) \wedge$

$\forall n(n \in P \Rightarrow s_G H^+ n).$

In other words, a solution graph SG_G of the graph G is such its subgraph that the root of G belongs to the solution graph, if n is an O-node in G and n is in SG_G then exactly one of the successors belongs to SG_G, if n is an A-node in G and n is in SG_G then all its successors belong to SG_G, and there are no other nodes in SG_G than the root of G and its successors in SG_G.

It would be easy to generalize the notion of the solution graph for graphs having more than one root, but we shall consider for the sake of simplicity only the case of one root graphs.

The problem of forming a consistent solution graph is determined by the A/O graph G itself (which is assumed to be given explicitly) and by a set of constraints which specify acceptable combinations of nodes in solution graphs. Solution to the problem is a solution graph SG_G such that all its nodes satisfy the given set of constraints. We shall assume the set of constraints is finite and consists solely of constraints referring to at most two nodes. The set of constraints can thus be represented as a set of pairs of nodes. There are various approaches to representing and handling constraint satisfaction problems known in literature [13].

A pair of nodes satisfying the constraints is said to be consistent. A pair of nodes which does not satisfy at least one of the constraints is said to be inconsistent. When both of them are included in a potential solution graph they cause a violation to occur (i.e., an inconsistency of the potential solution graph).

Some nodes of an A/O graph can be inconsistent with the graph G as a whole.

A node n is inconsistent with a graph G for the given set of constraints if either n is inconsistent with the root of G or n is inconsistent with all of its immediate successors or predecessors. Any such node can never be included in a consistent solution graph because it would cause a violation.

2 RELATED WORK

A constraint satisfaction problem (CSP) consists of a set of variables, a finite and discrete value domain for each variable, and a set of constraints. For binary constraints, the constraint graph has a node representing each variable, and an edge connecting each pair of nodes representing variables on which a binary constraint is defined. In [1], an optimal algorithm for solving tree structured constraint satisfaction problems is presented and evaluated. The algorithm, which solves the problem in linear time, is an improvement over earlier solution presented by [7]. In [7], an enhancement of chronological backtrack is presented; however, extensive preprocessing on the problem input in order to reduce the search space was considered a drawback. The enhancement itself was based on a form of backjumping and shallow forms of learning. The approach taken in [1] tackles the problem of avoiding extensive preprocessing steps before attempting solution. It introduces an enhancement to chronological backtrack by recording the "good" domain values encountered during backtrack. It allows skipping ahead to avoid solving the same subproblem multiple times. Another work [19] concentrates on modelling a search space that captures principal effects of searching algorithms by exploiting the deep structure of constraint problems. An earlier work [10] employs a method for avoiding redundant and irrelevant computation, based on caching past successes and reusing them later. Similarly, impossibilities are cached and further exploration of inferences dependent upon them is prohibited.

All the algorithms make use of some kind of backtracking in one way or another. Backtrack searching is one of the most commonly used techniques in artificial intelligence. The basic algorithm searches the state space in some systematic way (e.g., depth first) until it either finds a solution or it reaches a state which is not a solution but at the same time it is not possible to go on in search from that node due to a violation of constraints (the situation called a deadend). When a deadend is reached, the searching backtracks to the previous point of choice, selects an unexplored alternative and continues in searching until either a solution is found or the fact is established that no solution exists (so called chronological backtrack) [17].

An increase in efficiency of the backtrack search can be achieved by reducing the currently searched state space. Besides the constraints, the size of the currently searched space depends on decisions made during the searching concerning the order in which the states are visited. All these influencing factors can be applied either before the actual searching, or during it. The static techniques include algorithms which transform the problem into an equivalent one, however with more explicit representation of constraints [8].

The dynamic techniques can roughly be divided into prospective (lookahead) and retrospective (lookback) ones. Prospective strategies select the next state to be visited after analyzing yet unexplored parts of the search space [11]. Retrospective strategies rely on analysis of the already explored part of the search space when selecting next state to be visited. They try to record as much information as suitable on the progress of searching, attempting to avoid redundant testing of consistency. There are known approaches which concentrate on the question how far back should the search go in case of a deadend. They try to analyze reasons for an occurence of a deadend [9, 8].

Of special relevance to the problem tackled in this paper are works dealing with the problem of searching A/O graphs. In [14], search of A/O graphs is studied. The graphs are given implicitly by a set of state transformation rules. On the other hand, in [6] algorithms are presented to find out the optimal cost solutions below a node in an explicit A/O graph.

Regarding the complexity of constraint satisfaction tasks, in general they belong to the class of NP-complete problems and, as such, normally lack realistic measures of performance. Worst-case analysis depends on extreme cases and therefore may not yield a correct view of typical performance of algorithms. Average-case analysis is extremely difficult and is highly sensitive to simplifying theoretical assumptions [7]. As the authors state, theoretical analysis must be supplemented by experimental studies. In their study, main results show that the average complexity of backtracking on their randomly generated problems is far from exponential.

Based on similar considerations, we have adopted the experimental approach to evaluating properties of our algorithm. We shall also employ randomly generated problems.

3 THE BASIC ALGORITHM FOR (CHRONOLOGICAL) BACKTRACK SEARCHING

Let us present our formulation of the known algorithm for chronological backtrack searching of A/O graphs. We formulate it as a logical program. Algorithm is enhanced by considering constraints on acceptable combinations of nodes in solution graphs. We include the formulation because later we attempt to enhance the algorithm by incorporating special knowledge on searching, and we present our new algorithm as a modification of the basic one. But the formulation itself may also be of interest as it documents a specific Prolog programming technique for implementing such a kind of algorithms.

Basic operation in graph searching is expanding of a not yet visited leaf of the potential solution graph. For a node being expanded, there could be three cases:

1. confirming success in searching the leaf node of the A/O graph or a node with all its successors explored; there is no need to visit any successors

2. the node is consistent with the potential solution graph but has some of its successors unexplored
3. the node violates some constraint (it is inconsistent).

In the third case, backtrack must be used to find a solution.

It is important to realise that all the nodes of the searched A/O graph can in any moment be in one of the following mutually disjunctive sets:

- nodes which have all their successors explored,
- nodes which have their predecessors explored, but not their successors,
- nodes which have neither their predecessors nor their successors explored.

In describing the algorithm for searching an A/O graph, a distinction between the nodes from the first and second set must be made. We shall use lists $OPEN$ and $CLOSED$ to keep record of the respective sets. In any moment during searching, nodes on $OPEN$ and $CLOSED$ are those forming a potential solution graph. With each processed node a pointer to its predecessor is recorded to allow for easy reconstruction of the solution graph. Elements on $OPEN$ and $CLOSED$ are represented as structures N/P, where N represents a node and P represents a set of its immediate predecessors in a potential solution graph.

The algorithm finds one solution graph for an A/O graph if it exists. It assumes that the A/O graph being searched is represented by clauses (facts `node(Node,Kind)` and `arc(Node1,Node2)`).

The algorithm is defined in Prolog by two predicates *search_forward* and *search_back*. The former widens the potential solution graph and the latter takes care of the situation when a violation (in deadend) is detected. The predicates maintain for each O-node N a list of possible candidates for successors of that node. The list is denoted by S_N. Searching is initiated by a goal

```
search_forward(OPEN, CLOSED, SOLUTION)
```

where the list $OPEN$ contains the root of the A/O graph and the list $CLOSED$ is empty. When the graph has more roots, all of them are included in $OPEN$.

The predicate *search_back* searches for a node in the potential solution graph such that there is an yet unexplored alternative of successor selection for it. The predicate either finds such a node, in which case the searching goes on always from that node by *search_forward*, or it fails to find one, in which case the overall failure is reported (i.e., a consistent solution graph could not be found).

```
/******* search_forward(OPEN, CLOSED, SOLUTION) *******/
      % 1. if list OPEN is empty
      %        then halt, reconstruct the solution graph from CLOSED
search_forward([], SOLUTION, SOLUTION) :-
   write(SOLUTION).

      % 2. N/P ← an element selected from OPEN
      % 3. if N/P_c is on CLOSED
      %        then N/P_c ← N/(P_c ∪ P), i.e. add a pointer to node N and
      %             search_forward
search_forward([N/P | OPEN], CLOSED, SOLUTION) :-
   member(N/P, CLOSED),
   add_pointer(N/P, CLOSED, New_CLOSED),
   search_forward(OPEN, New_CLOSED, SOLUTION).

      % 4. if N does violate consistency of the potential solution graph
      %        then search_back
search_forward([N/P | OPEN], CLOSED, SOLUTION) :-
   not(consistent(N, CLOSED)), !,
   search_back([N/P | OPEN], CLOSED, SOLUTION).

      % 5. if N is a leaf node
      %        then remove N/P from OPEN and
      %             put N/P on CLOSED and
      %             search_forward
search_forward([N/P | OPEN, CLOSED, SOLUTION) :-
   leaf_node(N),
   search_forward(OPEN, [N/P | CLOSED], SOLUTION).

      % 6. if N is an O-node
      %        then S_N ← list of successors of node N
      %    if the list S_N is not empty
      %        then N_succ ← an element selected from the list S_N and
      %             remove N_succ from the list S_N and
      %             put N_succ/{N} on OPEN and
      %             remove N/P from OPEN and
      %             put N/P on CLOSED and
      %             search_forward
      %        else search_back
search_forward([N/P | OPEN], CLOSED, SOLUTION) :-
   o_node(N),
   successors(N, Succ),
   ( member(N_succ, Succ),
     delete(N_succ, Succ, New_Succ),
     insert_successors(N, New_Succ),
     search_forward([N_succ/N | OPEN], [N/P | CLOSED], SOLUTION)
     ;
     search_back([N/P | OPEN], CLOSED, SOLUTION)
   ).

      % 7. if N is an A-node
      %        then for each its successor N_succ:
      %                 put N_succ/{N} on OPEN
      %             remove N/P from OPEN and
      %             put N/P on CLOSED and
      %             search_forward
```

```
search_forward([N/P | OPEN], CLOSED, SOLUTION) :-
   a_node(N),
   findall(Succ/P, arc(N, Succ), List_N_succ),
   append(List_N_succ, OPEN, New_OPEN),
   search_forward(New_OPEN, [N/P | CLOSED], SOLUTION).


/******* search_back(OPEN, CLOSED, SOLUTION) *******/

      % if element N/P is on CLOSED and
      %       there exists a nonempty list S_N for node N
      %   then N_succ <- an element selected from S_N and
      %       remove N_succ from the list S_N and
      %       remove from the potential solution graph the subgraph with the root N and
      %       put N_succ/{N} on OPEN and
      %       search_forward
search_back(OPEN, CLOSED, SOLUTION) :-
   member(N/_, CLOSED),
   show_successors(N, Succ),
   member(N_succ, Succ),              % there exists an nonempty list S_N
   delete(N_Succ, Succ, New_Succ),
   insert_successors(N, New_Succ), !,
   delete_subgraph(N, [OPEN, CLOSED], [New_OPEN, New_CLOSED]),
   search_forward([N_succ/N | New_OPEN], New_CLOSED, SOLUTION).

      %   else halt, no solution
search_back(_,_,_) :-
   write('NO SOLUTION'),
   !, fail.
```

The predicate $consistent(N, List)$ expects $List$ to be a list of nodes representing the potential solution graph, along with pointers to predecessors. It is satisfied when the node N does not violate any constraint after having been added to $List$. The predicate $successors(N, List)$ with an input parameter N and an output parameter $List$ represents the relation between the O-node N and its alternative successors.

The predicates $insert\_successors(N, Succ)$ and $show\_successors(N, Succ)$ are used to maintain and reference lists of possible candidates for successors of N, respectively.

The predicate $delete\_subgraph(N, State, New\_State)$ has an output parameter New_State. It will contain a modified $State$ (i.e., a pair $[OPEN, CLOSED]$) after the subgraph rooted in N has been rejected.

4 THE ENHANCED ALGORITHM

Our intention has been to enhance the basic algorithm for backtrack searching by using problem specific knowledge to make the next node selection more efficient. Our aim is to reduce the number of visited nodes of the A/O graph as well as the number of consistency checks. The selection points follow directly from the above formulation of the basic algorithm:

1. selecting a node from the list $OPEN$ for processing (in the basic algorithm, the most recently inserted node is always selected),
2. selecting the successor of an O-node from among the set of its successors during $search\_forward$ or $search\_back$ (in the basic algorithm, selection is based on their order),
3. selecting a node for which we attempt to explore the yet not visited alternatives (in the basic algorithm, there is selected always the node that was most recently inserted in $CLOSED$).

By making "correct" decisions in the above described points, we can make the searching more efficient. We propose that in selecting the node from the list $OPEN$, it is advantageous to use heuristics preferring nodes which will most likely cause rejecting the hypothesis that the current subgraph is part of the solution graph. Our assumption is that if the potential solution graph being formed is not a part of the solution, the sooner this fact is established, the better. The heuristics can be problem specific, but they can also express knowledge related to the structure of the graph, e.g. *from among several nodes, prefer those having more pointers to their predecessors.* The above heuristic originates from a consideration that if a node which has more pointers to its predecessors is going to come out as the one causing the violation then this fact should be discovered as soon as possible. We take into account that when a node has more pointers to its predecessors then there are more predecessors demanding its adding to the solution graph. As an example of another heuristic, we have *prefer the node which is involved in the greatest number of constraints.* Similar approach was taken by Freuder and Wallace [11].

Decisions in points according to the second alternative can also have influence on efficiency of searching, and more specifically on finding the first solution. For example, if there exists a solution subgraph of the A/O graph and there is always selected the "correct" successor to an O-node then the solution can be found entirely without any backtrack. As an example of a heuristic, we have *prefer the successor of an O-node which constrains the future choices in the least possible way.* Specific case of this heuristic can be: *prefer the successor of an O-node which is involved in the least number of constraints.* In this case the heuristic is less expensive for computation. As an example of another heuristic, we have *prefer the successor of an O-node which has the least number of successors.* It is based on a consideration that if a node which has less successors is selected the overall number of nodes in consideration is smaller with the consequence that a risk of deadends is lower.
Decisions on selecting successors to O-nodes depend frequently on the particular problem, and consequently on our knowledge about the problem represented by the A/O graph.

Modifying the way decisions are made in points according to alternative 3. opens room for backtracking over several levels in case of deadends (so called backjumping), instead of chronological backtracks. We propose to use problem related knowledge and to identify possible reasons for arriving at deadend.

When a deadend is recognised during constrained searching of the A/O graph, there could occur two different situations:

1. the node being processed is inconsistent with the A/O graph as a whole; we call the node to be **impossible**, or
2. the node being processed is inconsistent with the current potential solution graph; however, it can still be consistent with some other potential solution graph; we call the node to be **conflicting**.

To be able to make the backtracking algorithm more efficient, we should identify when the above described situations can occur and we should define appropriate procedure, i.e. where the alternative nodes should be sought in the graph.

4.1 Handling the case of an impossible node

Impossible nodes are identified during a *search_forward* on two occasions. First, when the node N being currently processed is being checked for consistency with the current potential solution graph. N is found impossible when it is not consistent with the root of the graph. Second, during expanding the node N, when (in case of O-node) it does not have any consistent successor, or (in case of A-node) it has at least one inconsistent successor. Another possibility is when all of its predecessors are inconsistent.

It is clear that nodes which are found to be impossible can be removed at the outset, thus reducing the search space before any searching is done. Sometimes however, checking constraints is a very expensive operation. It this case, better solution is not to preprocess graph at the start if it can be avoided altogether.

4.2 Handling the case of a conflicting node

When a decision is made on whether to add a node to the potential solution graph, all the constraints defined for the problem must be considered. However, in each step there can be separated a subset of constraints such that it includes only constraints referring to nodes from the potential solution graph. Let us call such subset of constraints a *current set of constraints*. When the node N is considered to be added to the potential solution graph, following violations can occur:

- conflicting constraints within the current set of constraints corresponding to a new potential solution graph,

- violations between the new current set of constraints (after the node N has been added) and the solution state, i.e. the subgraph formed until that moment,
- violations between the current set of constraints (before the node N has been added) and the node N.

The violations can be resolved in one of the two ways:

1. an attempt is made to add the node N to the potential solution graph, or
2. adding the node N to the potential solution graph is rejected.

Considering both of the ways secures completeness of the method. We can make use of specific knowledge in each case. In the first case, the violation could be resolved by rejecting inconsistent parts of the potential solution graph, i.e. those which are inconsistent with the node N. This procedure is appropriate only if as a consequence of the rejection, the node N will not be rejected as well, i.e. there will not be rejected any subgraph which includes N during the process of rejecting incosistent parts and subsequent searching of alternatives from them. When this cannot be secured, we have the second case, and the procedure is analogous to the case of arriving at a deadend as a consequence of recognising an impossible node, i.e. an alternative solution must be sought at the predecessors of the node N.

The above procedure of rejecting violations is essentially the technique of truth maintenance [9] well known in artificial intelligence. By rejecting the inconsistent part we get a new potential solution graph in which only consistent parts remain, i.e. we have a justified backtrack.

Which are the nodes that should be rejected? They are the minimal set of inconsistent nodes with the node N in the given state, reduced by all the nodes which have predecessors in the same set.

The mininal set of inconsistent nodes Min_IN with N in state $[OPEN, CLOSED]$ is a set of all nodes N' such that:

1. N' is on $CLOSED$,
2. state $[OPEN, (CLOSED \setminus Min\_IN) \cup \{N\}]$ is consistent,
3. there does not exist a node U from Min_IN such that state $[OPEN, (CLOSED \setminus Min\_IN) \cup \{U, N\}]$ is consistent.

4.3 Applying the technique of node marking

Another advantage in an effort to improve the backtracking search can be sought in appropriate marking of nodes. We see as an advantage when a visit to nodes and a subsequent checking for consistency with the potential solution graph can be avoided altogether. This is true for nodes which in the current state do not let themselves to be added to the potential solution graph.

As could be already seen, when in deadend node N is processed it is either $impossible$, i.e. inconsistent with any solution graph, or $conflicting$, i.e. inconsistent with the current potential solution graph.

The property of inconsistency of N with a graph as a whole is permanent and does not change in the course of searching. It is a unary property, i.e. an attribute, and we shall record it by the mark $impossible$ attached to the node N. As a consequence of the definition of the solution graph, there can be at the same time identified as $impossible$ also some of the predecessors of N according to following rules:

> An O-node shall be marked $impossible$ if all of its successors are marked $impossible$.
>
> An A-node shall be marked $impossible$ if at least one of its successors is marked $impossible$.

Therefore it makes sense to search only nodes which are not marked $impossible$. The predicate $successors(N, Succ)$ used in $search\_forward$ is modified in such a way that the list $Succ$ shall contain only "possible" alternatives of further searching from N. Successors marked $impossible$ are not considered.

Similarly, we introduce marking for nodes which cannot in the current state be included in the potential solution graph. Let us mark such nodes as $conflicting$. The situation with their predecessors is different. Predecessors of a conflicting node can still be suitable i.e. consistent with respect to the current state. Therefore, they cannot be marked as $conflicting$. We mark such nodes as $unsuitable$.

The process of marking predecessors is analogous to that of marking $impossible$ nodes. This time, mark $unsuitable$ is used. Different marking ($conflicting$ vs. $unsuitable$) is justified because we want to have only those nodes marked $conflicting$ which do not satisfy the current constraint set in the current state.
Summing up, nodes shall be marked $unsuitable$ according to the following rules:

> An O-node which has all the successors marked as either $impossible$, $conflicting$, or $unsuitable$, with at least one of them marked as $conflicting$ or $unsuitable$ shall be marked $unsuitable$. Note that it cannot be marked $impossible$.
>
> An A-node which has at least one successor marked $conflicting$ or $unsuitable$, but not even one successor marked $impossible$ shall be marked $unsuitable$. Note that it cannot be marked $impossible$.

The mark attributed to a node expresses the reason why it cannot, at least for the time being, be added to the potential solution graph:

- *impossible*: the node cannot be added to any solution graph at all.
- *conflicting*: the node does not satisfy the condition of consistency (the current set of constraints). It is possible that when the current state is changed it could be added to the potential solution graph.
- *unsuitable*: the node cannot be added to the potential solution graph just because some of its successors do not suit.

Due to the fact that the state of the solution may change, and consequently also the current set of constraints may change, it is necessary to maintain the consistency of markings of nodes with *unsuitable* and *conflicting* marks.

Now we present our new backjumping algorithm enhanced with marking of nodes with *impossible*, *conflicting*, and *unsuitable* marks. Comparing with the basic algorithm as formulated above, the modification concerns essentially the part of the algorithm which handles the deadend situation, i.e. the predicate *search_back*. We present only this part of the new algorithm.

The predicate *search_forward* is to be modified so that it identifies the state when deadend occurs as a consequence of processing an *impossible* node. The node itself as well as its predecessors according to the above rules are to be marked *impossible* or *unsuitable*. In this case, the algorithm backtracks as implemented by the predicate *back_parents*. In all other cases, it backtracks as implemented by the predicate *back_conflicting* which is also responsible for marking nodes as *conflicting* and *unsuitable*.

```
/******* back_parents(OPEN, CLOSED, SOLUTION) *******/

      % 1. Let N be a node processed when deadend occured
      %    PRED ← list of all nodes N' such that:
      %     a) N' is an immediate predecessor of N,
      %     b) N' is on CLOSED,
      %     c) there does not exist a node N'' such that
      %           it is on PRED and at the same time there exists a path from N'' to N',
      %     d) there does not exist any node in the A/O graph such that
      %           it satisfies the conditions a), b), and c) and is not on PRED
      % 2. if PRED is not empty
      %         then find a yet unexplored suitable alternative for all elements on PRED and
      %              search_forward
back_parents([N/_ | OPEN], CLOSED, SOLUTION) :-
   predecessors(N, CLOSED, PRED),
   PRED \= [],
   alternative_parents(PRED, [OPEN, CLOSED], [New_OPEN, New_CLOSED]),
   search_forward(New_OPEN, New_CLOSED, SOLUTION).

      %         else halt, no solution
back_parents(_, _, _) :-
   write('NO SOLUTION'), !, fail.

      % for each element N on PRED find a yet unexpored suitable alternative
alternative_parents([], State, State).
```

```
alternative_parents([N | Rest], State, New_State) :-
   alternative(N, State, State1),
   alternative_parents(Rest, State1, New_State).
```

% if N is marked as either *impossible*, *conflicting*, or *unsuitable*
% then find a yet unexplored suitable alternative for predecessors of node N
% acquired as stated in 1

```
alternative(N, State, New_State) :-
   not(suitable(N)), !,          % mark impossible, conflicting, or unsuitable
   predecessors(N, State, PRED),
   PRED \= [],
   alternative_parents(PRED, State, New_State).
```

% if there exists a nonmarked element on S_N
% then remove from the potential solution graph the subgraph rooted in N and
% $N\_back \leftarrow$ a nonmarked element from S_N and
% remove N_back from S_N and
% put $N\_back/\{N\}$ on $OPEN$

```
alternative(N, State, [[N_back/N | OPEN], CLOSED]) :-
   show_successors(N, Succ),
   member(N_back, Succ),          % list S_N is not empty
   suitable(N_back), !,
   remove_subgraph(N, State, [OPEN, CLOSED]),
   remove(N_back, Succ, New_Succ),
   insert_successors(N, New_NewSucc).
```

% else find a yet unexplored suitable alternative for predecessors of node N
% acquired as stated in 1

```
alternative(N, State, New_State):-
   predecessors(N, State, PRED), PRED \= [],
   alternative_parents(PRED, State, New_State).

/******* back_conflicting(OPEN, CLOSED, SOLUTION) *******/
```

% 1. Let N be a node processed when deadend occured
% if N is not consistent with any of its immediate successors
% then mark it as *impossible* (and propagate the marking to predecessors)
% *back_parents*

```
back_conflicting([N/P | OPEN], CLOSED, SOLUTION) :-
   successors(N, Succ),
   not(consistent(N, Succ)), !,
   mark(impossible, N),
   back_parents([N/P | OPEN], CLOSED, SOLUTION).
```

% 2. $CON \leftarrow$ list of all nodes N' such that
% (assuming current state is described by $[OPEN, CLOSED]$):
% a) N' is on $CLOSED$
% b) state $[OPEN, (CLOSED \setminus CON) \cup \{N\}]$ is consistent
% c) there does not exist a node $U \in CON$ such that
% state $[OPEN, (CLOSED \setminus CON) \cup \{U, N\}]$ is consistent
% 3. if CON is not empty
% then find an alternative solution for all nodes on CON and
% *search_forward*

```
back_conflicting([N/_ | OPEN], CLOSED, SOLUTION) :-
   conflicting(N, CLOSED, CON), CON \= [],
   alternative_conflicting(CON, N, [OPEN, CLOSED], [New_OPEN, New_CLOSED]),
```

```
   search_forward(New_OPEN, New_CLOSED, SOLUTION).

      %       else back_parents(N)
back_conflicting(OPEN, CLOSED, SOLUTION):-
   back_parents(OPEN, CLOSED, SOLUTION).

      % for each element N_con on CON find a suitable not yet used alternative
      %       provided N is in the current state
alternative_conflicting([], _, State, State).
alternative_conflicting([N_con | Rest], N, State, New_State) :-
   mark(conflicting, N_con, State, State1),
   alternative(N_con, State1, State2),
   is_in_current(N, State2),
   alternative_conflicting(Rest, N, State2, New_State).
```

The predicate $mark(Mark, N, State, New\_State)$ assigns the $Mark$ ($impossible$ or $conflicting$) to the node N. It propagates the marking to predecessors according to rules described above. It modifies the current state by deleting subgraphs rooted in all impossible or conflicting nodes. It restores consistency of marking by $conflicting$ and $unsuitable$ marks if necessary.

The predicate $delete\_subgraph(N, State, New\_State)$ is enhanced by maintaining the consistency of marking by $conflicting$ and $unsuitable$ marks. This is necessary to do whenever some parts of the potential solution graph are rejected. In such cases, the current constraint set could be modified. As a consequence, some previously conflicting or unsuitable nodes may have become admissible for inclusion in the potential solution graph.

5 APPLICATION IN BUILDING A SOFTWARE CONFIGURATION

The above presented new method for searching a solution of a problem described as an A/O graph can be applied to a range of problems for which a solution is a subgraph satisfying given constraints. Nevertheless, it is to be noted that our special concern has been design and implementation of a method for building a software configuration. We have devised such a method and report on it elsewhere [2]. Here, we give only a flavour of its conceptual framework.

We describe a software system by a model in form of an A/O graph. This allows us to represent versions of software components and relations between them, as they come into existence and undergo modification during the system development and maintenance. Subsets of software components which are "close" enough form families of versions. They are in that sense equivalent. When building a configuration, first a decision must be made regarding which families should be represented in it. Due to the nature of the process of forming software versions, sometimes a special kind of versions is created to implement an alternative concept. These versions are called variants and are defined as equivalence classes within respective families. As a consequence, a variant must be selected as the next decision for each selected family. In turn, variants refer to software families which they depend on, or from which they import, etc. We find it necessary to represent families and variants as nodes in our model. Families are parents to variants, acting as O-nodes. Variants are parents to families they need, acting as A-nodes. There is another kind of versions, so called revisions, which is not repesented explicitly in our model.

As an example, let us consider an A/O-graph (see Figure 1), that depicts a software system with families $EVAL$, $INIT$, $ACTIONS$, $INFER$, $MANAGER$ and $INTERR$ and variants $EVAL.1$, $EVAL.2$, $EVAL.3$, $INIT.1$, etc.

Fig. 1. Software system modelled by an A/O graph.

Let us note that the example presented in Figure 1 is a small one mainly due to understandibility. Problems we are solving are much bigger than this, so that finding a feasible solution is extremely time consuming.

It is assumed that from the model and configuration requirements a generic configuration is formed, i.e. a subgraph satisfying the requirements. Only then we proceed to select a revision satisfying requirements for each variant in the generic configuration. This results finally in a bound configuration.

Forming the generic configuration is the crucial phase of our method for building software system configuration. The problem is how to find a consistent subgraph (the solution graph) of the A/O graph (the model) such that all its nodes satisfy the requirements constraining the whole configuration as well as requirements constraining particular nodes. We assume that the set of constraints is finite. Consistency of software components included in the configuration can be constrained in two ways:

1. by a predicate over attributes of one node,
2. by a predicate over attributes of a set of nodes.

As an example of the former, the condition

$$\forall x(x \in K \Rightarrow (problem\_type(x) = design \wedge operating\_system(x) = DOS))$$

where K is a set of components in the configuration, constrains selection of components only to those which have the value *design* of the attribute *problem_type*, and the value DOS of the attribute *operating_system*.

As an example of the latter, the condition

$$\forall x, y((x \in K \wedge y \in K) \Rightarrow (not(language(x) = Prolog \wedge language(y) = C)))$$

constrains selection of components only to such that there will not be in the set K components which have the value *Prolog* of the attribute *language* and also other components which have the value C of the same attribute.

In the former case, constraint checking can be done independently from the current solution graph. In the latter case, constraint checking involves potentially all the nodes in the current solution graph.

A former type constraint referring to a node N can always be rewritten in a set of binary constraints with N being their first element. For a latter type constraint, there can be found so called *maximal consistent sets of graph nodes*, i.e. sets which

– contain nodes satisfying the given constraint, and

– cannot be enlarged by any node unless they become inconsistent.

Such sets can easily be transformed into sets of binary constraints by forming all the combinations.

From the above it can be seen that in our method of building a software configuration, generic configurations can be produced by applying the dependency-directed backtracking algorithm being presented in the paper. The constraints are to be preprocessed in the way outlined above. A condition constraining the whole configuration will be related to the root of the A/O graph.

The constraints do not have to be preprocessed into the binary form in case of reimplementing the operations of

- consistency checking of a node with respect to a subgraph (predicate *consistent/2*),
- consistency checking of a node with respect to the whole graph (predicate *impossible/2*),
- maintaining the consistency of markings *conflicting* and *unsuitable* after the potential solution graph has been modified (predicates *mark/3* and *remove_subgraph/3*).

In the case when constraints are explicitly represented as a set of binary constraints more efficient approach is first to preprocess an A/O-graf by removing *impossible* nodes. On the other hand, when constraints are symbolic conditions and heuristics with possible expensive computations, better approach is to use a search algorithm which can recognize *impossible* nodes (the one which we presented above).

In our prototype implementation, constraints are expressed by logical expressions.

With respect to our method of building software system configuration which defines also the way successor to an O-node is selected, the predicate *successors/2* must be modified accordingly. Version selection refers to properties of the particular successors of the node and makes use of specific knowledge on ordering them. When the version selection method fails to select a successor, a deadend occurs.

Let us demonstrate application of the method by an example of building a generic configuration for a system modelled by the A/O graph in Figure 1.
Let us assume that the node $MANAGER.4$ is constrained by a logic expression:

$\forall x(problem\_type(x) = design)$.

The above constraint preprocessed to the form of a set of nonacceptable combinations of nodes in solution graphs according to their atributes could be
$\{(MANAGER.4,\ INTERR.1)\}$.

Searching the A/O graph starts by visiting the root of the graph, i.e. the O-node $EVAL$. Version selection method selects the successor *EVAL.2*. After visiting this node, state of the searching process is:

$$OPEN \quad = \{INIT/\{EVAL.2\},\ ACTIONS/\{EVAL.2\}\}$$

$$CLOSED = \{EVAL/\{\},\ EVAL.2/\{EVAL\}\}$$

where indices identify pointers to predecessors.

In the next step, O-node *INIT* is removed from *OPEN*. Version selection method fails to select a successor to *INIT*. The deadend occurs. As a consequence, $INIT$ is marked *impossible*. *INIT* has two predecessors, i.e. the A-nodes *EVAL.1* and *EVAL.2*. They are marked *impossible*. To jump back, the node $EVAL$ is selected from among the predecessors of *INIT*. It has yet unexplored alternative successors. State of the searching process after this step is the same as at the beginning of the process:

$$OPEN \quad = \{EVAL/\{\}\}$$

$$CLOSED = \{\}$$

with the difference that two out of three successors of $EVAL$ are marked *impossible*. Processing the node $EVAL$ results in selecting the only possible successor, i.e. *EVAL.3*.

In the sequel, following nodes will be added to the potential solution graph: *ACTIONS*, *ACTIONS.1*, *INTERR*, *INTERR.1*, *MANAGER*. When attempting to add the "most suitable" successor *MANAGER.4* of *MANAGER* to the potential solution graph, a violation occurs (node *INTERR.1* does not satisfy defined constraint).

An attempt to reject *INTERR.1* does not resolve the violation, therefore *MANAGER.4* must be rejected. It will be marked *conflicting*. An alternative solution is sought in its predecessor, i.e. in *MANAGER*. Processing the *O*-node *MANAGER* selects its successor *MANAGER.1* which is consistent with the current solution graph and the "best" from the remaining alternatives.

Finally, solution is found as a subgraph of the given A/O graph containing the following nodes: *EVAL, EVAL.3, ACTIONS, ACTIONS.1, INTERR, INTERR.1, MANAGER, MANAGER.1, INFER, INFER.2.*

6 EXPERIMENTS

The new algorithm has been experimentally verified in the context of our new method of building a software system configuration. It has been implemented in Prolog language. We performed several experiments aiming at empirical evaluation of performance of our algorithm. There were performed extensive tests, in fact thousands of them, to gather data allowing certain conclusions.

6.1 Method of Experimentation

The aim of our experiments has been to verify basic properties of the proposed algorithm. In particular, we concentrate our attention to an evaluation of the efficiency of the new algorithm for searching A/O graphs with constraints.

Our hypothesis is that the new algorithm (RB) is an improvement over the standard searching algorithm with chronological backtracking (CHB).

As an independent measure to test efficiency of this kind of algorithms, we have chosen to count the number of deadends occuring during searching a particular graph (BACK). An alternative measure can be the number of consistency checks performed during searching a particular graph. In fact, we have experimented with both of these measures. We report on results using the former measure in this paper. Due to space limitations, we omit results using the latter. However, there has been a strong similarity between them. Similar measure was used also by others[11, 8].

Except of evaluating efficiency of the proposed algorithm, we have experimented with the following heuristics based on the A/O graph structure:

A. heuristics for selecting a node for processing from the list $OPEN$

   1. *from among several nodes, prefer those having more pointers to their predecessors*
   2. *prefer the node which is involved in the greatest number of constraints*

B. heuristics for selecting the successor of an O-node from among the set of its successors during search

1. *prefer the successor of an O-node which is involved in the least number of constraints*
2. *prefer the successor of an O-node which has the least number of successors.*

In order to achieve unbiased independent results, we have tested the algorithms with randomly generated problems (i.e., A/O graphs and sets of constraints). In the first series of experiments, we worked with such general problems. In the second series of experiments, we worked also with randomly generated problems, but the A/O graphs were constrained to have a structure similar to the typical model of a software system, not an arbitrary A/O graph. In particular, the graphs are special because their structure is such that A-nodes and O-nodes alternate regularly along every path, every A-node has exactly one predecessor, there is one root which is an O-node, and every O-node has at least one successor. Finally, in the third series of experiments, we worked with randomly generated problems, but the A/O graphs were further constrained to resemble even more faithfully the typical model of a software system. We have assumed that variants (A-nodes) included in a family of components (O-nodes) share part of their architectural relations. As a consequence, some successors to A-nodes which are successors to a single O-node are common.

For each kind of problems a specially designed generator produced 3000 different problems with number of nodes equal to 500. The reason for not including experiments with higher numbers of nodes was that they are obviously more time consuming and we could not have afforded them with our relatively modest computing environment. On the other hand, we have tried some experiments with other values as well. Based on them, we feel supported in the claim that our results are representative regardless this value.

The two algorithms CHB, RB were applied to the sample problems. First, numbers of deadends were counted occuring during application without using heuristics. Then, numbers of deadends were counted occuring during application that incorporated combination of heuristics from the above two groups (four combinations for each algorithm).

6.2 Results of the Experiments

The experiments show that all three types of generated problems are basically of the same character.

We summarize results of our experiments in two graphs. One serves for comparison of the two algorithms and the other one for evaluating the use of heuristics.

In Figure 2 we see how the number of deadends (denoted as BACK) for the new proposed algorithm which allows recording and propagating results of the analysis of the current deadend node (RB) depends on the number of deadends for the basic algorithm with chronological backtracking (CHB).

Fig. 2. Number of deadends for algorithms CHB, RB.

The presented graph shows that the new proposed algorithm performs better than the basic algorithm with chronological backtracking.

In Figure 3 we show how heuristics can improve searching of the new algorithm (RB). Figure is based on average values of 10 series of experiments. In each series we generated 50 different graphs with the same parameters set to the generator. We performed experiments for the following combinations of heuristics:

- without heuristics (denoted as RB-00)
- heuristics A1 and B1 (denoted as RB-11)
- heuristics A1 and B2 (denoted as RB-12)
- heuristics A2 and B1 (denoted as RB-21)
- heuristics A2 and B2 (denoted as RB-22).

To summarize, from the graph we can see that each combination of heuristics improves efficiency of the algorithm. We have observed very similar improvement for the basic algorithm (CHB). The best results are when the algorithm prefers a successor of an O-node which has the least number of successors (RB-12 a RB-22).

7 CONCLUSION

We have presented a Prolog implementation of two backtracking algorithms. The first one is the standard chronological backtracking algorithm for searching A/O graphs. The second one is our new algorithm for searching A/O graphs to find a solution satisfying constraints. The new algorithm follows a look-back scheme when it encounters a deadend. It uses also marking technique to record reasons for the deadend. This allows backjumping directed towards more promising alternatives instead of the backtracking towards the chronological predecessor. The new algorithm has been experimentally verified in the context of our new method of building a software system configuration.

Fig. 3. Improvement of algorithm RB by use of heuristics

Methods of implementing search with backtracking have been given a constant interest of researchers. From the recent works, Chakrabarti [6] proposed an algorithm which employs a marking technique similar to ours; however, he uses it to mark the current best solution. In the context of constraint logic programming, van Hentenryck and Ramachandran [18] proposed a new scheme called semantic backtracking. Again, the idea of using additional semantic information, which is mainly of heuristic nature, is similar to ours.

The choice of Prolog as an implementation language has been justified by our experience. In fact, we should not speak about implementation language only. Prolog, being a language that offers quite a powerful set of concepts allowing a highly declarative style of programming, is also a design, and perhaps to some extent a specification language as well. This provides for a design flexibility, including experimentation with prototype algorithm versions.

Of course, using standard Prolog faces limitations of various kinds. Most notably perhaps from the software system implementation point of view, is the lack of higher program structuring constructs. The problem has been addressed by several authors discussing modules [15], inheritance, contexts, or a complete language for structured logic programming [12]. The approaches make use of another very powerful concept of the standard Prolog, cf. that of meta-programming. In fact, we have used this concept, in combination with the technique of introspection, to structure programs as multi-level ones [4].

REFERENCES

[1] R.D.Jr. Bayardo and D.P. Miranker. An optimal backtrack algorithm for tree structured constraint satisfaction problems. *Artificial Intelligencce*, 71(1):159–181, 1994.

[2] M. Bieliková and P. Návrat. An approach to building software configuration using heuristic knowledge. In D. Karpić and V.H. Dobrić, editors, *Proc. of the 17th International Conference on Information Technology Interfaces ITI'95*, pages 575–580, Pula, 1995.

[3] M. Bieliková and P. Návrat. Modelling software systems in configuration management. *Applied Mathematics and Computer Science*, 5(4):751–764, 1995.

[4] M. Bieliková and P. Návrat. A programming technique of implementing multi-level logic programs. Technical report, Slovak Technical University, Bratislava, 1995.

[5] M. Bieliková and P. Návrat. A knowledge based method for building a software system configuration. *Knowledge Based Systems*, 9(1):61–65, 1996.

[6] P.P. Chakrabarti. Algorithms for searching explicit AND/OR graphs and their applications to problem reduction search. *Artificial Intelligence*, 65(2):329–346, 1994.

[7] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[8] R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.

[9] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[10] C. Elkan. Conspiracy numbers and caching for searching and/or trees and theorem proving. In *Proceedings ICJAI-89*, pages 341–348, Detroit, 1989.

[11] E. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[12] L. Giordano, Martelli A., and Rossi G. *Structured Prolog*: A language for structured logic programming. *Software - Concepts and Tools*, 15(3):125–145, 1994.

[13] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[14] A. Mahanti and A. Baghi. And/or graph search methods. *Journal of the ACM*, 32(1):28–51, 1985.

[15] D.A. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[16] P. Návrat and M. Bieliková. Knowledge controlled version selection in software configuration management. *Software - Concepts and Tools*, 17:40–48, 1996.

[17] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[18] P. van Hentenryck and V. Ramachandran. Backtracking without trailing in CLP($\Re_{Lin}$). *ACM Trans. on Programming Languages and Systems*, 17(4):635–671, 1995.

[19] C.P. Williams and T. Hogy. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70(1-2):73–118, 1994.

Pavol Návrat was born in 1952 in Bratislava, Slovakia. He received his Ing. (MSc.) summa cum laude in 1975, and his CSc. (PhD.) degree in Computing Machinery in 1983 both from Slovak Technical University in Bratislava. He has been with Slovak Technical University since 1975. In 1989-1990, he spent 5 months as a visiting associate professor with the Department of Informatics, University of Athens, Greece. In 1992-1994, he joined as an associate professor the Department of Mathematics, University of Kuwait for two academic years. Since 1996, he is a full professor of Computer Science and Engineering. He (co-)authored two books and numerous scientific papers. His scientific interests include automated programming and software engineering, knowledge-based methods for assistance in programming and software development, as well as other topics in Software Engineering. He is a member of the IEEE and its Computer Society, the American Association for Artificial Intelligence, and the Slovak Society for Informatics. He is also a member of the Association for the Advancement of Computing in Education, for which he volunteers as a regional liaison officer.

Mária Bieliková was born in Slovakia in 1966. She received her Ing. (MSc.) in 1989 from Slovak Technical University in Bratislava and her CSc. (PhD.) degree in 1995 from the same university. She is an assistant professor at the Department of Computer Science and Engineering at Slovak Technical University Bratislava, where she is a member of the Intelligent support of software development group. Her research interests include logic programming, knowledge engineering, software development and management of versions and software configurations.