

# Termination Problem of the APO Algorithm

Tal Grinshpoun, Moshe Zazon, Maxim Binshtok, and Amnon Meisels

Department of Computer Science  
Ben-Gurion University of the Negev  
Beer-Sheva, Israel

**Abstract.** Asynchronous Partial Overlay (APO) is a search algorithm that uses cooperative mediation to solve Distributed Constraint Satisfaction Problems (DisCSPs). The algorithm partitions the search into different subproblems of the DisCSP. The proof of completeness of the APO algorithm is based on the growth of the size of the subproblems. The present paper presents an example DisCSP and a detailed run of APO on the example. In the resulting scenario, the run of the algorithm enters an infinite loop. The presented example and scenario contradict the termination and consequently the completeness of the APO algorithm. A correction to the problem that prevents the infinite loop in our example is proposed. A reference to the problematic part in the proof of APO's completeness is also given.

## 1 Introduction

Asynchronous Partial Overlay (APO) [4, 3] is an algorithm for solving Distributed Constraint Satisfaction Problems (DisCSPs) that uses the concept of mediation to centralize the search procedure in parts of the DisCSP. The APO algorithm belongs to a family of DisCSP search algorithms that is radically different than distributed backtracking algorithms. Distributed backtracking, which forms the majority of DisCSP algorithms, can take many forms. Asynchronous Backtracking (ABT) [7], Asynchronous Forward-Checking (AFC) [5], and Concurrent Dynamic Backtracking (ConcDB) [8] are representative examples of the family of distributed backtracking algorithms. All of these algorithms maintain one or more partial solutions of the DisCSP and attempt to extend the partial solution into a complete one. The extension of partial solutions can be performed asynchronously (as in ABT [7]) or concurrently over multiple solutions (as in ConcDB [8]).

The uniqueness of APO lies in its basic operation of merging partial solutions into a complete one. Merging of different partial solutions is distinct from extending partial solutions. To our best knowledge, Descending Requirement Search (DesRS) [6], is the only search algorithm, beside APO, that merges partial solutions. However, the DesRS algorithm is based on a hierarchical partition of the DisCSP. The imposed hierarchical structure guarantees the correctness of DesRS.

The APO algorithm partitions the agents into groups that attempt to find consistent partial solutions. The partition mechanism is dynamic during search

and enables a dynamic change of groups. The key factor in the termination (and consequently the completeness) of the APO algorithm as presented in the correctness proof in [4] is the growth of initially partitioned group during search. This growth is possible, since the subproblems overlap, allowing agents to increase the size of the subproblems they solve. The proof argues that a subset of agents could cycle infinitely through their allowable values without reaching a solution, if the size of that subset does not increase over time.

The proof of APO's completeness in [4] relies on the assumption that a set of agents  $V' \subseteq V$  cannot enter such a state of oscillation. This assumption is considered to be true, since after a mediation session, at least one conflict must be created or must remain. Otherwise, the oscillation would stop and the problem would be solved [4].

The present paper constructs a scenario in which concurrent mediation sessions do not have any remaining conflicts, nor are they aware of any new conflicts created. The solution of these sessions creates new conflicts that become known only after the mediation sessions are over. Thus, the size of the subproblems does not increase. Moreover, the demonstrated scenario leads to an infinite loop of the APO algorithm's run. Such a scenario contradicts the termination and consequently the completeness of APO.

A simple DisCSP with 3 or 4 agents is not enough to achieve such an infinite loop scenario, since we must have at least two concurrent mediation sessions. Consequently, we use a DisCSP with 8 agents and a symmetrical constraint graph. Following the asynchronous nature of the algorithm, some of the messages in our scenario are delayed. All delays are finite.

In the rest of this paper, we briefly describe the APO algorithm accompanied by its pseudo-code as presented in [4] (section 2). We then present our infinite loop scenario in detail (section 3). A proposed correction to the problem is described in section 4. Section 5 presents a discussion.

## 2 Asynchronous Partial Overlay

Asynchronous Partial Overlay (APO) is an algorithm for solving DisCSPs that applies cooperative mediation. The pseudo-code in figures 1, 2, and 3 follows closely the presentation of APO in [4].

Using mediation, agents can solve subproblems of the DisCSP by conducting an internal Branch and Bound search. For a complete solution of the DisCSP, the solutions of the subproblems must be compatible. When solutions of overlapping subproblems have conflicts, the solving agents increase the size of the subproblems that they work on. The original paper uses the term *preferences* to describe potential conflicts between solutions of overlapping subproblems. The present paper uses the term *external constraints* to describe such conflicts. A detailed description of the APO algorithm can be found in [4], the source of the complete version of APO that is used in the present paper.

```

procedure initialize
   $d_i \leftarrow \text{random } d \in D_i;$ 
   $p_i \leftarrow \text{sizeof}(\text{neighbors}) + 1;$ 
   $m_i \leftarrow \text{true};$ 
   $\text{mediate} \leftarrow \text{false};$ 
  add  $x_i$  to the good_list;
  send (init,  $(x_i, p_i, d_i, m_i, D_i, C_i)$ ) to neighbors;
  initList  $\leftarrow$  neighbors;
end initialize;

when received (init,  $(x_j, p_j, d_j, m_j, D_j, C_j)$ ) do
  add  $(x_j, p_j, d_j, m_j, D_j, C_j)$  to agent_view;
  if  $x_j$  is a neighbor of some  $x_k \in \text{good\_list}$  do
    add  $x_j$  to the good_list;
    add all  $x_l \in \text{agent\_view} \wedge x_l \notin \text{good\_list}$ 
      that can now be connected to the good_list;
     $p_i \leftarrow \text{sizeof}(\text{good\_list});$ 
  end if;
  if  $x_j \notin \text{initList}$  do
    send (init,  $(x_i, p_i, d_i, m_i, D_i, C_i)$ ) to  $x_j$ ;
  else
    remove  $x_j$  from initList;
  end if;
  check_agent_view;
end do;

when received (ok?,  $(x_j, p_j, d_j, m_j)$ ) do
  update agent_view with  $(x_j, p_j, d_j, m_j)$ ;
  check_agent_view;
end do;

procedure check_agent_view
  if initList  $\neq \emptyset$  or  $\text{mediate} \neq \text{false}$  do
    return;
   $m'_i \leftarrow \text{hasConflict}(x_i);$ 
  if  $m'_i$  and  $\neg \exists j (p_j > p_i \wedge m_j == \text{true})$  do
    if  $\exists (d'_i \in D_i) (d'_i \cup \text{agent\_view} \text{ does not conflict})$ 
      and  $d_i$  conflicts exclusively with lower priority neighbors do
         $d_i \leftarrow d'_i;$ 
        send (ok?,  $(x_i, p_i, d_i, m_i)$ ) to all  $x_j \in \text{agent\_view}$ ;
      else
        do mediate;
      end if;
    else if  $m_i \neq m'_i$  do
       $m_i \leftarrow m'_i;$ 
      send (ok?,  $(x_i, p_i, d_i, m_i)$ ) to all  $x_j \in \text{agent\_view}$ ;
    end if;
  end check_agent_view;

```

**Fig. 1.** APO procedures for initialization and local resolution.

```

procedure mediate
  preferences  $\leftarrow \emptyset$ ;
  counter  $\leftarrow 0$ ;
  for each  $x_j \in good\_list$  do
    send (evaluate?, ( $x_i, p_i$ )) to  $x_j$ ;
    counter++;
  end do;
  mediate  $\leftarrow$  true;
end mediate;

when received (wait!, ( $x_j, p_j$ )) do
  update agent_view with ( $x_j, p_j$ );
  counter--;
  if counter == 0 do choose_solution;
end do;

when received (evaluate!, ( $x_j, p_j, labeled D_j$ )) do
  record ( $x_j, labeled D_j$ ) in preferences;
  update agent_view with ( $x_j, p_j$ );
  counter--;
  if counter == 0 do choose_solution;
end do;

procedure choose_solution
  select a solution  $s$  using a Branch and Bound search that:
  1. satisfies the constraints between agents in the good_list
  2. minimizes the violations for agents outside of the session
  if  $\neg \exists s$  that satisfies the constraints do
    broadcast no solution;
  for each  $x_j \in agent\_view$  do
    if  $x_j \in preferences$  do
      if  $d'_j \in s$  violates an  $x_k$  and  $x_k \notin agent\_view$  do
        send (init, ( $x_i, p_i, d_i, m_i, D_i, C_i$ )) to  $x_k$ ;
        add  $x_k$  to initList;
      end if;
      send (accept!, ( $d'_j, x_i, p_i, d_i, m_i$ )) to  $x_j$ ;
      update agent_view for  $x_j$ ;
    else
      send (ok?, ( $x_i, p_i, d_i, m_i$ )) to  $x_j$ ;
    end if;
  end do;
  mediate  $\leftarrow$  false;
  check_agent_view;
end choose_solution;

```

**Fig. 2.** Procedures for mediating an APO session and for choosing a solution during an APO mediation.

```

when received (evaluate?, (xj, pj)) do
  mj ← true;
  if mediate == true or ∃k(pk > pj ∧ mk == true) do
    send (wait!, (xi, pi));
  else
    mediate ← true;
    label each d ∈ Di with the names of the agents
      that would be violated by setting di ← d;
    send (evaluate!, (xi, pi, labeled Di));
  end if;
end do;

when received (accept!, (d, xj, pj, dj, mj)) do
  di ← d;
  mediate ← false;
  send (ok?, (xi, pi, di, mi)) to all xj ∈ agent_view;
  update agent_view with (xj, pj, dj, mj);
  check_agent_view;
end do;

```

**Fig. 3.** Procedures for receiving an APO session.

### 3 An infinite loop scenario

Consider the 3-coloring problem presented in figure 4 by the solid lines. Each agent can assign one of the three available colors Red, Green, or Blue. To the standard inequality constraints that the solid lines represent, we add four weaker constraints (diagonal dashed lines) that do not allow only the combinations (Green,Green) and (Blue,Blue) to be assigned by the agents.

The initial selection of values by all agents is depicted in figure 4. In the initial state, two constraints are violated – (A1,A2) and (A5,A6). Assume that agents A3, A4, A7, and A8 are the first to complete their initialization phase by exchanging *init* messages with all their neighbors (procedure **initialize** in figure 1). These agents do not have conflicts, therefore they set  $m_i \leftarrow false$  and send *ok?* messages to their neighbors when each of them runs the **check\_agent\_view** procedure (see figure 1). After the arrival of the *ok?* messages, agents A1, A2, A5, and A6 accept *init* messages from all of their neighbors and complete the initialization phase. Agents A2 and A6 have conflicts, however they complete the **check\_agent\_view** procedure without mediating or changing their state. This is true, because in the agent views of A2 and A6,  $m_1 = true$  and  $m_5 = true$ , respectively. These neighbors have higher priority over agents A2 and A6. We denote by *configuration 1* the states of all the agents at this point of the processing and present the configuration in Table 1.

After all agents complete their initializations, agents A1 and A5 detect that they have conflicts, and that they have no neighbor with a higher priority that

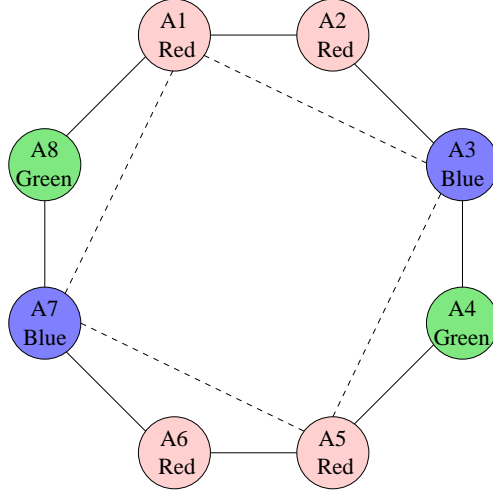


Fig. 4. The constraints graph with the initial setting.

Agent	Color	$m_i$	$d_j$ values	$m_j$ values
A1	R	$m_1 = t$	$d_2 = R, d_3 = B, d_7 = B, d_8 = G$	$m_2 = t, m_3 = f, m_7 = f, m_8 = f$
A2	R	$m_2 = t$	$d_1 = R, d_3 = B$	$m_1 = t, m_3 = f$
A3	B	$m_3 = f$	$d_1 = R, d_2 = R, d_4 = G, d_5 = R$	$m_1 = t, m_2 = t, m_4 = f, m_5 = t$
A4	G	$m_4 = f$	$d_3 = B, d_5 = R$	$m_3 = f, m_5 = t$
A5	R	$m_5 = t$	$d_3 = B, d_4 = G, d_6 = R, d_7 = B$	$m_3 = f, m_4 = f, m_6 = t, m_7 = f$
A6	R	$m_6 = t$	$d_5 = R, d_7 = B$	$m_5 = t, m_7 = f$
A7	B	$m_7 = f$	$d_1 = R, d_5 = R, d_6 = R, d_8 = G$	$m_1 = t, m_5 = t, m_6 = t, m_8 = f$
A8	G	$m_8 = f$	$d_1 = R, d_7 = B$	$m_1 = t, m_7 = f$

Table 1. Configuration 1.

wants to mediate. Consequently, agents A1 and A5 start mediation sessions, since they cannot change their own color to a consistent state with their neighbors.

We will first observe A1’s mediation session. A1 sends *evaluate?* messages to its neighbors A2, A3, A7, and A8 (procedure **mediate** in figure 2). All these agents reply with *evaluate!* messages (see code in figure 3). A1 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A1, A2, A3, A7, and A8, and also minimizes external constraints (procedure **choose\_solution** in figure 2). In our example, A1 finds the solution (A1←Green, A2←Blue, A3←Red, A7←Blue, A8←Red), which satisfies the internal constraints, and minimizes to zero the external constraints. A1 sends *accept!* messages to its neighbors, informing them of its solution. A2, A3, A7, and A8 receive the *accept!* messages and send *ok?* messages with their new states to their neighbors (see code in figure 3). However, the *ok?* messages from A8 to A7 and from A3 to A4 and to A5 are delayed.

Agent	Color	$m_i$	$d_j$ values	$m_j$ values
A1	G	$m_1 = f$	$d_2 = \mathbf{B}$ , $d_3 = \mathbf{R}$ , $d_7 = \mathbf{B}$ , $d_8 = \mathbf{R}$	$m_2 = f$ , $m_3 = f$ , $m_7 = f$ , $m_8 = f$
A2	B	$m_2 = f$	$d_1 = \mathbf{G}$ , $d_3 = \mathbf{R}$	$m_1 = f$ , $m_3 = f$
A3	R	$m_3 = f$	$d_1 = \mathbf{G}$ , $d_2 = \mathbf{B}$ , $d_4 = \mathbf{G}$ , $d_5 = \mathbf{R}$	$m_1 = f$ , $m_2 = f$ , $m_4 = f$ , $m_5 = t$
A4	G	$m_4 = f$	$d_3 = \mathbf{B}$ , $d_5 = \mathbf{R}$	$m_3 = f$ , $m_5 = t$
A5	R	$m_5 = t$	$d_3 = \mathbf{B}$ , $d_4 = \mathbf{G}$ , $d_6 = \mathbf{R}$ , $d_7 = \mathbf{B}$	$m_3 = f$ , $m_4 = f$ , $m_6 = t$ , $m_7 = f$
A6	R	$m_6 = t$	$d_5 = \mathbf{R}$ , $d_7 = \mathbf{B}$	$m_5 = t$ , $m_7 = f$
A7	B	$m_7 = f$	$d_1 = \mathbf{G}$ , $d_5 = \mathbf{R}$ , $d_6 = \mathbf{R}$ , $d_8 = \mathbf{G}$	$m_1 = f$ , $m_5 = t$ , $m_6 = t$ , $m_8 = f$
A8	R	$m_8 = f$	$d_1 = \mathbf{G}$ , $d_7 = \mathbf{B}$	$m_1 = f$ , $m_7 = f$

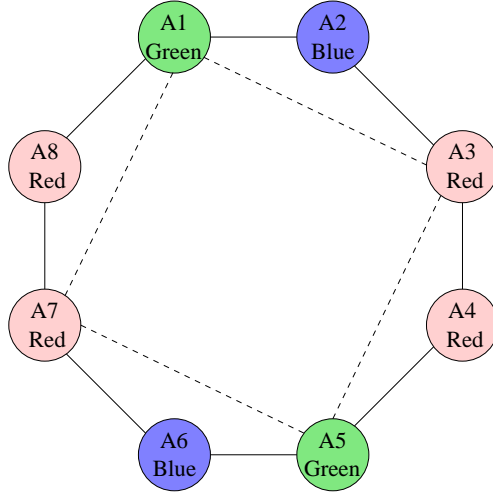
**Table 2.** Configuration 2 – obsolete data in *agent\_views* is in bold face.

Concurrently with the above mediation session of A1, agent A5 starts its own mediation session. A5 sends *evaluate?* messages to its neighbors A3, A4, A6, and A7. Let us assume that the message to A7 is delayed. A4 and A6 receive the *evaluate?* messages and reply with *evaluate!*, since they do not know any agents of higher priority than A5 that want to mediate. A3, is in A1’s mediation session, so it replies with *wait!*. We denote by *configuration 2* the states of all the agents at this point of the processing (see Table 2).

Only then, after A1’s mediation session is over, A7 receives the delayed *evaluate?* message from A5. Since A7 is no longer in a mediation session, nor does it expect a mediation session from a node of higher priority than A5 (see A7’s view in Table 2), agent A7 replies with *evaluate!*. Notice that A7’s view of  $d_8$  is obsolete (the *ok?* message from A8 to A7 is still delayed). When agent A5 receives the *evaluate!* message from A7, it can continue the mediation session involving agents A4, A5, A6, and A7. Since the *ok?* messages from A3 to A4 and A5 are also delayed, agent A5 starts its mediation session with knowledge about agents A3 and A8 that is not updated (see bold-faced data in Table 2).

Agent A5 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A4, A5, A6, and A7, that also minimizes external constraints. In our example, A5 finds the solution (A4←Red, A5←Green, A6←Blue, A7←Red), which satisfies the internal constraints, and minimizes to zero the external constraints (remember that A5 has wrong data about the assignments of A3 and A8). A5 sends *accept!* messages to A4, A6, and A7, informing them of its solution. The agents receive these messages and send *ok?* messages with their new states to their neighbors. By now, all the delayed messages get to their destinations, and two constraints are violated – (A3,A4) and (A7,A8). Consequently, agents A3, A4, A7, and A8 want to mediate, whereas agents A1, A2, A5, and A6 do not wish to mediate, since they do not have any conflicts. We denote by *configuration 3* the states of all the agents after A5’s solution has been assigned and all delayed messages arrived at their destinations (see figure 5 and Table 3).

Up until now, we have shown a series of steps that led from *configuration 1* to *configuration 3*. Next, we will show a very similar series of steps that will lead us right back to *configuration 1*.



**Fig. 5.** The graph in configuration 3.

Agent	Color	$m_i$	$d_j$ values	$m_j$ values
A1	G	$m_1 = f$	$d_2 = B, d_3 = R, d_7 = R, d_8 = R$	$m_2 = f, m_3 = t, m_7 = t, m_8 = t$
A2	B	$m_2 = f$	$d_1 = G, d_3 = R$	$m_1 = f, m_3 = t$
A3	R	$m_3 = t$	$d_1 = G, d_2 = B, d_4 = R, d_5 = G$	$m_1 = f, m_2 = f, m_4 = t, m_5 = f$
A4	R	$m_4 = t$	$d_3 = R, d_5 = G$	$m_3 = t, m_5 = f$
A5	G	$m_5 = f$	$d_3 = R, d_4 = R, d_6 = B, d_7 = R$	$m_3 = t, m_4 = t, m_6 = f, m_7 = t$
A6	B	$m_6 = f$	$d_5 = G, d_7 = R$	$m_5 = f, m_7 = t$
A7	R	$m_7 = t$	$d_1 = G, d_5 = G, d_6 = B, d_8 = R$	$m_1 = f, m_5 = f, m_6 = f, m_8 = t$
A8	R	$m_8 = t$	$d_1 = G, d_7 = R$	$m_1 = f, m_7 = t$

**Table 3.** Configuration 3.

Agents A3 and A7 detect that they have conflicts and that they have no neighbor with a higher priority that wants to mediate. Consequently, agents A3 and A7 start mediation sessions, since they cannot change their own color to a consistent state with their neighbors.

We will first observe A3's mediation session. A3 sends *evaluate?* messages to its neighbors A1, A2, A4, and A5. All these agents reply with *evaluate!* messages. A3 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A1, A2, A3, A4, and A5, and also minimizes external constraints. Agent A3 finds the solution (A1←Green, A2←Red, A3←Blue, A4←Green, A5←Red), which satisfies the internal constraints, and minimizes to zero the external constraints. A3 sends *accept!* messages to its neighbors, informing them of its solution. A1, A2, A4, and A5 receive the *accept!* messages and send *ok?* messages with their new states to their neighbors. However, the *ok?* messages from A2 to A1 and from A5 to A6 and to A7 are delayed.

Agent	Color	$m_i$	$d_j$ values	$m_j$ values
A1	G	$m_1 = f$	$d_2 = \mathbf{B}$ , $d_3 = \mathbf{B}$ , $d_7 = \mathbf{R}$ , $d_8 = \mathbf{R}$	$m_2 = f$ , $m_3 = f$ , $m_7 = t$ , $m_8 = t$
A2	R	$m_2 = f$	$d_1 = \mathbf{G}$ , $d_3 = \mathbf{B}$	$m_1 = f$ , $m_3 = f$
A3	B	$m_3 = f$	$d_1 = \mathbf{G}$ , $d_2 = \mathbf{R}$ , $d_4 = \mathbf{G}$ , $d_5 = \mathbf{R}$	$m_1 = f$ , $m_2 = f$ , $m_4 = f$ , $m_5 = f$
A4	G	$m_4 = f$	$d_3 = \mathbf{B}$ , $d_5 = \mathbf{R}$	$m_3 = f$ , $m_5 = f$
A5	R	$m_5 = f$	$d_3 = \mathbf{B}$ , $d_4 = \mathbf{G}$ , $d_6 = \mathbf{B}$ , $d_7 = \mathbf{R}$	$m_3 = f$ , $m_4 = f$ , $m_6 = f$ , $m_7 = t$
A6	B	$m_6 = f$	$d_5 = \mathbf{G}$ , $d_7 = \mathbf{R}$	$m_5 = f$ , $m_7 = t$
A7	B	$m_7 = t$	$d_1 = \mathbf{G}$ , $d_5 = \mathbf{G}$ , $d_6 = \mathbf{B}$ , $d_8 = \mathbf{R}$	$m_1 = f$ , $m_5 = f$ , $m_6 = f$ , $m_8 = t$
A8	R	$m_8 = t$	$d_1 = \mathbf{G}$ , $d_7 = \mathbf{R}$	$m_1 = f$ , $m_7 = t$

**Table 4.** Configuration 4 – obsolete data in *agent\_views* is in bold face.

Concurrently with the above mediation session of A3, agent A7 starts its own mediation session. A7 sends *evaluate?* messages to its neighbors A1, A5, A6, and A8. Let us assume that the message to A1 is delayed. A6 and A8 receive the *evaluate?* messages and reply with *evaluate!*, since they do not know any agents of higher priority than A7 that want to mediate. A5, is in A3’s mediation session, so it replies with *wait!*. We denote by *configuration 4* the states of all the agents at this point of the processing (see Table 4).

Only after A3’s mediation session is over, A1 receives the delayed *evaluate?* message from A7. Since A1 is no longer in a mediation session, nor does it expect a mediation session from a node of higher priority than A7 (see A1’s view in Table 4), agent A1 replies with *evaluate!*. Notice that A1’s view of  $d_2$  is obsolete (the *ok?* message from A2 to A1 is still delayed). When agent A7 receives the *evaluate!* message from A1, it can continue the mediation session involving agents A1, A6, A7, and A8. Since the *ok?* messages from A5 to A6 and A7 are also delayed, agent A7 starts its mediation session with knowledge about agents A2 and A5 that is not updated (see bold-faced data in Table 4).

Agent A7 conducts a Branch and Bound search to find a solution that satisfies all the constraints between A1, A6, A7, and A8, that also minimizes external constraints. In our example, A7 finds the solution (A1←Red, A6←Red, A7←Blue, A8←Green), which satisfies the internal constraints, and minimizes to zero the external constraints (remember that A7 has wrong data about A2 and A5). A7 sends *accept!* messages to A1, A6, and A8, informing them of its solution. The agents receive these messages and send *ok?* messages with their new states to their neighbors. By now, all the delayed messages get to their destination, and two constraints are violated – (A1,A2) and (A5,A6). Consequently, agents A1, A2, A5, and A6 want to mediate, whereas agents A3, A4, A7, and A8 do not wish to mediate, since they do not have any conflicts. Notice that all the agents have returned to the exact states they were in *configuration 1* (see figure 4 and Table 1).

The cycle that we have just shown between *configuration 1* and *configuration 3* can continue indefinitely. This example contradicts the termination and completeness of the APO algorithm.

It should be noted that we did not mention all the messages passed in the running of our example. We mentioned only those messages that were important for the understanding of the examples, since the example was complicated enough. For instance, after agent A5 completes its mediation session (before *configuration 2*), there is some straightforward exchange of messages between agents, before the  $m_j$  values of all the agents are correct (as presented in Table 2).

## 4 Proposed correction to APO

The scenario presented in the previous section becomes possible, due to delayed updating of *agent\_views* after completion of a mediation session. Agent A7 in *configuration 2* and agent A1 in *configuration 4* are the key players in our scenario, but we will focus on agent A7. Since agent A7 (in *configuration 2*) receives A1's **accept!** message before the arrival of A5's **evaluate?** message, agent A7 is ready to engage in a mediation session with agent A5. However, A7's *agent\_view* is not yet updated, since the **ok?** message from agent A8 was delayed. This situation allows A5 to conduct a local search with wrong knowledge of external constraints, leading to a new conflict between agents A7 and A8. Since agent A5 finds a solution with no conflicts, it does not add any new agents to its neighborhood (procedure **choose\_solution** in figure 2). This enables the infinite reoccurrence of our scenario.

A solution to this problem could be obtained if agent A7 would agree to participate in a new mediation session only when its *agent\_view* is updated with all the changes of the previous mediation session. This can be achieved by the mediator sending its entire solution  $s$  in the **accept!** messages, instead of just particular  $d'_j$ 's. The sending of **accept!** in procedure **choose\_solution** can be changed to the following:

$$\text{send } (\mathbf{accept!}, (s, x_i, p_i, d_i, m_i)) \text{ to } x_j;$$

Upon receiving the new **accept!** message, agent  $i$  now updates all the  $d_k$ 's in the received solution  $s$  (accept for  $d_k$ 's that are not in  $i$ 's *agent\_view*). Notice that agent  $i$  still has to send **ok?** messages to its neighbors, since not all of its neighbors were necessarily involved in the mediation session. The new code for receiving an **accept!** message is in figure 6.

Looking back at *configuration 2* from the previous section, we can observe that after the proposed correction, agent A7 will join A5's mediation session only with updated knowledge of A8's color. In such a case, A5 finds in its search a different solution, for example (A4←Green, A5←Red, A6←Green, A7←Blue) that has no external conflicts. The original solution of this step in our scenario (A4←Red, A5←Green, A6←Blue, A7←Red) now imposes a conflict between agents A7 and A8, since A7 is aware of A8's real color. The infinite cycle cannot occur in this scenario, because it relies on the possibility that after A5's mediation session, agents A7 and A8 will share the same color. This possibility is prevented by our proposed correction.

```

when received (accept!, ( $s, x_j, p_j, d_j, m_j$ )) do
  for each  $x_k \in s$  do
    if  $x_k \in agent\_view$  do
      update  $agent\_view$  with ( $x_k, d_k$ );
    end if;
  end do;
   $mediate \leftarrow \text{false}$ ;
  send (ok?, ( $x_i, p_i, d_i, m_i$ )) to all  $x_j \in agent\_view$ ;
  update  $agent\_view$  with ( $x_j, p_j, d_j, m_j$ );
  check\_agent\_view;
end do;

```

**Fig. 6.** The proposed procedure for receiving an **accept!** message.

The above correction involves no additional exchange of messages. It only adds some data (the complete solution  $s$  instead of just  $d'_j$ ) to the **accept!** messages of the original algorithm. This addition has negligible effect on the overall communication scope of the algorithm.

## 5 Discussion

The APO search algorithm is designed as an asynchronous distributed algorithm on DisCSPs. The proof of APO's completeness as presented in [4] relies on the incorrect assumption that the mediator always adds an agent to its *good\_list* in case an external constraint is violated during a mediation session. This paper has presented a detailed example that uses delays in the delivery of messages. In the example, we show how a mediator can find a solution to its subproblem with no external conflicts created according to its *agent\_view*, due to an obsolete view of external constraints in the time of mediation. This solution does however lead to violation of external constraints contrary to the assumption in [4].

In order to focus on the correctness problem discovered, we presented a scenario in which the APO algorithm enters an infinite loop. To solve the specific problem discovered we proposed a correction to the algorithm that prevents this scenario from happening. Our proposed correction adds necessary data synchronization at the end of a mediation session. Although our proposed correction is shown to be essential, it does not help in proving the correctness of the assumption that was mentioned before. Consequently, the termination and completeness of the APO algorithm remain unclear.

Benisch and Sadeh [1, 2] have proposed several new versions of the APO algorithm. In [1] they suggest an alternative way for selecting the mediators by changing the metric used in calculating the priority  $p$  of an agent. It is important to mention this proposed flexibility to the priority, since the proof of soundness of the APO algorithm in [4] relies on the fact that priorities only increase over time. This is not valid when changing the metric used in calculating the priority

as proposed in [1]. Consequently, the proof of soundness of the APO algorithm must be re-considered for different policies of priorities.

In addition to the mediation procedure originally proposed with APO (procedure **choose\_solution** in figure 2), Benisch and Sadeh [2] propose two new mediation procedures – APO-BT and APO-ABT. Both of these versions of the APO algorithm do not include conflict minimization with agents outside of the mediation session. The scenario presented in section 3 can be easily simplified to generate an infinite loop, when conflicts with external agents are not minimized. According to the original publication of the APO algorithm, these versions of the algorithm do not affect the proofs of soundness and completeness. In fact, the proofs in [4] do not refer to the external conflict minimization.

The present paper presents a termination problem of the APO algorithm. The proposed correction solves the specific problem that was presented. However, there still lies a question mark over the completeness of the algorithm. Moreover, the proof of soundness of the algorithm does not seem to be robust enough to deal with different versions of the APO algorithm.

## References

1. Michael Benisch and Norman Sadeh. How (not) to choose mediators for distributed constraint satisfaction. In *Proceedings of LSMAS at AAMAS'05*, 2005.
2. Michael Benisch and Norman Sadeh. Examining dcsp coordination tradeoffs. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1405–1412. ACM, 2006.
3. Roger Mailler and Victor Lesser. Using cooperative mediation to solve distributed constraint satisfaction problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS'04)*, pages 446–453. ACM, 2004.
4. Roger Mailler and Victor Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research (JAIR)*, 25:529–576, 2006.
5. Amnon Meisels and Roie Zivan. Asynchronous forward-checking for distributed csps. In *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
6. Michael Orlov and Amnon Meisels. Descending requirements search for discsps. In *Proceedings of Distributed Constraint Satisfaction Workshop at ECAI-2006*, Riva del Garda, August 2006.
7. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, pages 614–621, 1992.
8. Roie Zivan and Amnon Meisels. Concurrent dynamic backtracking for distributed csps. In *CP-2004*, pages 782–787, Toronto, 2004.