An efficient algorithm for searching implicit AND/OR graphs with cycles

P. Jiménez C. Torras

Institut de Robòtica i Informàtica Industrial (CSIC - UPC)
Gran Capità 2-4 (Edifici NEXUS), E-08034 Barcelona, Spain

Abstract

We present an efficient AO^*-like algorithm that handles cyclic graphs without neither unfolding the cycles nor looping through them. Its top-down search strategy is based on Mahanti and Bagchi's *CF* [17], whereas its bottom-up revision process is inspired in Chakrabarti's *REV** [6]. However, important modifications have been introduced in both algorithms to attain a true integration and gain efficiency. Proofs of correctness and completeness are included. Up to our knowledge, the resulting algorithm –called CFC_{REV^*}– is the most efficient one available for this problem.

KEYWORDS: *AND/OR graphs, cyclic graphs, heuristic search, assembly/disassembly problems.*

1 Introduction

The use of AND/OR graphs for representing problems originated in the sixties within the domain of Artificial Intelligence. Since then, it has spread to other fields, such as Operations Research, Automation and Robotics, where AND/OR graphs are nowadays being used to represent cutting problems [1], interference tests [15], failure dependencies [3], robotic task plans [4], and assembly/disassembly sequences [8]. In the last three contexts, the need to develop algorithms for dealing with cyclic AND/OR graphs, possibly generated only as needed, has been explicitly stated [3, 4, 11]. The motivation for the present work comes from the last such context, namely assembly/disassembly sequencing, as presented in Section 2.

When turning to the literature on algorithms for searching AND/OR graphs, one realises that there have been two main periods in its historical development. Until the mid eightees, all the proposed algorithms worked on *implicit* graphs and made the assumption that the graphs be *acyclic* [7, 18, 19, 2, 17]. The usual way of approaching cyclic graphs with these AO^*-like algorithms was to "unfold" the cycles, but this can be highly inefficient when cycles have relative low-cost arcs [6] or even prevent algorithms from terminating.

In the early ninetees, efforts were directed to solving *explicit cyclic* AND/OR graphs. Both bottom-up algorithms –like *BUS* [16] and REV^* [6]– as well as top-down ones –like *Iterative_Revise* [6]– were proposed to find the least-cost solution trees of AND/OR graphs which may have cycles.

What options does one have nowadays for searching an implicit AND/OR graph with cycles? Only two proposals have appeared in the literature, both consisting of a modification to the cost revision part within an AO^*-like algorithm. The former relies on the observation that the cost revision process amounts to looking for an optimal solution subgraph within an *explicit* AND/OR graph. Section 6.1 in [6] gives some indications on how REV^* and *Iterative_Revise* could be used within AO^*. The second proposal consists of using cost bounds in the process of estimating the minimal cost of solution subgraphs, so as to prevent infinite looping [12, 13].

Both proposals permit finding an optimal subgraph within an implicit AND/OR graph with cycles. Now that correct and complete procedures to solve the problem are available, the next issue to address is *efficiency*. In this respect, none of the current options is entirely satisfactory. The complexity of the revision process proposed in [12, 13] depends not only on the size of the graph, but also on the costs of the arcs, it being unnecessarily high in the case of low-cost arc cycles. REV^* and *Iterative_Revise* have very similar worst-case complexities, the selection of one or the other depending mainly on the nature of the problem graph [6]. Even if these last two algorithms are efficient for searching explicit graphs, their direct use in the cost revision process within AO^* does not lead to an efficient algorithm for searching implicit graphs, as discussed in Section 5.

The aim of this paper is to present such an efficient algorithm. Its top-down search strategy is based on Mahanti and Bagchi's *CF* [17], whereas its bottom-up revision process is inspired in Chakrabarti's *REV** [6]. Thus, in Section 4, both previous algorithms are briefly reviewed and, in Section 5, a first attempt at their integration is discussed. Section 6 describes the modifications introduced to attain an efficient integration, leading to the CFC_{REV^*} algorithm. In Section 7, the correctness and completeness of CFC_{REV^*} are proved. Section 8 discusses its efficiency in relation to previous approaches. Finally, Section 9 summarizes the conclusions of the present work.

2 Motivating example

The motivation for this work comes from the assembly planning domain [20], where finding optimal or near-optimal assembly sequences, rather than just feasible sequences, is still an open problem. The assembly-by-disassembly approach has been widely adopted, both because starting with the final product leads to a much more constrained search, and also because maintenance and recycling often require partial disassembly plans. Within this approach, directional blocking graphs have become a standard tool [21]. The nodes in such a graph stand for the parts of an assembly, and, for a given direction of motion, an edge A $\longrightarrow$ B is included if part A collides with part B when A moves in that direction while B remains stationary. Often several directions of motion are considered, and then if one combines the different blocking graphs into a single AND/OR graph, cycles may appear [9].

A simple example where cyclic AND/OR graphs arise is shown in Figure 1(a). It is a 2-directional key part removal problem. Part K needs to be removed from the 2D assembly by executing at most a single translation for each part, either to the right or downwards. Thus, each part corresponds to an OR node, for it can be removed along either direction, and in turn each direction is represented by an AND node, since all the parts blocking the removal along the corresponding direction need to be taken away first. Now, observe that part B blocks the downward motion of part C, while part C blocks the motion of B to the right. This clearly corresponds to a cycle in the graph, as can be seen in Figure 1(b).

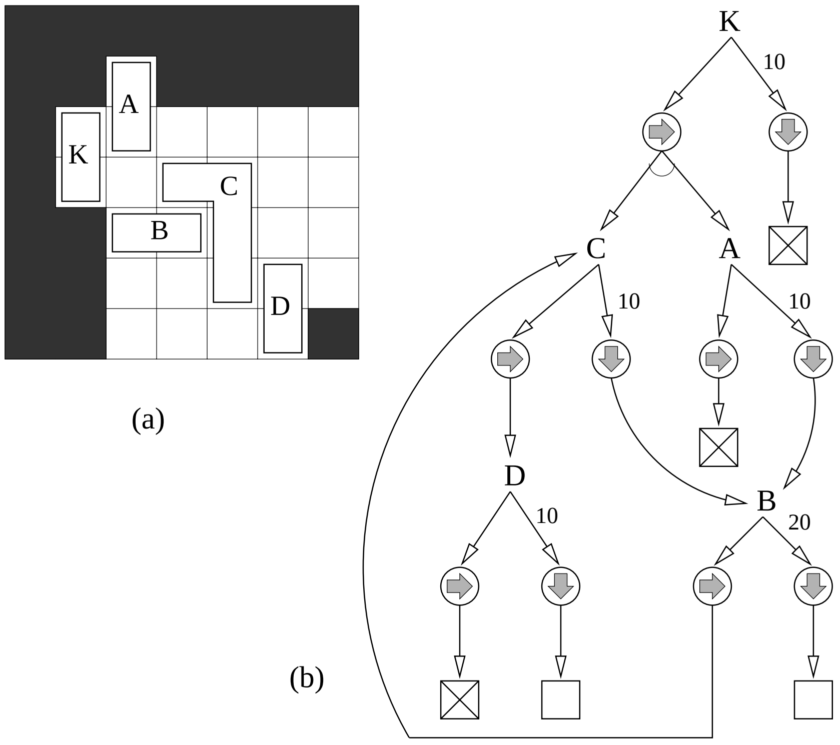

Figure 1: *(a) A 2D assembly. (b) The cyclic AND/OR graph to be searched for finding an optimal disassembly sequence to remove part K. OR and AND nodes are represented by letters and circles, respectively. Crossed boxes represent dead-ends, while non-crossed ones stand for terminal nodes. The costs associated to the arcs have been selected to illustrate the behaviour of the proposed algorithms, as discussed in the text. All the non-specified arc costs are equal to 1.*

Each solution subgraph of the AND/OR graph provides a partial ordering of removals, from which at least one[1] feasible disassembly sequence can be derived. For many products, there exists an exponentially large set of feasible sequences, so that obtaining one is not difficult, whereas generating them all to determine the best one is prohibitely expensive. Even for the simple example of Figure 1, there are as many as nine feasible sequences[2], stemming from three solution subgraphs.

The source of the high computational cost is not only traversing the graph, but especially generating it, since expanding each AND node entails collision checking of one part against all the others along a given direction, a particularly expensive operation [14]. Therefore, using an algorithm that, on the basis of an evaluation function, expands the graph only as needed, i.e., an algorithm that works on implicit graphs, is mandatory.

Concerning solution optimality, several primitive complexity measures for assembly sequences have been defined [10]: number of removed parts, number of directions, number of changes of direction, depth of the assembly sequence if parallelism is allowed, etc., some of which can be readily incorporated into the AND/OR graph structure, whereas others are more difficult to adapt. For example, the depth of the assembly corresponds directly to using the maxcost criterion at AND nodes, whereas the number of removed parts can be implemented through the sumcost criterion only if solution subgraphs have a tree structure. Nevertheless, these primitive measures are seldom used in isolation, but instead they are combined into cost functions that balance the different factors.

The algorithms that follow have been developed for the usual functions employed in the search literature, namely additive cost functions. This means evaluating the cost of a solution by adding up the costs associated to all its arcs. Other functions, such as those using the maxcost criterion, or the more general monotone ones defined in [17], could be readily used as well.

In the example of Figure 1, the best solution in terms of the number of removed parts starts by moving B downwards, then A and C also downwards in any order, and finally taking away K to the right. In order to illustrate the features of the developed algorithms –in particular the handling of cycles–, we assume that moving any part to the right has a cost of 1, while the downward motion has a cost of 10 for all parts, except for B for which it has a cost of 20. All the remaining arcs have a unitary cost. With this cost function, the best sequence is to move D downwards, then C and B to the right, A downwards, and finally take away K to the right.

Note, finally, that the assembly problem described is a particular instance of the general problem of scheduling with AND/OR precedence constraints.

3 Notation

We adhere to the standard notation and definitions stated in [17]. G denotes the implicit graph, i.e. the entire problem graph, implicitly specified by a start node s and a successor function. We assume that each node m in G has a finite set of immediate successors $S(m)$ and that every arc (m,n) in G has a cost $c(m,n) \geq \delta > 0$, where $\delta \in \Re$. Moreover, $P(m)$ denotes the set of immediate predecessors of node m.

For any node m in G, a solution graph $D(m)$ with root m is a finite subgraph of G defined as:

1. m is in $D(m)$;
2. if n is an OR node in $D(m)$, then exactly one of its immediate successors is in $D(m)$;
3. if n is an AND node in $D(m)$, then all its immediate successors are in $D(m)$;
4. every maximal (directed) path in $D(m)$ ends in a terminal node; and
5. no node other than m or its successors in G are in $D(m)$.

The portion of G generated up to a given point in the search is called the explicit graph G'. The notion of solution graph in G has its counterpart in that of a potential solution graph (*psg*) in G', the only difference concerning the fourth item: maximal paths end now at leaf nodes of G', called tip nodes, which might be either terminal or non-terminal.

A cost function $h(n, G)$ on every node n in G is defined in the usual way:

$$h(n,G) = g.l.b.\{h(n, D(n)) \mid D(n) \text{ is a solution graph with root } n \text{ in } G\},$$

where, for a node n in $D(m)$,

[1]The sequence is unique if the solution subgraph provides a complete ordering.

[2]Note that the structure of the graph itself prunes sequences with superfluous removals, otherwise the number of sequences would grow to twenty.

- $h(n, D(m)) = 0$, if n is a terminal node,
- $h(n, D(m)) = c(n, n') + h(n', D(m))$, if n is an OR node and n' its immediate successor in $D(m)$,
- $h(n, D(m)) = \sum_{i=1}^{k} [c(n, n_i) + h(n_i, D(m))]$, if n is an AND node with immediate successors $n_1, \cdots n_k$ in $D(m)$.

If there is no solution graph with root n in G, then $h(n, G) = \infty$. It follows immediately that $D(n)$ is a minimum-cost solution graph if and only if $h(n, D(n)) = h(n, G)$.

Finally, we assume that an heuristic estimate $\hat{h}(n)$ of $h(n, G)$ is associated to each node n in G. This estimate will be used to guide the search and reduce the number of expanded nodes.

4 Review of CF and REV^*

The previous algorithms CF and REV^* were expressed in quite different formats in their original sources. In order to describe later the non-trivial modifications required for their efficient integration, we first need to present them in a common pseudocode format.

4.1 The CF algorithm

Among the several AO^*-like algorithms proposed in the literature [7, 18, 19, 2, 17], we have chosen to extend CF to the case of cyclic graphs, because it was shown in [17] to have the best performance.

Algorithm CF

```
CF1: [Initially $G'$ consists only of the start node $s$.]
     $f(s) := 0$;
     **if** $s$ is a terminal node **then** label $s$ SOLVED;
CF2: **while** $s$ is not SOLVED **do**
       **begin**
         /* NODE EXPANSION */
         CF2.1: choose any unsolved tip node $n$ of the marked *psg* below $s$;
                expand $n$ generating all its immediate successors $S(n)$;
                **for** each $n' \in S(n)$ not already in $G'$ **do**
                  **begin**
                    $f(n') := 0$;
                    **if** $n'$ is a terminal node **then** label $n'$ SOLVED;
                  **end**;
         /* COST REVISION */
         CF2.2: create a set of nodes $Z := \{n\}$;
         CF2.3: **while** $Z \neq \emptyset$ **do**
                  **begin**
                    remove a node $m$ from $Z$ such that no descendant
                      of $m$ in $G'$ occurs in $Z$;
                    **if** $m$ is an AND node **then**
                      **begin**
                        $f\text{new} := \sum_{m' \in S(m)} [c(m, m') + \max\{\hat{h}(m'), f(m')\}]$;
                        mark the arc $(m, m')$ for each $m' \in S(m)$;
                        **if** every $m' \in S(m)$ is SOLVED **then** label $m$ SOLVED;
                      **end**;
                    **if** $m$ is an OR node **then**
                      **begin**
                        $f\text{new} := \min_{m' \in S(m)} [c(m, m') + \max\{\hat{h}(m'), f(m')\}]$;
                        let this minimum occur for $m' = m'_0$ [resolve ties
                          arbitrarily, but always in favour of a SOLVED node];
                        mark the arc $(m, m'_0)$;
                        **if** $m'_0$ is SOLVED **then** label $m$ SOLVED;
                      **end**;
                    if ($f(m) < f_{\text{new}}$) or ($m$ is SOLVED) then add to $Z$
                      all immediate predecessors of $m$ along marked arcs;
                    if $f_{\text{new}} > f(m)$ then $f(m) := f_{\text{new}}$;
                  end;
      end;
    exit with $f(s)$ as output.
```

The outer loop CF2 implements the top-down growth of G', while the inner loop CF2.3 carries out the bottom-up revision of costs. At every moment, below each node in G', exactly one *psg* is marked, meaning that all its arcs are marked. Cost estimates f are revised from the expanded node up only along marked arcs. During this revision, arc-marking changes[3] may occur at OR nodes in the currently marked *psg*, leading to an alternative, more promising, marked *psg*.

It was proved in [17] that, if $\hat{h}$ is admissible (i.e., $\hat{h}(n) \leq h(n, G), \forall n \in G$), then *CF* terminates by either finding a minimum-cost solution graph rooted at s or else returning $f(s) = \infty$. Moreover, *CF* was proven to expand less nodes than other variants of the AO^* algorithm, and also to behave better than them when $\hat{h}$ is not admissible.

4.2 The REV^* algorithm

REV^* and *Iterative_Revise* are both equally suitable to directly replace the cost revision part of the *CF* algorithm to allow it to handle cyclic graphs. However, if a true integration is sought in order to maximize efficiency, then REV^* is the best option. This is mainly due to the bottom-up nature of REV^*, which permits visiting only nodes whose costs may be affected by the expansion of the tip node. The situation is even more unfavourable to *Iterative_Revise* if one takes into account that its worst-case complexity is quadratic in the number of visited nodes. Another reason is the maintenance of updated marks and updated costs for all nodes in G', which *Iterative_Revise* cannot guarantee. On the other hand, the drawbacks of REV^* in front of *Iterative_Revise* are not so in this particular setting: G' will never be disconnected and the need to consider all leaf nodes, even if they do not contribute to the solution, can easily be avoided. In sum, REV^* can be modified to both exploit and maintain the information (cost estimates and arc markings) incrementally collected by the *CF* algorithm, as will become clear in Section 5, while *Iterative_Revise* cannot accommodate some of these changes.

Algorithm REV^*

```
/* COST INITIALIZATION */
REV*1: create a set of nodes $O := \{ \}$;
         for each $n \in G$ do
           if $n$ is a terminal node then
             begin
               found[$n$] := true;
               $f(n) := t(n)$;
               add $n$ to $O$;
             end
           else
             begin
               found[$n$] := false;
               $f(n) := \infty$;
             end;
/* COST REVISION */
REV*2: while not (found[$s$]) do
         begin
           REV*2.1: if $O = \emptyset$ then exit ("no solution");
           REV*2.2: remove the node $m$ from $O$ with smallest $f(m)$
                      [resolve ties for $s$ if it is a candidate];
           REV*2.3: found[$m$] := true;
           REV*2.4: if not(found[$s$]) then
                      for each $p \in P(m)$ and not(found[$p$]) do
                        begin
```

[3]Marking a successor of an OR node in the pseudocode means also deleting the previous mark.

```
                if and(found[$p'$]) for each $p' \in S(p)$ then
                  begin
                    if $p$ is an AND node then
                      $f(p) := \sum_{p' \in S(p)}[c(p,p') + f(p')]$;
                    if $p$ is an OR node then
                      $f(p) := \min_{p' \in S(p)}[c(p,p') + f(p')]$;
                    if $p \in O$ then remove it from $O$;
                    $m := p$;
                    go to REV*2.3;
                  end;
                if $p$ is an OR node then
                  begin
                    cost := $c(p,m) + f(m)$;
                    if cost $< f(p)$ then
                      begin
                        $f(p)$ := cost;
                        add $p$ to $O$;
                      end;
                  end;
              end;
end.
```

Initially, only the terminal nodes are declared found and assigned a finite cost $t(n)$. The found status and the corresponding costs are propagated upwards to all nodes for which all immediate successors are declared found. OR nodes not satisfying this condition get their costs updated as well, but their status are not changed to found. Instead, they are added to a list O of pending nodes. When no further propagation is possible, the node from O having the lowest f value is extracted from O and declared found. It has been proved [6] that a node n is declared found only when $f(n)$ equals the minimum cost $h(n, G)$. By following this bottom-up conservative strategy similar to Dijkstra's shortest path, it is ensured that the cost revision process will never loop around a cycle.

This is the way the algorithm is intended to work, as described in [6]. However, the code contains a bug derived from the "go to" statement: the **for** loop in $REV^*2.4$ needs to be completed regardless of whether the "go to" statement is executed or not, because otherwise the found status and costs would not be properly propagated upwards. But this is not the conventional meaning of a "go to", a recursive scheme should be used instead, as presented in the next section.

5 Integrating CF and REV^*

The most direct way to combine CF and REV^* to yield an algorithm that solves *cyclic implicit* AND/OR graphs is to replace the cost revision part of CF by REV^*. After the cost revision, then, it is necessary to perform a top-down pass to obtain the currently best *psg* below s, from which a tip node will be selected for expansion. Some minor changes need to be introduced even in this case. For instance, the termination condition of CF should be changed since now the "solved" status is not propagated upwards and, more importantly, cyclic graphs may not have a solution –a possibility that didn't exist for CF. The new algorithm will terminate when either the currently best *psg* below s doesn't contain any tip node or when REV^* returns "no solution".

Of course, this is not a very efficient combination of the two algorithms, since the costs of all nodes in G' are revised at every iteration (even if most of them are not affected by the expansion of the new node) and an extra top-down pass has to be carried out. One may note that this is due to the fact that the two key features of CF, namely incremental cost calculations and arc markings, are not exploited in this arrangement.

The next degree of integration is thus to *make use of arc markings.* Only the costs of the ancestors of the expanded node along marked arcs should be revised. Thus, all such nodes will be placed in the set Z of "revisable" nodes after node expansion and, within REV^*, only the costs of nodes in Z will be computed[4]. Moreover, arc markings will be appropriately changed for these nodes.

A further degree of integration is to *exploit incremental cost calculations.* The idea is to use the costs from the preceding iteration as lower bounds on the corresponding costs in the current one. In this way,

[4] Z has been implemented as a "revisable" flag, but it is denoted as a set to simplify the code.

the number of times OR nodes are visited (and their costs are updated) may decrease, because they can be declared "found" as soon as their lower bound is reached.

These are the main modifications we introduced into *CF* and *REV**, following the indications in [6], in our first attempt at an integrated algorithm. Other minor modifications will be discussed after presenting the pseudocode description of this algorithm, which we call *INT*. Its overall structure is that of *CF*, with a top-down growth of G' and a bottom-up revision of costs. Moreover, like in *CF*, below each node in G', exactly one *psg* is marked at every instant. The resulting algorithm has four parts:

1. **Node expansion.** The same as in *CF*.
2. **Initialization of revisable nodes.** Revisable nodes (i.e., ancestors of the expanded node along marked arcs) are identified and their cost estimates are set to ∞ after saving their old values.
3. **Initialization of open nodes.** All tip nodes, regardless of whether they are terminal or not, are placed in the list O of nodes from which the cost revision process will proceed upwards. The "found" status of the remaining nodes is set to false.
4. **Definitive cost assignment and arc marking.** Nodes are extracted from O in order of increasing f values and their ancestors are visited recursively. Only the cost estimates of revisable nodes are updated. A node is declared "found" whenever one of the following three conditions is satisfied: all its children are "found", it is removed from O because it has the smallest f value, or its f value has not changed from the preceding iteration. Arcs markings and "solved" labels are processed as in the *CF* algorithm.

A pseudocode description of the algorithm follows. The modifications with respect to *CF* and *REV** are enclosed in boxes. The meaning of the doubly framed parts will become clear when we present the CFC_{REV^*} algorithm in the next section.

Algorithm *INT*

```
INT1: [Initially $G'$ consists only of the start node $s$.]
      $f(s) := 0$;
      if $s$ is a terminal node then label $s$ SOLVED;
INT2: while ($s$ is not SOLVED) $\boxed{\text{and } (f(s) \neq \infty)}$ do
        begin
          /* NODE EXPANSION */
          INT2.1: choose any unsolved tip node $n$ of the marked psg below $s$;
                  expand $n$ generating all its immediate successors $S(n)$;
                  for each $n' \in S(n)$ not already in $G'$ do
                    begin
                      $f(n') := 0$;
                      if $n'$ is a terminal node then label $n'$ SOLVED;
                      $\boxed{\text{found}[n'] := \text{true};}$
                    end;
          /* INITIALIZATION OF REVISABLE NODES */
          INT2.2: create a set of nodes $Z := \{n\}$;
                  ┌──────────────────────────┐
                  │ build-revisable($n$);    │
                  │ for each $m \in Z$ do    │
                  │   begin                  │
                  │   ┌────────────────────┐ │
                  │   │ $f_{old}(m) := f(m)$; │ │
                  │   │ $f(m) := \infty$;  │ │
                  │   └────────────────────┘ │
                  │   end;                   │
                  └──────────────────────────┘
          /* INITIALIZATION OF OPEN NODES */
                  create a set of nodes $O := \{ \}$;
                  for each $m \in G'$ do
                    if $m$ is a $\boxed{\text{tip node}}$ then
                      add $m$ to $O$
```

```
                else
                  found[$m$] := false;
        /* DEFINITIVE COST ASSIGNMENT AND ARC MARKING */
        INT2.3: while $\boxed{O \neq \emptyset}$ do
                begin
                  remove the node $m$ from $O$ with smallest $f(m)$;
                  if $m$ is a non-tip OR node then
                    begin
                      let $m'_0 \in S(m)$ be such that $f(m) = c(m, m'_0) + f(m'_0)$ [resolve
                        ties arbitrarily, but always in favour of a SOLVED node];
                      mark the arc $(m, m'_0)$;
                      if $m'_0$ is SOLVED then label $m$ SOLVED;
                    end;
                  cost-prop($m$);
                end;
    end;
    exit with $f(s)$ as output.
```

Subroutines

```
procedure build-revisable($m$);
begin
  for each $p \notin Z$ immediate predecessor of $m$ along a marked arc do
    begin
      add $p$ to $Z$;
      build-revisable($p$);
    end;
end.
```

```
procedure cost-prop($m$);
begin
   found[$m$] := true;
   for each $p \in P(m)$ and not(found[$p$]) do
   $\boxed{\textbf{if } p \in Z \textbf{ then}}$
      if and(found[$p'$]) for each $p' \in S(p)$ then
        begin
          if $p$ is an AND node then
            begin
              $f(p) := \sum_{p' \in S(p)} [c(p, p') + \max\{\hat{h}(p'), f(p')\}]$;
              mark the arc $(p, p')$ for each $p' \in S(p)$;
              if every $p' \in S(p)$ is SOLVED then label $p$ SOLVED;
            end;
          if $p$ is an OR node then
            begin
              $f(p) := \min_{p' \in S(p)} [c(p, p') + \max\{\hat{h}(p'), f(p')\}]$;
              let this minimum occur for $p' = p'_0$ [resolve ties arbitrarily,
                but always in favour of a SOLVED node];
              mark the arc $(p, p'_0)$;
              if $p'_0$ is SOLVED then label $p$ SOLVED;
            end;
          if $p \in O$ then remove it from $O$;
          cost-prop($p$);
        end
```

```
    else
      if $p$ is an OR node then
        begin
          cost := $c(p,m) + \max\{\hat{h}(m), f(m)\}$;
          if cost $< f(p)$ then
            begin
              $f(p)$ := cost;
              ┌──────────────────────────────────────────────┐
              │if $f_{\text{old}}(p) < f(p)$ then            │
              │  add $p$ to $O$                              │
              │else                                          │
              │  begin                                       │
              │    if $p \in O$ then remove it from $O$;     │
              │    mark the arc $(p,m)$;                     │
              │    if $m$ is SOLVED then label $p$ SOLVED;   │
              │    cost-prop($p$);                           │
              │  end;                                        │
              └──────────────────────────────────────────────┘
            end;
        end;
  ┌─────────────────┐
  │else             │
  │  cost-prop($p$);│
  └─────────────────┘
end.
```

The key idea beneath the algorithm is to update only the cost estimates of revisable nodes, by exploiting arc markings. To this end, the cost initialization of REV^* has been replaced by the initialization of both revisable nodes and open nodes in the integrated algorithm. Moreover, the cost revision of REV^* has been transformed into the cost-prop subroutine above.

Another important modification is to use the cost estimates from the preceding iteration f_{old} as effective lower bounds, so as to stop the cost revision at OR nodes whenever the bound is reached.

Besides these two, many other minor modifications have been introduced for the algorithm to perform appropriately. Thus, in order to have the cost estimates of all nodes in G' updated after each iteration, the condition in the main loop REV*2 has been changed to INT2.3, otherwise nodes in G' with a cost higher than $f(s)$ wouldn't get updated. Moreover, the "go to" in REV^* has been converted into a recursive call to procedure **cost-prop**. Besides improving the algorithm structure, this change permits overcoming the possible malfunctioning of REV^* mentioned at the end of the preceding section. The termination condition of the main loop CF2 has been changed to INT2, since cyclic graphs may not have a solution and then the algorithm will return $f(s) = \infty$. Note that, not just s, but all nodes not having a *psg* below them will have an infinite cost estimate at termination of the algorithm, this being one of the purposes of the initialization INT2.2. Finally, the propagation of "solved" labels, the use of $\hat{h}$ values and the updating of arc markings have been adapted, maintaining the same roles they play in CF.

Figures 2 and 3 show how the algorithm INT works for the problem in Figure 1, and may help illustrate its main features:

- *An arc closing a marked path will never be marked, ensuring that G' will always be free of marked cycles.* In iteration 13, the expansion of the left successor of node B generates a cycle in G'. However, at step INT2.3, the **cost-prop** routine reaches C before the expanded node and, therefore, an arc marking change occurs at C before the arc stemming from the expanded node is marked.

- *The set of revisable nodes Z is a subset (sometimes small) of G'.* In iteration 13, G' contains 13 non-leaf nodes, only 8 of which are revised. The savings are not large in this case due to the nature of the example chosen: a small highly interconnected graph. In general, the current potential solution graph (*psg*) will be a small fraction of the entire explicit graph, and Z is always a subset of the current *psg*.

- *Some cost updates at OR nodes may be saved by using the cost estimates from the preceding iteration (f_{old}) as lower bounds.* In iteration 14, all OR nodes (D, C, B, A and K) are declared found before all their children are found, because the cost estimates they inherit from their respective first children found equal their costs in the preceding iteration.

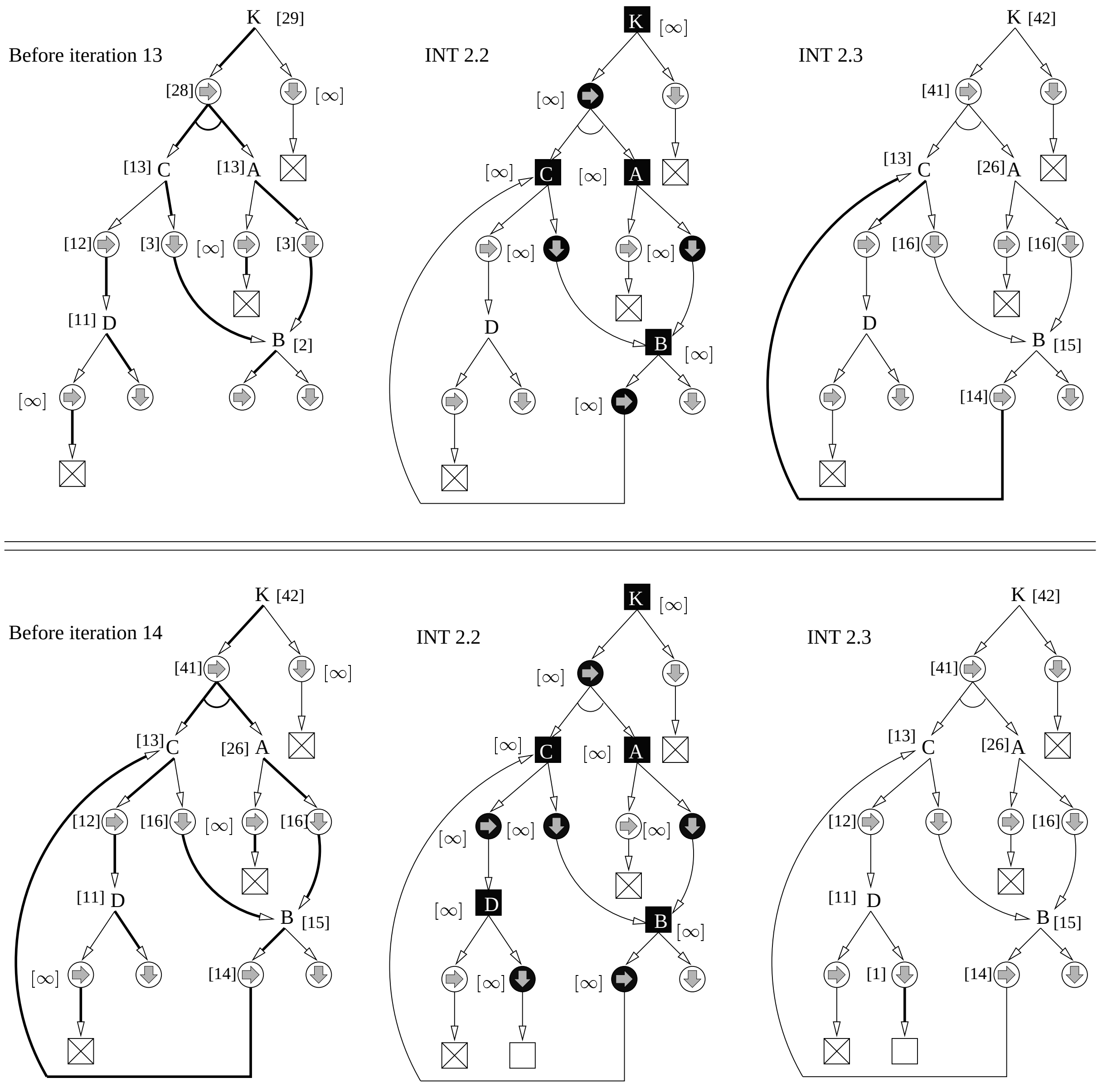

Figure 2: *Execution of algorithm INT on the graph displayed in Figure 1. The heuristic estimates* $\hat{h}$ *have been set to 0 and 1, for terminal and non-terminal nodes, respectively. Tip nodes from the marked psg are selected for expansion from left to right and from top to bottom. The behaviour of cost estimates* $f(n)$ *at the two last iterations of loop INT2 is shown. In particular, the* f *values before initiating the iteration, those modified by the execution of INT2.2, and those calculated at INT2.3 are displayed between brackets. The entire* G' *is displayed each time, with revisable nodes filled in black.*

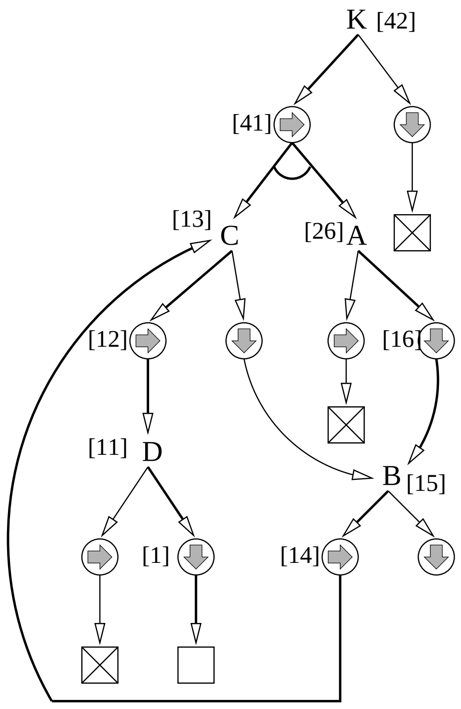

Figure 3: *The solution found for the graph in Figure 1 is displayed. Note that an arc which initially closed a cycle, is now part of the solution subgraph. Moreover, not all nodes in the graph have been expanded, highlighting the interest of working with implicit graphs. Although, in this example, only the right successor of B has not been expanded, a whole tree could be hanging from this node.*

6 The CFC_{REV^*} algorithm

In the algorithm presented in the preceding section, many cost updates are saved by exploiting arc markings and incremental cost calculations. However, it is still unsatisfactory in that the cost revision has to start at all leaf nodes every time to ensure that nodes within a cycle are revised in the appropriate order. Moreover, the cost propagation proceeds upwards even if cost estimates remain the same as those in the preceding iteration.

The ideal goal is to visit only nodes whose costs, arc markings or "solved" status change as a result of the expansion of a new node. In other words, the cost propagation should not only be confined to revisable nodes (avoiding to start at all leaf nodes), but also stopped wherever no change from the preceding iteration is performed.

The algorithm CFC_{REV^*}[5] does exactly this, through *a deeper exploitation of arc markings and incremental cost calculations.* Two visual metaphors may help illustrate its workings. The initialization metaphor is that of "collapsing" G' onto the set of revisable nodes Z, so that nodes that have a *psg* outside Z play the role of leaves. This is attained through an appropriate local cost revision process carried out within Z. The closing metaphor is that of extinguishing the propagation fronts where no changes occur. This affects both the local cost revision and the definitive cost assignment processes.

The CFC_{REV^*} algorithm has the same structure and number of parts as *INT*, but the third part is entirely different, and the second and fourth parts have changed slightly:

1. **Node expansion.** The same as in *INT*.

2. **Initialization of revisable nodes.** The same as in *INT*, except that the cost estimates f of revisable nodes are not set to ∞. This is to avoid that they remain ∞ if the propagation is stopped before reaching them. Now, instead of saving the old cost estimates in f_{old} and updating f directly, the new tentative values will be recorded in f_{new} (in the third part of the algorithm) and f will only be updated, if needed, in the definitive cost assignment step. Moreover, while in *INT* the "found" propagation process sweeps all G', here it is confined within Z and, therefore, only the "found" status of nodes in Z are initialized to false.

[5]The second C stands for cyclic.

3. **Local cost revision to initialize open nodes.** Revisable nodes having a *psg* below them disjoint with Z are assigned the minimum cost estimate coming from their children and placed in O. This process starts at the expanded node and propagates upwards along Z (which can be thought of as an acyclic graph rooted at the expanded node), stopping propagation in all those branches where no cost changes occur. The nodes traversed that are not placed in O are assigned an infinite new cost estimate.

4. **Definitive cost assignment and arc marking.** All the updatings (of cost estimates, arc markings and SOLVED status) are carried out within this step. The main difference with respect to *INT* is the pruning of revisable nodes whose cost estimates cannot change. This is achieved by means of the procedure prune-revisable. Since the SOLVED status of these nodes may change, the procedure SOLVED-propagation takes care of propagating these changes. Through this mechanism, many cost updates are saved. Three minor modifications are the assignment of infinite costs to nodes that remain "unfound" after emptying O, the marking of descendants of an AND node only if it is the expanded node (since, in the other cases, they are already marked), and the need of removing from O nodes that have been pruned from Z.

A pseudocode description of the algorithm follows. The additions with respect to *INT* are enclosed in boxes, while the removals from *INT* appeared doubly framed in the code of that algorithm.

Algorithm CFC_{REV^*}

```
CFC1: [Initially $G'$ consists only of the start node $s$.]
      $f(s) := 0$;
      **if** $s$ is a terminal node **then** label $s$ SOLVED;
CFC2: **while** ($s$ is not SOLVED) and ($f(s) \neq \infty$) **do**
        **begin**
          /* NODE EXPANSION */
          CFC2.1: choose any unsolved tip node $n$ of the marked *psg* below $s$;
                  expand $n$ generating all its immediate successors $S(n)$;
                  **for** each $n' \in S(n)$ not already in $G'$ **do**
                    **begin**
                      $f(n') := 0$;
                      $f_{\text{new}}(n') := 0$;
                      **if** $n'$ is a terminal node **then** label $n'$ SOLVED;
                      found[$n'$] := true;
                    **end**;
          /* INITIALIZATION OF POTENTIALLY REVISABLE NODES */
          CFC2.2: create a set of nodes $Z := \{n\}$;
                  **build-revisable**($n$);
                  **for** each $m \in Z$ **do**
                    $\boxed{\text{found}[m] := \text{false};}$
          /* LOCAL COST REVISION TO INITIALIZE OPEN NODES */
                  create a set of nodes $O := \{\ \}$;
                  $\boxed{\textbf{init-open}(n);}$
          /* DEFINITIVE COST ASSIGNMENT AND ARC MARKING */
          CFC2.3: **while** $O \neq \emptyset$ **do**
                    **begin**
                      remove the node $m$ from $O$ with smallest $f_{\text{new}}(m)$;
                      **if** $m$ is an OR node **then**
                        **begin**
                          let $m'_0 \in S(m)$ be such that $f_{\text{new}}(m) = c(m, m'_0)+$
                            $\max\{\hat{h}(m'_0), f_{\text{new}}(m'_0)\}$ [resolve ties arbitrarily,
                            but always in favour of a SOLVED node];
                          mark the arc $(m, m'_0)$;
                          **if** $m'_0$ is SOLVED **then** label $m$ SOLVED;
                        **end**;
```

```
            if $f_{\text{new}}(m) > f(m)$ then
              $f(m) := f_{\text{new}}(m)$
            else
              begin
                if $m$ is SOLVED then SOLVED-propagation($m$);
                prune-revisable($m$);
              end;
            cost-propagation($m$);
          end;
        for each $m \in Z$ and not(found[$m$]) do
          $f(m) := \infty$;
    end;
    exit with $f(s)$ as output.
```

Subroutines

```
procedure init-open($m$);
begin
  if ($m$ is an AND node) and ($S(m) \cap Z = \emptyset$) then
    begin
      $f_{\text{new}}(m) := \sum_{m' \in S(m)} [c(m, m') + \max\{\hat{h}(m'), f(m')\}]$;
      mark the arc $(m, m')$ for each $m' \in S(m)$;
      if every $m' \in S(m)$ is SOLVED then label $m$ SOLVED;
      add $m$ to $O$;
      if $f_{\text{new}}(m) > \max\{\hat{h}(m), f(m)\}$ then
        for each $p \notin O$ immediate predecessor of $m$ along a marked arc do
          init-open($p$);
    end
  else
    if ($m$ is an OR node) and ($S(m) \setminus Z \neq \emptyset$) then
      begin
        $f_{\text{new}}(m) := \min_{m' \in S(m) \setminus Z} [c(m, m') + \max\{\hat{h}(m'), f(m')\}]$;
        add $m$ to $O$;
        if $f_{\text{new}}(m) > f(m)$ then
          for each $p \notin O$ immediate predecessor of $m$ along a marked arc do
            init-open($p$);
      end
    else
      begin
        $f_{\text{new}}(m) := \infty$;
        for each $p \notin O$ immediate predecessor of $m$ along a marked arc do
          init-open($p$);
      end;
end.
```

```
procedure SOLVED-propagation($m$);
begin
  for each $p$ immediate predecessor of $m$ along a marked arc such that $p$ is not SOLVED do
    if (($p$ is an AND node) and (every $p' \in S(p)$ is SOLVED)) or ($p$ is an OR node) then
      begin
        label $p$ SOLVED;
        SOLVED-propagation($p$);
      end;
end.
```

```
procedure prune-revisable($m$);
begin
  remove $m$ from $Z$;
  for each $p \in Z$ immediate predecessor of $m$ along a marked arc do
    if ($p$ is an AND node) and ($S(p) \cap Z = \emptyset$) or ($p$ is an OR node) then
      begin
        $f_{\text{new}}(p) := f(p)$;
        prune-revisable($p$);
      end;
end.
```

```
procedure cost-propagation($m$);
begin
   found[$m$] := true;
   for each $p \in P(m)$ and not(found[$p$]) do
      if $p \in Z$ then
         if and(found[$p'$]) for each $p' \in S(p)$ then
            begin
               if $p$ is an AND node then
                  begin
                     $f_{\text{new}}(p) := \sum_{p' \in S(p)} [c(p,p') + \max\{\hat{h}(p'), f(p')\}]$;
                     $\boxed{\textbf{if } p = n \textbf{ then}}$ mark the arc $(p,p')$ for each $p' \in S(p)$;
                     if every $p' \in S(p)$ is SOLVED then label $p$ SOLVED;
                  end;
               if $p$ is an OR node then
                  begin
                     $f_{\text{new}}(p) := \min_{p' \in S(p)} [c(p,p') + \max\{\hat{h}(p'), f(p')\}]$;
                     let this minimum occur for $p' = p'_0$ [resolve ties arbitrarily,
                        but always in favour of a SOLVED node];
                     mark the arc $(p, p'_0)$;
                     if $p'_0$ is SOLVED then label $p$ SOLVED;
                  end;
               if $p \in O$ then remove it from $O$;
               if $f_{\text{new}}(p) > f(p)$ then
                 $f(p) := f_{\text{new}}(p)$
               else
                 begin
                   if $p$ is SOLVED then SOLVED-propagation($p$);
                   prune-revisable($p$);
                 end;
               cost-propagation($p$);
            end
         else /* at least one successor of $p$ is unfound */
            if $p$ is an OR node then
               begin
                  cost := $c(p,m) + \max\{\hat{h}(m), f(m)\}$;
                  if cost $< f_{\text{new}}(p)$ then
                     begin
                        $f_{\text{new}}(p)$ := cost;
                        if $f_{\text{new}}(p) > f(p)$ then
                           add $p$ to $O$
                        else
                           begin
                              if $p \in O$ then remove it from $O$;
                              mark the arc $(p,m)$;
                              if $m$ is SOLVED then $\boxed{\textbf{SOLVED-propagation}(m)}$;
```

```
                    $\boxed{\textbf{prune-revisable}(p);}$
                    **cost-propagation**($p$);
                  **end**;
              **end**;
          **end**;
    else /* $p \notin Z$ */
      **begin**
        $\boxed{\textbf{if } p \in O \textbf{ then} \text{ remove it from } O;}$
        **cost-propagation**($p$);
      **end**;
**end**.
```

To ease the comparison with *INT*, the same two iterations recorded for that algorithm in Figure 2 are now worked out for CFC_{REV^*} in Figure 4. The main improvements are as follows:

- *The set of open nodes O is now a subset of Z (thus, the propagation has no longer to start at all leaf nodes).* In iteration 13, the set O contains the nodes B and C, which play the role of leaves from which cost estimates are propagated upwards.
- Only nodes having a *psg* disjoint with Z may be included in O; therefore, the only AND node susceptible of getting into O is the expanded node. Moreover, *not all OR nodes having a psg disjoint with Z are included in O. By using the cost estimates from the preceding iteration, the local cost revision stops at nodes whose estimates do not change.* In iteration 14, the local cost revision stops at the right successor of D, since its cost estimate equals its $\hat{h}$ value. Only this node is included in O.
- *The nodes effectively revised constitute a subset (sometimes small) of those potentially revisable* Z, which, as shown before, is a subset (sometimes small) of G'. In iteration 14, Z contains 11 nodes, but only the costs of 2 of them (D and its right successor) are effectively revised by CFC_{REV^*}. The cost estimate of D is updated in CFC2.3, while that of its right successor is computed in CFC2.2.
- *When a node becomes SOLVED maintaining its previous cost estimate, then only the SOLVED status is propagated upwards.* This is illustrated in iteration 14, where D maintains its previous cost estimate.
- *Even when the cost estimate of a given revisable node does not change, so that it and its marked predecessors are pruned from Z, the "found"-status propagation needs to continue upwards to appropriately update non-marked predecessors.* In particular, this is the way in which an expanded node initially generating a cycle gets updated, as illustrated in iteration 13.

Figure 5 illustrates how the algorithm handles cases where some nodes must be assigned infinite cost estimates, because they are within a cycle and do not have a *psg* below them. Note that CFC_{REV^*} assigns infinite cost estimates to the nodes that remain "unfound" after having propagated costs from all open nodes.

7 Validity of INT and CFC_{REV^*}

The correctness and completeness of *INT* follow directly from those of *CF* and REV^*. The only substantial modifications introduced, namely using the preceding cost estimates as lower bounds and restraining updates to revisable nodes, do not affect these properties, since they just prune steps that would leave the state of the graph unchanged. Moreover, changes in arc markings and SOLVED status are performed only when minimum cost estimates have been reached for the corresponding nodes, thus ruling out the possibility of having marked cycles.

In order to prove the correctness and completeness of CFC_{REV^*}, let us first introduce some notation.

Instant j is the moment at which step CFC2 (or, in its case, INT2) is initiated for the j-th time.

$G'_{CFC}(j)$ (resp. $G'_{INT}(j)$) is the state of the explicit graph –topology, costs, arc markings and SOLVED status– at instant j of the execution of CFC_{REV^*} (resp. *INT*). Reference to the algorithm is dropped when both sets coincide.

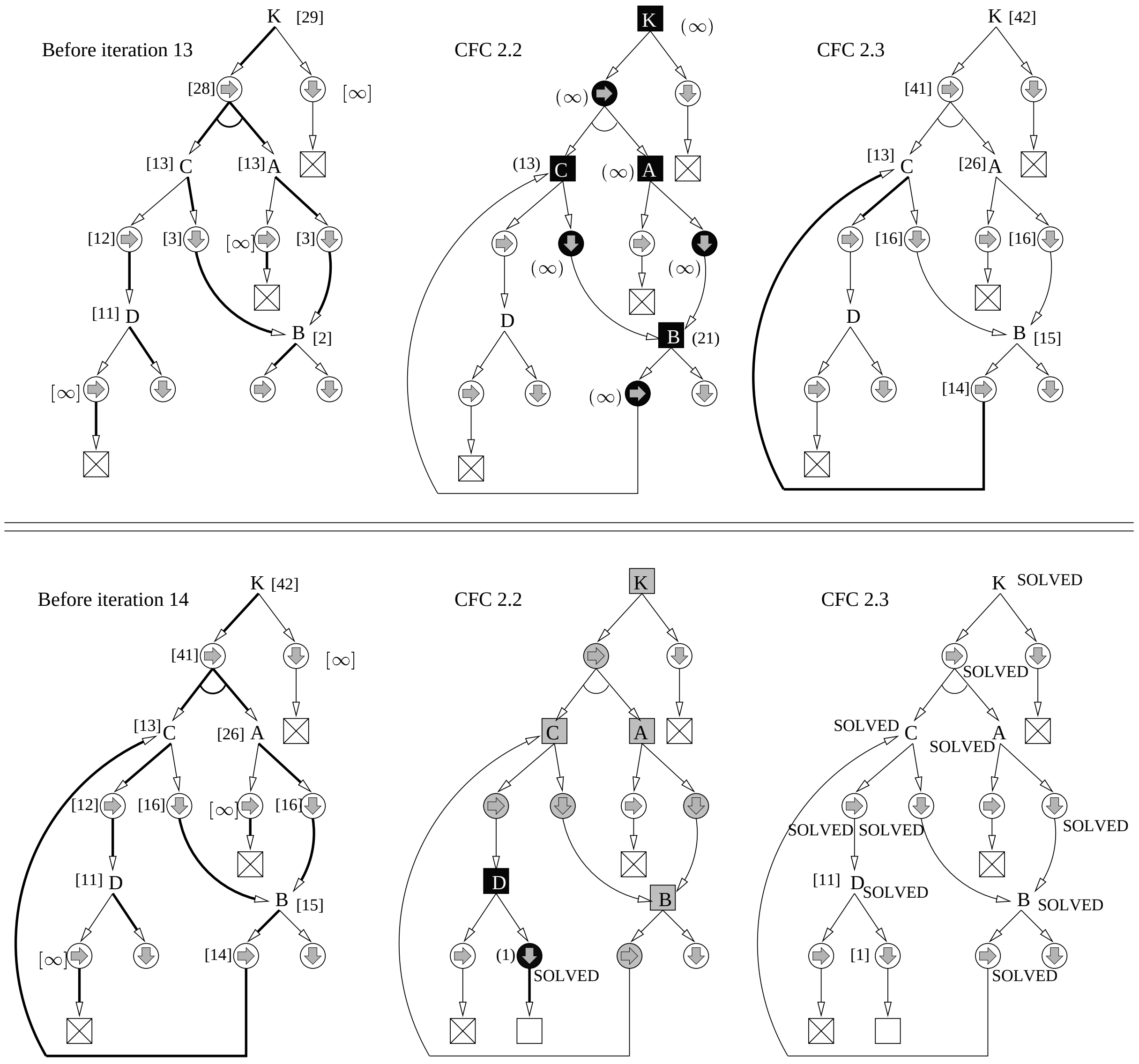

Figure 4: *Execution of CFC_{REV^*} on the graph displayed in Figure 1. The same instants as those shown in Figure 2 are displayed to favour the comparison with INT. The f_{new} values are shown in parentheses, maintaining the brackets for the f values. Two shadings are used for the revisable nodes in step CFC2.2, the darker one signaling those nodes whose costs are actually revised during this step (their f_{new} values are either set to infinity or assigned a finite value and, in the latter case, the corresponding nodes are placed in O).*

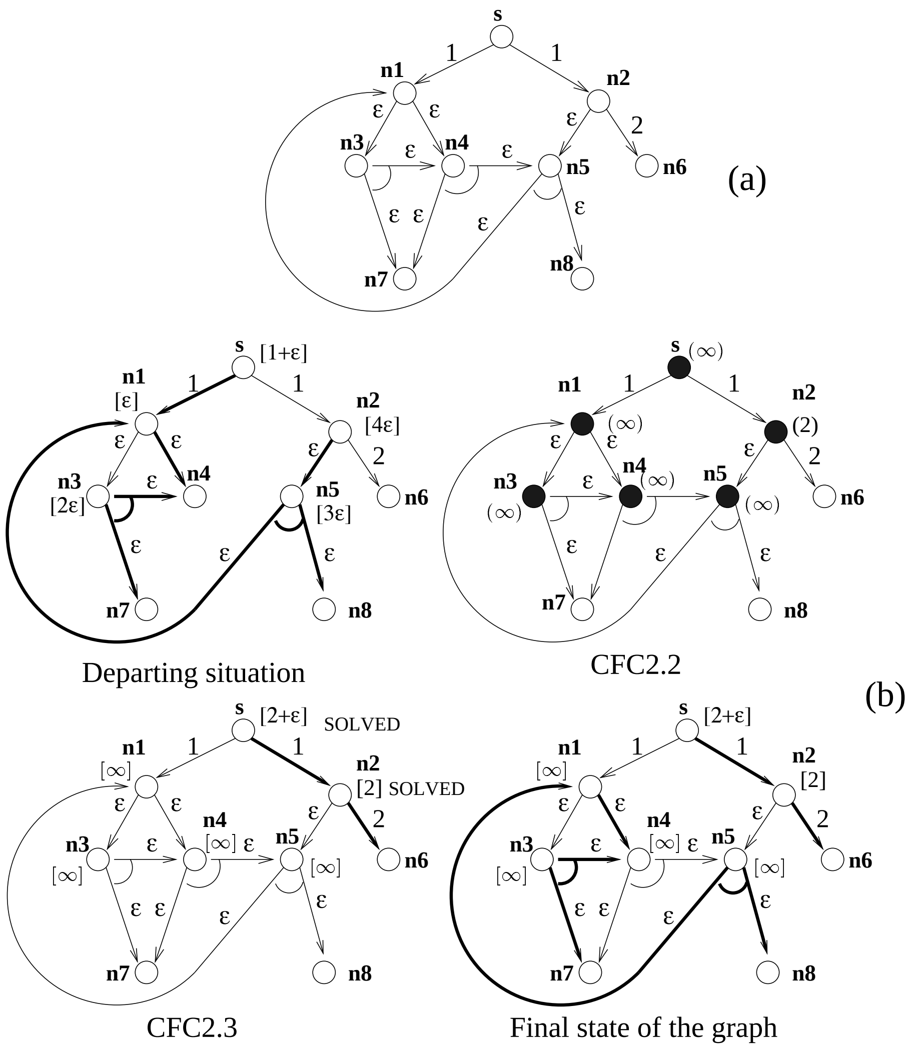

Figure 5: *(a) Example taken from [6]. ϵ denotes an infinitesimally small value. (b) Behaviour of the CFC_{REV^*} algorithm during one iteration of loop CFC2. The cost and arc marking changes performed at steps CFC2.2 and CFC2.3 are displayed. Only two cost calculations are performed, that of node n_2 at step CFC2.2 and that of node s at step CFC2.3.*

$Z(j)$ is the contents of Z after the execution of step CFC2.2 (or, in its case, INT2.2) following instant j. It consists of the expanded node as well as all its ancestors along marked arcs. Reference to the algorithm is not included in this case because it is not needed in the proofs below.

$Z_{\text{fin}}(j)$ is the contents of Z when leaving the while loop of step CFC2.3 following instant j.

$f_{CFC}(m,j)$ (resp. $f_{INT}(m,j)$) denotes the value of $f(m)$ at instant j in the execution of CFC_{REV^*} (resp. *INT*).

In what follows we assume that the selection of a tip node for expansion obeys the same criterion in the two algorithms.

The following lemma guarantees that the cost updating performed at each iteration of CFC_{REV^*} is the same as that produced by the corresponding iteration of *INT*.

Lemma 5.1. If $G'_{CFC}(j) = G'_{INT}(j)$, then $f_{CFC}(m,j+1) = f_{INT}(m,j+1), \forall m \in G'(j)$.

Proof. If $m \in G'(j) \setminus Z(j)$, then neither CFC_{REV^*} nor *INT* update $f(m)$, and the result follows.

We next prove that the result holds for nodes $m \in Z(j)$ that are declared "found" by CFC_{REV^*}. Let m_1 be the first such node. Necessarily m_1 is the first node extracted from O and, thus, it has a solution subgraph disjoint with $Z(j)$ of minimum cost among all those rooted at nodes from $Z(j)$. Therefore, $f(m_1)$ holds the minimum cost estimate of m_1 within $G'(j)$, i.e., $f_{INT}(m_1, j+1)$ (note that this estimate may differ or not from that in the preceding iteration).

Suppose, as induction hypothesis, that the result holds for all nodes "found" up to a given point. There are four places within CFC_{REV^*} where the next node m_i may be declared "found":

1. First self-call within the cost-propagation procedure: m_i has all its children "found" and the result is obvious.

2. Second self-call within the same procedure: m_i is an OR node with a "found" child that grants to it the same cost estimate as in the preceding iteration, thus the result follows trivially.

3. Third self-call within the same procedure: $m_i \in Z(j) \setminus Z_{\text{fin}}(j)$ and, therefore, m_i has been removed from Z by the procedure prune-revisable, meaning that along each maximal path in the marked *psg* below m_i, there exists a node that maintains its cost estimate from the preceding iteration. Thus, $f_{INT}(m_i, j+1)$ will necessarily be equal to $f_{INT}(m_i, j)$, which in turn is equal to $f_{CFC}(m_i, j+1)$, since the cost estimate of m_i is not updated by CFC_{REV^*}.

4. Call within step CFC2.3: m_i is an OR node removed from O at the start of an iteration of the while loop CFC2.3, due to its having the smallest f_{new} value. Since O is initialized with and continuously maintains all "unfound" nodes having a *psg* disjoint with $Z(j)$, this implies that m_i has a solution subgraph disjoint with $Z(j)$ of minimum cost among all those rooted at nodes from $Z(j)$ that remain still "unfound". Therefore, at that time, $f(m_i)$ holds the minimum cost estimate of m_i within $G'(j)$, i.e., $f_{INT}(m_i, j+1)$.

To complete the proof, we have to consider the nodes m that remain "unfound" when leaving the while loop CFC2.3. These nodes do not have a *psg* below them disjoint with $Z(j)$ and, therefore, they are not declared "found" by *INT* either. Thus, it is clear that $f_{CFC}(m,j+1) = f_{INT}(m,j+1) = \infty$. □

Theorem 5.2. If G is finite and $\hat{h}$ is admissible, then the algorithm CFC_{REV^*} terminates by either finding a minimal-cost graph rooted at s or else returning $f(s) = \infty$.

Proof. By induction on the number of iterations of the while loop CFC2, it is proved that $G'_{CFC}(j)$ contains a marked *psg* below s at every instant j, provided $f(s) \neq \infty$. This is true at instant 1. If we assume it to be true at instant j, then Lemma 5.1 together with the fact that the changes in arc markings and SOLVED status are only performed for nodes that have reached their minimum cost estimates, guarantee the result at instant $j+1$. (It is worth noting that $G'_{CFC}(j)$ may differ from $G'_{INT}(j)$, due to the different impact that the use of the preceding cost estimates as lower bounds may have on both algorithms). This proves the correctness of the algorithm.

The completeness of CFC_{REV^*} follows from that of CF. □

8 Efficiency of CFC_{REV^*}

First we present evidence supporting the claim that CFC_{REV^*} is the most efficient algorithm for searching *implicit* AND/OR graphs with cycles, among those described in literature until now. For this, we compare its performance with that of *INT*, the algorithm devised following the indications in [6], which constitutes the most satisfactory available option for the reasons adduced in Section 1. In this way we verify that the modifications we have introduced to the algorithm sketched by Chakrabarti are truly improvements.

Algorithms for searching implicit graphs were originally devised to deal with problems for which the generation of the entire graph is very costly, because of either the large number of nodes involved or the expensive process of generating the successors of a node. However, we have found that, even in cases where the entire graph is available, it is often advantageous to use CFC_{REV^*} instead of an algorithm for searching *explicit graphs*, such as REV^*. Evidence for this is provided in Section 8.2.

8.1 Implicit graphs: comparing CFC_{REV^*} and INT

It is not difficult to see that CFC_{REV^*} and *INT* have an $O(n^3)$ worst-case complexity (n being the number of nodes in the graph), the same complexity as AO^* and CF. This is because, for each node expansion, the number of cost calculations is proportional to the number of edges in the explicit graph [6]. Of course, the complexity can be much lower for particular graph topologies and heuristic functions.

We next show that CFC_{REV^*} is *more efficient* than *INT* in that it visits less nodes and performs less cost computations than *INT*. Between instants j and $j+1$, *INT* visits all nodes in $G'(j)$, while CFC_{REV^*} visits only nodes in $Z(j)$. But the important savings lie in the cost calculations: while *INT* computes the costs of all nodes in $Z(j)$, CFC_{REV^*} computes only the costs of nodes in $Z_{\text{fin}}(j) \cup O(j)$, where $O(j)$ is the set of nodes that are included in O between the two instants. The larger the difference between the respective sets, the larger the savings. This was illustrated in Figure 4, iteration 14.

Note that the savings necessarily include the cost computations for all AND nodes belonging to $Z(j) \setminus Z_{\text{fin}}(j)$, and are likely to include those for most OR nodes in that set as well. The ideal would be that the algorithm revised only nodes whose costs or arc markings change as a result of the expansion of the new node. However, when a node expansion generates a cycle in the marked *psg*, one cannot determine the least costly arc-marking change to open the loop without actually computing the costs of the different alternatives (procedure **init-open**). Therefore, the aforementioned ideal seems unreachable, and CFC_{REV^*} looks as the closest approximation to it.

Both algorithms have been tested on the same set of arbitrary graphs, those depicted in Figure 6 as well as the examples shown in the present article, in order to compare execution times (see Table 1). It can be observed that even for graphs with a low number of nodes, CFC_{REV^*} performs consistently better. The power of the pruning strategies built into CFC_{REV^*} becomes evident in graphs like **k**, **k'**, **s**, and **s'**. In all cases where we have concatenated graphs, the percentage of savings has grown with the concatenation, giving support to the intuitive idea that, for similar graph structures, the CFC_{REV^*} algorithm should become proportionally more advantageous as the size of the graph grows.

It is worth noting that this collection of graphs was designed well before the development of the CFC_{REV^*} algorithm, thus ruling out any bias to favor the particular features of this algorithm. Actually, it was devised to test the correct behaviour of search algorithms on cyclic graphs, not really to compare performances. In principle, graphs with high branching factors together with well-informed heuristics should constitute the most favorable setting for CFC_{REV^*}, a situation quite distant from the graphs in the collection.

8.2 Explicit graphs: comparing CFC_{REV^*} and REV^*

We have performed two sets of experiments. The former corresponds to extreme conditions where CFC_{REV^*} attains savings of up to two orders of magnitude with respect to REV^*, while the latter is aimed at characterizing the frontier between the most advantageous use of one or the other algorithm.

The graphs for the first experiments are based on a disjunctive binary tree with 10 levels, i.e., having 2047 nodes. With a given probability, the nodes in the tree are replaced by AND nodes having three successors: the two regular ones, plus the node's grandfather. In this way, cyclic graphs of a very particular type are generated. Table 2 shows the results. When no AND nodes are included, REV^* performs better than CFC_{REV^*}. As the percentage of AND nodes increases from 10 to 70, the speed factor of CFC_{REV^*} with respect to REV^* passes from 1 to 250, approximately. Note that this is not surprising, since REV^* traverses always the entire graph, whilst CFC_{REV^*} avoids visiting subgraphs that can only be reached by passing through a loop and, in this experiment, every AND node generates a loop.

The results of the second experiment are more important, since they establish the degree of cyclicity and size of the graph, above which it is advantageous to use CFC_{REV^*} instead of REV^*. The graphs in this case are generated following the rules of disassembly sequencing problems: OR nodes have only

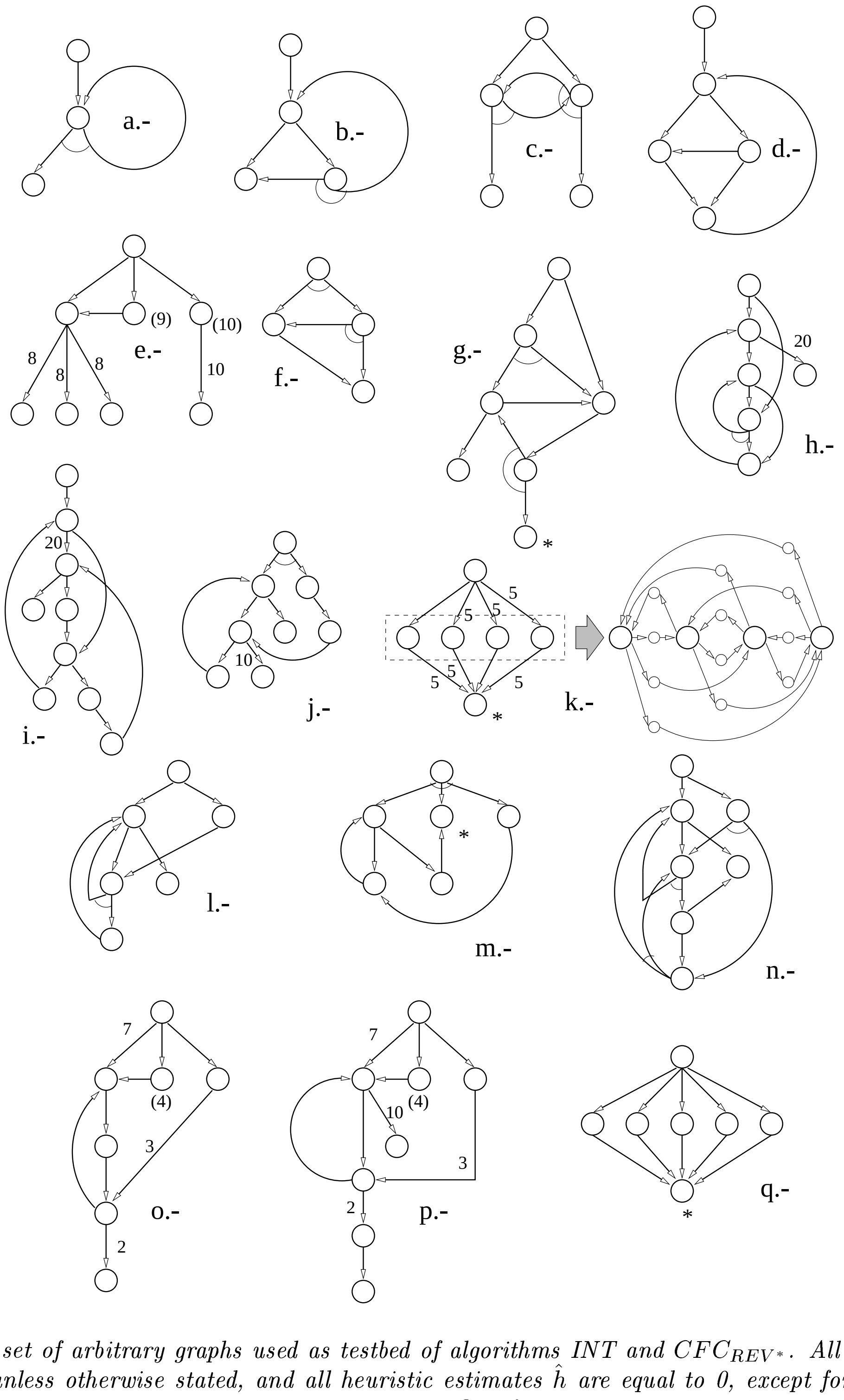

Figure 6: *A set of arbitrary graphs used as testbed of algorithms INT and* CFC_{REV^*}*. All arc costs are equal to 1, unless otherwise stated, and all heuristic estimates* $\hat{h}$ *are equal to 0, except for those nodes where another value is indicated in parentheses. The first four graphs are unsolvable. In graph* **k**, *the four central nodes are completely interconnected through AND nodes (depicted in this case with smaller circles). The cost of the outgoing arc of each one of these AND nodes is equal to 5.*

graph	*INT*	CFC_{REV^*}	*graph*	*INT*	CFC_{REV^*}
a	0.18	0.16	k'	17.85	11.70
b	0.35	0.28	l	0.85	0.73
c	0.39	0.33	m	0.83	0.64
d	0.59	0.58	m'	1.72	0.90
e	0.68	0.45	n	1.06	0.82
f	0.36	0.28	o	1.12	0.92
g	0.88	0.76	p	1.30	1.05
g'	2.45	1.90	q	1.13	0.98
h	0.94	0.77	q'	3.73	2.52
i	1.60	1.42	r	1.25	0.96
j	0.98	0.79	s	4.35	3.10
k	4.30	2.70	s'	6.60	4.90

Table 1: *Execution times of algorithms INT and CFC_{REV^*}, in milliseconds, on a SUN Ultra 2 2200 (SPEC int95 7.81, SPEC fp95 12.9). Graphs* **g'**, **k'**, **m'**, *and* **q'** *result from duplicating* **g**, **k**, **m**, *and* **q**, *respectively, and attaching each duplicate to the corresponding terminal node marked with an asterisk (see Fig. 6). Graph* **r** *corresponds to Figure 5, while* **s** *is the motivating example used throughout the paper, and* **s'** *is the same graph repeated twice, by joining the root of the second to the leftmost leaf of the first.*

Percentage of AND nodes	REV^*	CFC_{REV^*}
0	2076.65	4794.02
10	1451.34	1429.14
30	950.67	96.35
50	671.01	10.15
70	584.38	2.23

Table 2: *Execution times of algorithms REV^* and CFC_{REV^*} for graphs with increasing percentages of AND nodes which, in this particular experiment, correspond to dead-ends. The figures, in milliseconds, are averages over 20 execution runs.*

AND successors and vice versa, and only OR nodes can have more than one parent (see Section 2 and Figure 1(b)). On an underlying binary tree with alternating OR and AND levels, cyclicity is introduced by randomly assigning as successor of an AND node any OR node three levels above it in the tree. Table 3 shows the results of applying REV^* and CFC_{REV^*} to graphs of increasing sizes and with different degrees of cyclicity. Observe that *the higher the degree of cyclicity, the lower the size of the graph above which it is advantageous to use* CFC_{REV^*}. When the probability of generating a backward successor is set to 0.2, CFC_{REV^*} is quicker for graphs with more than 5000 nodes, while for a probability of 0.3, it is advantageous above 300 nodes, approximately. It is worth noting that the cost of generating the graphs (which would penalize REV^*, as well as any other algorithm for searching explicit graphs) is not included.

	Probability of backward arc				
Depth	0.10	0.20	0.25	0.30	*Algorithm*
	1.17	1.30	1.38	1.26	CFC_{REV^*}
4	0.43	0.38	0.34	0.39	REV^*
	[23,2]	[22,4]	[22,7]	[21,6]	
	2.37	2.12	2.74	2.24	CFC_{REV^*}
5	0.92	0.83	0.81	0.84	REV^*
	[46,7]	[45,11]	[44,14]	[45,16]	
	4.59	3.93	4.63	3.64	CFC_{REV^*}
6	2.62	2.00	2.02	1.34	REV^*
	[101,13]	[93,27]	[93,28]	[86,38]	
	11.65	6.57	8.16	5.53	CFC_{REV^*}
7	4.65	4.11	4.07	3.59	REV^*
	[189,21]	[181,48]	[182,61]	[174,77]	
	26.25	13.32	10.31	6.77	CFC_{REV^*}
8	12.37	9.87	8.34	7.44	REV^*
	[376,47]	[368,100]	[361,125]	[355,151]	
	62.86	24.32	17.44	16.15	CFC_{REV^*}
9	22.47	20.07	19.74	17.22	REV^*
	[757,98]	[731,206]	[723,249]	[705,312]	
	203.43	52.65	40.85	21.73	CFC_{REV^*}
10	59.61	48.81	42.61	37.04	REV^*
	[1523,200]	[1471,410]	[1445,515]	[1443,605]	
	461.10	113.70	90.04	64.71	CFC_{REV^*}
11	111.36	101.32	97.55	91.10	REV^*
	[3011,408]	[2927,808]	[2881,1022]	[2841,1217]	
	1519.69	236.91	108.20	65.18	CFC_{REV^*}
12	320.42	256.66	229.85	196.70	REV^*
	[6085,808]	[5871,1635]	[5771,2054]	[5657,2459]	

Table 3: *Execution times of algorithms* REV^* *and* CFC_{REV^*} *for graphs representing disassembly sequencing problems with increasing degrees of cyclicity. Columns are labelled by the probability that an arc stemming from an AND node goes three levels above it in the underlying binary tree structure. Rows, representing increasing graph sizes, are labelled by the depth of the underlying tree structure. The main figures, in milliseconds, are averages over 20 execution runs, while the average size of the graph and its average number of backward arcs are included as pairs between brackets.*

9 Conclusions

Two approaches had previously been proposed to find the optimal solution of implicit AND/OR graphs containing cycles [6, 13], but none of them was designed having efficiency in mind. The computational cost of the algorithm in [13] depends not only on the size of the graph, but also on the costs of the arcs, it being unnecessarily high in the case of low-cost arc cycles. In [6], only some indications on how to use REV^* within an AO^* algorithm were provided.

We have followed these indications to come up with the *INT* algorithm presented in Section 5. Then, we have introduced some modifications into this algorithm in order to save as many node visits and cost computations as possible. This has resulted in the CFC_{REV^*} algorithm, which has been shown to be more efficient than *INT* and, for high degrees of cyclicity, more efficient than REV^* too.

The executable C-code for the algorithm is available, together with a brief user's manual, at the address http://www-iri.upc.es/people/jimenez/CANDOR.html. The input to be supplied is a description of the implicit graph, whereas the output provided by the algorithm consists of a description of the solution subgraph, together with the optimal costs associated to its nodes. Up to our knowledge, this is the first available implementation of an algorithm for solving implicit AND/OR graphs with cycles.

Concerning future work, we will use the algorithm to plan disassembly sequences. Some assembly complexity measures can be directly casted in our sumcost formulation, as sketched in Section 2, while others will require an extension of that formulation to other more general (monotone) cost functions. Likewise, the admissibility assumption may prove to be too restrictive for some of the above measures, and then the generalization to the case of cycles of the results in [17, 5] concerning solution quality and efficiency when heuristics overestimate will have to be undertaken.

Acknowledgements

This work has been partially supported by the Spanish Science and Technology Commission (CICYT) under contracts TIC96-0721-C02-01 (project "Analysis of spatial constraints and its application to mechanical design and robotic task simulation") and TAP99-1086-C03-01 (project "Constraint-based computation in robotics and resource allocation"). The authors would like to thank F. Thomas and P. Meseguer for their very helpful comments on a previous version of this paper. Moreover, the influence of two anonymous referees on the way this work has finally developed is gladly acknowledged.

References

[1] Arenales, M., and Morabito, R. An and/or-graph approach to the solution of two-dimensional non-guillotine cutting problems. *European Journal of Operational Research 84* (1995), 599–617.

[2] Bagchi, A., and Mahanti, A. Admissible heuristic search in and/or graphs. *Theoret. Comput. Sci.*, 24 (1983), 207–219.

[3] Barnett, J. A., and Verma, T. Intelligent reliability analysis. In *Proc. Tenth IEEE Conf. on Artificial Intelligence for Applications* (San Antonio (TX), 1994), pp. 428–433.

[4] Cao, T., and Sanderson, A. C. And/or net representation for robotic task sequence planning. *IEEE Trans. on Systems, Man, and Cybernetics–Part C: Applications and Reviews 28*, 2 (May 1998), 204–218.

[5] Chakrabarti, P., Ghose, S., and DeSarkar, S. Admissibility of ao* when heuristics overestimate. *Artificial Intelligence 34* (1988), 97–113.

[6] Chakrabarti, P. P. Algorithms for searching explicit and/or graphs and their applications to problem reduction search. *Artificial Intelligence 65*, 2 (1994), 329–345.

[7] Chang, C. L., and Slagle, J. R. An admissible and optimal algorithm for searching and/or graphs. *Artificial Intelligence 2* (1971), 117–128.

[8] DeMello, L. S. H., and Sanderson, A. C. A correct and complete algorithm for the generation of mechanical assembly sequences. *IEEE Trans. of Robotics and Automation 7*, 2 (Apr. 1991), 228–240.

[9] Goldwasser, M. *Complexity measures for assembly sequences.* PhD thesis, Stanford University, 1997.

[10] Goldwasser, M., Latombe, J.-C., and Motwani, R. Complexity measures for assembly sequences. In *Proc. IEEE Int. Conf. on Robotics and Automation* (Minneapolis (MN), Apr. 1996), vol. 2, pp. 1851–1857.

[11] Goldwasser, M., and Motwani, R. Intractability of assembly sequencing: unit disks in the plane. In *Proc. of the Workshop on Algorithms and Data Structures, LNCS (Springer Verlag)* (1997), vol. 1272, pp. 307–320.

[12] Hvalica, D. On the cost of potential solution subgraphs. In *Proc. of the 3rd Symp. on Operations Research in Slovenia* (1995), pp. 81–88.

[13] Hvalica, D. Best first search algorithm in and/or graphs with cycles. *Journal of Algorithms 21* (1996), 102–110.

[14] Jiménez, P., Thomas, F., and Torras, C. Collision detection algorithms for motion planning. In *Robot Motion Planning and Control, Jean-Paul Laumond ed., LNCS (Springer Verlag)* (1998), vol. 229, pp. 305–343.

[15] Jiménez, P., and Torras, C. Speeding up interference detection between polyhedra. In *Proc. IEEE Int. Conf. on Robotics and Automation* (Minneapolis (MN), Apr. 1996), vol. 2, pp. 1485–1492.

[16] Kumar, V. A general heuristic bottom-up procedure for searching and/or graphs. *Inform. Sci. 56* (1991), 39–57.

[17] Mahanti, A., and Bagchi, A. And/or graph heuristic search methods. *J. Assoc. Comput. Mach. 32*, 1 (1985), 28–51.

[18] Martelli, A., and Montanari, U. Optimizing decision trees through heuristically guided search. *Comm. ACM 21*, 12 (1978), 1025–1039.

[19] Nilsson, N. J. *Principles of Artificial Intelligence.* Tioga, 1980.

[20] Thomas, F., and Torras, C. Inferring feasible assemblies from spatial constraints. *IEEE Trans. on Robotics and Automation 8*, 2 (Apr. 1992), 228–239.

[21] Wilson, R., and Latombe, J.-C. Geometric reasoning about mechanical assembly. *Artificial Intelligence 71*, 2 (1994), 371–396.