Distributed constraint satisfaction: a literature review

Angel F. Garcia Contreras

August 10, 2015

Abstract

(Pending)

1 Introduction

Decision-making is the process through which a person selects on course of action among multiple alternative scenarios. People carry out decision-making processes as part of their daily lives: selecting which clothes to wear, food to eat, routes to take while driving, among many others. This internal process is driven by *criteria*: weather influences the type and number of garments to wear, food allergies and health concerns affect a person's dietary preferences, and the city traffic makes certain streets and routes more preferable than others.

For these mundane decisions, the process is for the most part simple and straight-forward. However, there are decisions that require additional planning, often aid by tools and techniques. Sometimes these problems have too many or too complex parameters; for example, a college student that considers dropping out of college is influenced by poor academic performance, lack of preparation, or economic troubles, to name a few. All of these factors can and are often influenced by others, such as prior academic achievements, economic situation, employment situation, or even whether the student is a first-generation college student [3, 32].

Some decision making processes multiple participants to agree on a solution. For example, a committee of engineering experts belonging to multiple companies are tasked with designing a new manufacturing standard. Each expert has a set of expected attributes for this standard, which does not necessarily coincide with the other experts', and can be in opposition with them. The committee members must communicate with each other to determine the attributes of the standard, in order to satisfy the requirements of each expert and their respective companies. This communication is complex, and even more so when each expert has knowledge that cannot be shared with the rest of the group, such as trade secrets; the expert has to try to satisfy his requirements without giving away private and sensitive information.

In this category of decisions that require the negotiation of multiple actors, there are problems that use little to no human intervention, such as coordinating communication protocols between computers at multiple locations through a long-distance network. Each computer knows its own parameters, as well as the restrictions / constraints it has on said parameters in relation to the communication channels it shares with its neighboring computers. Each computer, known as an *agent*, has to determine its parameter value, while communicating it to its neighbors and revising this knowledge if that value violates a neighbor's constraints.

In addition to solving the problem itself, this situation introduces additional complications: what information is transmitted between agents? How can coordination be guaranteed? How to make sure a solution can be reached without spending too much time trying to coordinate all the actors? These questions are the concerns of *distributed problem solving*, a research area that focuses on problems spread in a network across multiple decentralized agents that require coordination and communication.

2 Distributed constraint satisfaction

2.1 Constraint satisfaction problem

A *constraint* is an expression that defines a relationship between a set of variables, in the form of a restriction to the possible values of the variables. A *constraint satisfaction problem* (CSP) is a model designed to find any / all value assignments that fulfill a set of constraints [1, 25].

A *solution* of a constraint satisfaction problem is a set of variable values within the corresponding variable domains such that all the constraints are satisfied [1].

Definition 1. A *constraint satisfaction problem* (CSP) contains a set of n variables

$$X = x_1, \dots, x_n$$

with respective domain values

$$D = D_1 \times \dots \times D_n$$

and a set of m constraints $p_k(x_{k1}, \dots, x_{kn})$ where $k \in \{1, \dots m\}$, and p_k being an expression that restricts values on X.

2.2 Distributed constraint satisfaction problem

A constraint satisfaction problem is solved in a centralized way. In a *distributed constraint satisfaction problem* (DisCSP), the elements of a CSP are distributed among multiple agents, so each agent holds only a part of the problem. In order to solve the problem, agents must communicate with each other and coordinate

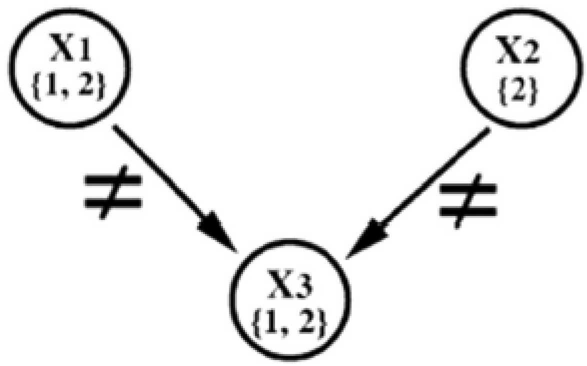

Figure 1: Directed graph of a DisCSP [40]

their efforts to find domain values that satisfy all the constraints of the problem. The general communication model for DisCSP [40] assumes that:

- An agent send messages to other agents, and only to those whose address is known. An agent only knows the addresses of those agents that contain information that the agent needs to solve its part of the problem.
- Messages are transmitted through a network. Due to the nature of the network, there is a random yet finite delay between one agent sending a message, and another receiving it; however, these delay will not cause the messages exchanged between two agents to be out of order, that is all messages arrive in the order in which they are sent.

The definition of the majority of DisCSP algorithms assumes that the CSP program can be represented as a *constraint graph*, with nodes representing agents that contain a single variable, and binary constraints (that is, constraints that involve only two variables) as edges. In this model, each agent knows a variable, as well as the constraints that involve this variable, which means that the constraints are shared between pares of agents. While this definition is common in the literature due to its simplicity, it is not the only model: for some problems, agents can hold more than one variable, have constraints with any number of variables and have private constraints that do not need to be resolved through communication. Unless otherwise noted, the DisCSP algorithms presented in this document use the simple model (one variable per agent, binary constraints) for definition; the relaxation into more a general model is trivial.

Definition 2. A *distributed constraint satisfaction problem* (DisCSP) contains a set of n variables

$$X = x_1, \ldots, x_n$$

with respective domain values

$$D = D_1 \times \ldots \times D_n$$

distributed among m agents, such that each agent assigns the value of 1 or more variables according to a set of o constraints $p_k(x_i, \ldots, x_j)$ where the agent knows some of the variables $x_i, \ldots, x_j$, and each p_k being an expression that restricts values on $x_i, \ldots, x_j$

2.3 Algorithms for DisCSP

2.3.1 Synchronous backtracking

The most intuitive method to solve distributed constraint satisfaction problems is to modify the backtracking algorithms used in non-distributed CSPs. The set of agents has a *total order*, and the first agent initiates the search by selecting a value for its variable, which is added to a *partial solution*. The agent then sends this partial solution to the next agent in the order. Each agent reviews the partial solution and attempts to select a value for its variable that does not violate the constraints. If the agent finds a consistent value, this is added to the partial solution and sent to the next agent in the order; if the agent cannot find an assignment, it *backtracks* the search by sending a *nogood* message to the previous agent. An agent that receives a *nogood* message removes its previous assignment from the partial solution and attempts to find a new consistent value. In general, every agent that receives either a partial solution or a *nogood* message assigns a value consistent with the latest partial solution, backtracking the search only if unable to find such value [43].

While the method is intuitive, it is hardly an effective method. It is no different from the non-distributed backtracking technique, as both *sequentially* attempt to find all assignments that satisfy the constraints. It does not take advantage of having multiple agents that can compute their own values concurrently, and every attempt to find a consistent assignment can result in multiple messages between two or more agents, increasing the communication time [40]. However, the algorithm is still relevant, as it serves as the basis and inspiration for other synchronous algorithms.

Algorithm 1. Synchronous Backtracking

```
main:
   if agent is first_agent
      CPA ← new CPA
      assign_CPA
   while not done
      if msg.type = stop
         stop_search
         done ← true
      if msg.type = backtrack
         remove_last_assignment
         assign_CPA
      if msg.type = CPA
         assign_CPA
assign_CPA:
   CPA ← assign_local_value(msg.CPA)
   if is_consistent(CPA)
      if is_full(CPA)
         return_solution(CPA)
         stop_search
      else
         send(CPA,next_agent)
   else
      do_backtrack
do_backtrack:
   if agent is first_agent
      CPA ← no_solution
      stop_search
   else
      send(backtrack,previous_agent)
stop_search:
   send(stop,all_agents)
   done ← true
```

2.3.2 Synchronous conflict-based backjumping

Maintaining a synchronous approach, the main improvement of synchronous backtracking lies in reducing the number of messages passed between agents. One technique that has proven to be effective is a distributed implementation of conflict-based backjumping [24]. In this improved algorithm, synchronous conflict-based backjumping (SynCBJ), each agent maintains a *conflict set* of previous agents' variables that have caused conflicts (*nogoods*) with the agent's variable. Using this conflict set, the agent who cannot find an assignment can send the *nogood* message directly to the agent responsible for the error [43], along with a partial solution with the conflicting variable values eliminated. The recipient of the *nogood* message reassigns its value and proceeds as usual, with the variable values eliminated from the partial solution being re-added as the search continues, and any inconsistent values are reevaluated. The method has the same lack of parallelism as synchronous backtracking, but its performance is considerably better due to the reduced number of messages [43].

Algorithm 2. Synchronous conflict-based backjumping

```
main:
  if agent is first_agent
    CPA ← new CPA
    assign_CPA
  while not done
    if msg.type = stop
      stop_search
      done ← true
    if msg.type = backtrack
      remove_last_assign'ment
      conflict_set ← (value,msg.conflict_set) ∪ conflict_set
      assign_CPA
    if msg.type = CPA_fwd
      assign_CPA
assign_CPA:
  CPA ← assign_local_value(msg.CPA)
  if is_consistent(CPA)
    if is_full(CPA)
      return_solution(CPA)
      stop_search
    else
      send(CPA_fwd,CPA,next_agent)
  else
    do_backtrack
```

Algorithm 2. Synchronous conflict-based backjumping (cont'd)

do_backtrack:
    if agent is first_agent or (domain is empty and conflict_set is empty)
        CPA ← no_solution
        stop_search
    else
        backjump_agent ← get_closest_prev_agent(conflict_set)
        send(backtrack,conflict_set,backjump_agent)

stop_search:
    send(stop,all_agents)
    done ← true

2.3.3 Asynchronous backtracking

To make the backtracking algorithm asynchronous, that is to enable the agents to assign and revise their values concurrently, the first change lies in the communication model. Asynchronous backtracking (ASB) works by introducing an additional message in addition to the *nogood* from the synchronous version, an *ok?* message that asks for confirmation. Like synchronous backtracking, the agents are also ordered. However, this order only exists to give priority to some agents over others whenever there needs to be a value revision in order to avoid infinite processing loops in which a change in one agent triggers a series of changes in other agents that eventually lead to a change in the original agent. Each agent also maintains an *agent_view* which records the values received by the agent from its neighboring agents. Another important assumption of this algorithm is that all constraints are *directed*, so when an agent assigns its value it sends the assignment as an *ok?* message to an evaluating agent that checks for consistency based on its value and *agent_view* [39, 40].

When the algorithm starts, all agents assign an initial value, communicate these to their respective neighbors in the form of an *ok?* message and wait for incoming messages. Upon receiving a value, the evaluating agent stores it in its *agent_view*, checking the consistency of the *agent_view* against its own value. If it is not consistent, the agent tries to change the value so it is consistent with its *agent_view*. If there is no value that can be consistent with the *agent_view*, the agent sends a *nogood* message to one of the neighbors that originally sent their value [39, 40].

The asynchronous nature of this algorithm introduces other problems: the messages an agent receives may no longer be relevant to the previous *agent_view*. In particular, a *nogood* reply may be received after the agent changed its value in response to a *nogood* from another agent. To correct this, each *nogood* message also includes the context, that is the *agent_view*, in which the *nogood* was generated. The agent receiving the error context compares it to its own value and *agent_view*, initiating a value update only if the value and view are *compatible* with the error context, that is the variables and values stored in both are the same. Additionally, an agent can identify implicit constraints with other agents, when it receives an unknown agent's variable and value inside an error context. With this, the agent can request to create a link/constraint with the previously-unknown agent, thus enabling them to communicate [39, 40].

The algorithm will find a solution when the system becomes stable, that is all the agents no longer need to send any message and are in a waiting state. If there are no solutions, eventually an agent in the set will send a *nogood* with an empty context, which implies that all possible assignments lead to contradictions [39, 40].

This algorithm is one of the cornerstones of the field, becoming the basis for many other algorithms in both DisCSP and distributed constraint optimization. To this day it is still used as a comparison metric in simulations against other newer algorithms.

Algorithm 3. Asynchronous backtracking

```
main:
  agent_value ← assign_new_value
  while not done
    if msg.type = ok?
      agent_view.add(msg.sender_id,msg.sender_value)
      if not consistent(agent_view,agent_value)
        agent_value ← assign_new_value
        if agent_value is null
          do_backtrack
        for each neighbor_agent in outgoing_links
          send(ok?,(agent_id,agent_value),neighbor_agent)
    if msg.type = nogood
      nogood_view = msg.nogood
      for each unknown_agent in nogood_view not in outgoing_links
        request_link(unknown_agent.id, agent_id)
        agent_view.add(unknown_agent.id,unknown_agent.value)
      if not compatible(agent_view,agent_value)
        send(ok?,(agent_id,agent_value),msg.sender_id)
      else
        agent_value ← assign_new_value
        if agent_value is null
          do_backtrack
        for each neighbor_agent in outgoing_links
          send(ok?,(agent_id,agent_value),neighbor_agent)
```

Algorithm 3. Asynchronous backtracking (cont'd)
do_backtrack:
nogoods ← obtain_inconsistencies(agent_view)
if nogoods = null
broadcast_no_solution
done ← true
else
for each inc_assignment in nogoods
id,value ← select_largest_id(inc_assignment)
send(nogood,(agent_id,inc_assignment),id)
agent_view.remove(id,value)

2.3.4 Asynchronous weak-commitment search

An expansion and revision of asynchronous backtracking, the asynchronous weak-commitment search algorithm uses a similar set of message, albeit handled differently and with additional information.

First of all, unlike ASB, agents in weak-commitment search send their variable values to all neighbors, not just the ones with lower priority order. Instead of a static total ordering, all agents keep a priority value that changes dynamically. This non-negative integer is initially set to 0 and sent to all neighbors along with the initial variable value assignment in the first *ok?* message. When two neighbor agents have the same priority, this value is updated according to the identifier of the agents [40, 35, 36].

When an agent receives an *ok?* message from a higher-priority neighbor that is not consistent with its *agent_view*, the agent will attempt to update its value so it is consistent with the higher priority neighbors, and also minimizes constraint violations with lower priority neighbors. If such a value is found, the agent sends an *ok?* message to all its neighbors with the corresponding update [40, 35, 36].

If the agent cannot find a value, it sends *nogood* messages to other agents and increases its priority by 1. However, due to the asynchronous nature of the messages it is possible to receive a repeated *nogood* message from other agents. In order to avoid this repetition, in addition to their *agent_view*, the agents also keep track of all generated and received *nogoods*. If an agent cannot change its value, it checks the list of previously generated *nogoods* and if the current *nogood* has been received before, the agent does not send a new *nogood* message, nor does it update its priority [40, 35, 36].

Algorithm 4. Asynchronous weak-commitmment search

```
main:
  agent_value ← assign_new_value
  while not done
    if msg.type = ok?
      agent_view.add(msg.sender_id,msg.sender_value,msg.sender_priority)
      check_agent_view
    if msg.type = nogood
      nogood_list.add(msg.nogood)
      for each (id,value,priority) in nogood_list where id is not in neighbors
        neighbors.add(id)
        agent_view.add(id,value,priority)
      check_agent_view

check_agent_view:
  if not consistent(agent_view,agent_value)
    new_value ← new_violation_min_value
    if new_value is null
      do_backtrack
    else
      agent_value ← new_value
      for each neighbor_agent in outgoing_links
        send(ok?,(agent_id,agent_value,agent_priority),neighbor_agent)

do_backtrack:
  nogoods ← obtain_inconsistencies(agent_view)
  if nogoods = null
    broadcast_no_solution
    done ← true
  if nogoods ∩ nogood_sent = null
    for each inc_assignment in nogoods
      nogood_sent.add(inc_assignment)
      for each (id,value,priority) in inc_assignment
        send(nogood,(agent_id,inc_assignment),id)
    priority_max ← get_max_p(agent_view)
    agent_priority ← priority_max + 1
    new_value ← new_violation_min_value
    agent_value ← new_value
    for each neighbor_agent in outgoing_links
      send(ok?,(agent_id,agent_value,agent_priority),neighbor_agent)
```

2.3.5 Distributed breakout

The distributed breakout algorithms (DBO) is a family of methods inspired by the *breakout* technique [19], which uses *weights* on the constraints and works from an initial value to try and minimize the violation on the constraints. The breakout technique is a local search algorithm, which means its search is not complete and the solution is just a local minimum [34].

When the algorithm begins, all agents assign an initial value to their variables, send an *ok?* message with their initial value to their neighbor agents and assign a*weight* to each constraint they are involved with. With the information of its neighbors and its own value, each agent calculates a *cost* of the valuation by aggregating the weighted violation of all its constraints. Each agent calculates its *gain* by selecting the maximum possible cost change if the agent were to modify its value, and proceeds to send this gain to its neighbors in the form of an *improve* message. After all agents receive *improve* messages from all its neighbors, each agent compares all neighboring gains to its own, and only if an agent recognizes its gain is the greatest amongst is neighbors, then it updates its value. If two or more agents have the greatest gain in their neighborhood, they update their values concurrently [34, 37].

Using this method, the algorithm can find a local minimum value for the constraint violation of the agents. However, to check if a value assignment is actually a local minimum the agents would have to communicate with all agents, by introducing a time-expensive *global communication* scheme. To avoid this, the distributed breakout algorithm introduces *quasi-local minimum*, defined as a state of the system in which an agent finds violations in its constraints, and neither the agent nor its neighbors can find a lower cost. When an agent finds itself at a quasi-local minimum, it attempts a *quasi-local breakout* operation by increasing the weights of its violated constraints [37].

So far, the algorithm shows how agents update their values and escape quasi-local minima. This process of updating and sharing values and gains is called a *round*, and the algorithm finds improvements by sequentially executing multiple rounds. The final piece of this algorithm is a *termination condition*, a series of steps carried out at the end of a round. Each agent maintains a *counter* initialized to zero. After receiving an *ok?*, if the agent finds constraint violations, it sets its counter to zero, otherwise it keeps the counter from the previous round. Next, all agents share their counter values as part of the *improve* message. Before the end of the turn, each agent updates its value to the minimum counter among its known counters, that is its own counter and its neighbors'. Finally, if neither the agent nor its neighbors have constraint violations, the counter is updated by 1. With this, each agent keeps track of the distance with no violations, and if that distance matches a number that ensures all agents are covered, then the problem is solved and the algorithm stops [37].

Algorithm 5. Distributed breakout

```
main:
  agent_value ← select_random_value
  for each constraint in constr_list
    constraint.weight ← 1
  t_counter ← 0
  round ← 0
  for each constraint in constr_list
    send(ok?,(agent_id,agent_value),constraint.agent)
  while t_counter ¡ upper_bound
    round ← round + 1
    neigh_values ← collect_ok_messages
    if not consistent(neigh_values,agent_value)
      t_counter ← 0
    (local_changes,cost) ← minimize(constr_list,neigh_values)
    for each constraint in constr_list
      send(improve,(agent_id,t_counter,local_changes,cost),constraint.agent)
    neigh_imp ← collect_improve_messages
    t_counter ← min(t_counter,neigh_imp.counters)
    if cost = 0 and contains_zeroes(neigh_imp.costs)
      t_counter ← t_counter + 1
    if is_quasi_local_min(constr_list,neigh_values,cost,neigh_imp.costs)
      for each constraint in constr_lists where constraint.violation > 0
        increase(constraint.weight)
    if not conflicts(local_changes,neigh_imp)
      agent_value ← update_value(local_changes)
    else
      agent_value ← resolve_conflicts(local_changes,neigh_imp)
    for each constraint in constr_list
      send(ok?,(agent_id,agent_value),constraint.agent)
```

2.3.6 Distributed backtracking with sessions

The *distributed backtracking with sessions* algorithm is a modification of asynchronous backtracking that attempts to improve the classic algorithm by reducing the amount of time each agent has to process messages. It has similar elements as ABT: all agents contain a value assignment, a priority and an *agent_view*, and exchange *ok?* and *nogood* messages. In distributed backtracking with sessions, the agents also send a *stop* message when there is no solution to the problem and the agents need to stop all execution. The agents also maintain a *session* value, a set of *proposed* values that have already been transmitted in the current session, a set of *received backtrack* values that includes all values that have elicited a backtrack request in the current session, and a set of *backtrack requests*. The *session* value is also included in the *ok?* and *nogood* messages, as well as the corresponding session values for each neighbor in the *agent_view*. The *nogood* message also includes a *backtrack set* of agents to continue backtracking in case the agent cannot find a value that resolves the constraint valuations [17].

At the beginning of the algorithm, all agents set their *session value* to 0, initialize empty *received backtrack* value set and proceed to assign their values in the same manner as ABT. After initialization, the agents send their first *ok?* messages to lower priority neighbors, recording the sent value in the *proposed* set. When an agent receives an *ok?* message, the current session is *closed* by incrementing the *session* value by 1 and emptying the respective *proposed* and *received backtrack* value sets, and then proceeds to process the message in the same way as ABT. On a *nogood* message, the agent processes the received value only if the *session* value in the *nogood* message is equal to the agent's own *session* value and the current value is not in the set of *received backtracks*; if they are different or the current value has already received a backtrack request, the message is considered to be *obsolete* and is ignored. After processing a *nogood*, the current agent value is added to the set of *received backtracks* and, if the message contains a *backtrack set*, add it to the *backtrack requests*. If the agent cannot change its value, it uses the set of *backtrack requests* to determine where to send a new *nogood* message [17].

Algorithm 6. Distributed backtracking with sessions

```
main:
  agent_value ← assign_new_value
  agent_session ← 0
  while not done
    if msg.type = ok?
      agent_view.add(msg.id,msg.value,msg.session)
      close_session
      check_agent_view
    if msg.type = nogood
      if msg.session = agent_session and not received_bt.contains(msg.value)
        received_bt.add(msg.value)
        total_bt.add(msg.bt_list)
        if msg.value = agent_value
          agent_value ← null
      close_session
      check_agent_view
```

```
Algorithm 6. Distributed backtracking with sessions (cont'd)
close_session:
  agent_value ← null
  agent_session ← agent_session + 1
  received_bt.remove_all
  for all value in agent_domain
    propose[value] ← false
check_agent_view:
  if not consistent(agent_view,agent_value)
    new_value ← select_consistent_val(agent_view,propose)
    if new_value is null
      do_backtrack
    else
      agent_value ← new_value
      propose[value] ← true
      for each neighbor_agent in outgoing_links
        send(ok?,(agent_id,agent_value,agent_session),neighbor_agent)
do_backtrack:
  nogoods ← obtain_inconsistencies(agent_view)
  if nogoods = null
    broadcast_no_solution
    done ← true
  else
    for each inc_assignment in nogoods
      id,value,session,bt_set ← select_current(inc_assignment,agent_session)
      send(nogood,(agent_id,agent_value,agent_session,bt_set),id)
      agent_view.remove(id,value,session)
      total_bt.remove(id,value,session,bt_set)
```

2.3.7 Asynchronous aggregation search

Many DisCSP algorithms work under the assumption that all agents will have a single variable, and that the model uses binary constraints shared between agents. Asynchronous aggregation search (AAS) works with a model that natively supports private (internal) constraints and non-binary constraints, which in turn means that agents can keep track of multiple variables, and some variables might be shared between agents [30, 29].

The first change is the general agent model. A *link* between agents represents two agents that have at least one shared variable. This link is directed, from the agent with lower priority to the agent with higher priority. The *end agents* are those agents with no incoming links. The *system agent* is a special agent that coordinates the entire process by assigning priority values and announce search termination [30, 29].

Each agent in AAS keeps track of proposed assignments, which are tuples representing all the agent's variables. Additionally, an assignment is not a singular value, but an aggregation of all values for all of the variables that are consistent with the agent's constraints. This means that any time an agent sends a message, it sends a list of of valid domains. Thus, the solution to the DisCSP is not given as an evaluation, but as set of domains that contain solutions. After that, the algorithm behaves like a modified version of ABT, with each agent building its set of potential assignments based on the received variables, initiating a backtrack when no assignment can be found, and changing the set of potential assignments by examining the backtrack request, that is a *nogood* message with a set of values that violate the constraints [30, 29].

The other modification of interest is the *termination mechanism.* ABT terminates only when all agents stop receiving and sending messages. AAS introduces an *accepted* message, sent by a recipient of an *ok?* message when the contents of the values sent in the *ok?* do not result in an invalid (empty) domain. The *accepted* is similar to an *ok?* message, only instead of sending the values of the agent, the message includes the *intersection* of the values contained in the received *ok?* message and the agent's own values. When an agent receives all *accepted* messages from its neighbors, the agent has found a *solution* to its subset of the problem. If the agent is an *agent*, it sends its *accepted* message to the *system agent.* Once the *system agent* has received all *accepted* messages from all the *end agents*, the algorithm has found a solution and will stop [30, 29].

The implementation of this algorithm determines its actual efficiency. The "aggregation" part of the algorithm is the proposed assignments, depending on the structure used to store them and the technique used to select the values that are incorporated into the respective aggregations (for the agent's values, as well as the values sent in the *nogood* messages) [30, 29].

Algorithm 7. Asynchronous aggregation search

```
main:
  agent_value ← assign_new_value
  while not done
    if msg.type = ok?
      if history[msg.var].invalidate(msg.hist)
        continue
      agent_view.add(msg.var,msg.value,msg.hist)
      reconsider_nogoods
      check_agent_view
    if msg.type = nogood
      nogood_view = msg.nogood
      agent_view.add(known_agents(nogood_view))
      for each unknown_agent in nogood_view not in outgoing_links
        request_link(unknown_agent.id, agent_id)
        agent_view.add((unknown_agent.values))
      nogood_list.add(nogood_view)
      old_agg ← inst_agg
      check_agent_view
      for all old_a in old_agg and curr_a in inst_agg c
        if old_a = curr_a:
          send(ok?,(var(curr_a),set(curr_a),history[curr_a]),msg.agent_id)
check_agent_view:
  if not is_consistent(agent_view,inst_agg)
    valuation ← select_consistent_agg(curr_sol,agent_view)
  if valuation = null
    do_backtrack
  else
    clean(inst_agg)
    for each agg in valuation
      if need_multicast(agg)
        var ← var(agg)
        counter ← increase(counter)
        history[agg].append(history[var],counter)
        for each neighbor_agent in outgoing_lp_links
          send(ok?,(var,set(agg),history[var]),neighbor_agent)
        inst_agg.add(agg)
      else if needed(agg)
        inst_agg.add(agg)
```

Algorithm 7. Asynchronous aggregation search (cont'd)

```
do_backtrack:
  nogoods ← obtain_inconsistencies(agent_view)
  if nogoods = null
    broadcast_no_solution
    done ← true
  else
    for each inc_assignment in nogoods
      id ← select_lowest_prio(inc_assignment)
      send(nogood,(agent_id,inc_assignment),id)
      agent_view.remove_all_proposals(id)
      reconsider_nogoods
    check_agent_view
```

2.3.8 Asynchronous forward-checking

Asynchronous forward-checking (AFC) is an algorithm that processes partial assignments synchronously, but does consistency checks by *forward-checking* asynchronously. In that respect, it follows a similar process to *synchronous backtracking*, by having each agent do partial assignments that are transmitted to the next agents in the partial order of agents [15].

In this algorithm, agents send a somewhat different set of messages. The first message is *CPA*, which carries the current consistent partial assignment (CPA), sent by an agent that has checked the consistency of the assignments from previous agents in addition to its own value assignment. The assignment also includes a *step counter*, used by the agent in all sent messages as a time stamp, and is increased only when the agent sends the latest CPA to the next agent in the order. The step counter is also kept as part of the agent view, to signify the latest update received. If the agent cannot assign its value and remain consistent with the received *CPA*, it sends a *backtrack* message, which works like the *nogood* in synchronous backtracking, to the previous agent in the order [15].

The second message, *FC_CPA*, is sent by an agent when adding an assignment to the CPA, to all agents that have not yet made an assignment. Through this message, the algorithm checks the consistency of the assignment against all future potential assignments, asynchronously and concurrently detecting solutions and invalid assignments. When an agent receives a *FC_CPA* message, it checks the step counter to see if the message is an updated CPA, or belongs to an older version that has already been processed and can be ignored. If the agent receives an update, the agent first checks the consistency of its agent view; if it's inconsistent, then the agent marks its view as consistent if the received CPA does not contain new changes to the view. After this, the agent updates its view based on the received CPA; however, if this assignment is not possible, that is there is no value that does not violate the constraints, the agent sends a *NotOK* message to all unassigned agents along with its view [15].

This *NotOK* message is used to inform all agents of an inconsistent assignment. The sender includes the *shortest inconsistent subset of assignments* from the *FC_CPA*, and the recipients of the *NotOK* message update their agent views with this subset of assignments if the received message is newer than the previous messages and the agent view contains updatable domains [15].

Algorithm 8. Asynchronous forward-checking

```
main:
  if agent is first_agent
    CPA ← new CPA
    assign_CPA
  while not done
    if msg.type = stop
      done ← true
    if msg.type = FC_CPA
      forward_check
    if msg.type = Not_OK
      process_Not_OK
    if msg.type = CPA or backtrack_CPA
      receive_CPA

assign_CPA:
  CPA ← assign_local_value(msg.CPA)
  if is_assigned(CPA)
    if is_full(CPA)
      return_solution(CPA)
      stop_search
    else
      CPA.step_ctr ← CPA.step_ctr + 1
      send(CPA,next_agent)
      for each un_agent in unassigned_agents
        send(FC_CPA,un_agent)
  else
    agent_view ← shortest_inconsistent_part_assignment
    do_backtrack
```

Algorithm 8. Asynchronous forward-checking (cont'd)

```
do_backtrack:
  if agent is first_agent
    send(stop,all_agents)
    done ← true
  else
    agent_view.consistent ← false
    back_agent ← last(agent_view)
    CPA ← agent_view
    send(backtrack_CPA,back_agent)
receive_CPA:
  CPA ← msg.CPA
  if not agent_view.consistent
    if CPA.contains(agent_view)
      do_backtrack
    else
      agent_view.consistent ← true
  if agent_view.consistent
    if msg.type = backtrack_CPA
      remove_last_assignment
      assign_CPA
    else
      if update_agent_view(CPA)
        assign_CPA
      else
      do_backtrack
forward_check:
  if msg.step_ctr > agent_view.step_ctr
    if not agent_view.consistent
      if not CPA.contains(agent_view)
        agent_view.consistent ← true
    if agent_view.consistent
      if not update_agent_view(FC_CPA)
        for each un_agent in agent_view.unassigned
          send(Not_OK,un_agent)
```

Algorithm 8. Asynchronous forward-checking (cont'd)

```
process_Not_OK:
   if agent_view.contains(Not_OK)
      agent_view ← Not_OK
      agent_view.consistent ← false
   else if not Not_OK.contains(agent_view)
      if msg.step_ctr > agent_view.step_ctr
         agent_view ← Not_OK
         agent_view.consistent ← false
update_agent_view(partial_assignment):
   if adjust_agent_view(partial_assignment) = null
      agent_view ← shortest_inconsistent_part_assignment
      return false
   return true
```

2.3.9 Distributed stochastic search

The family of *distributed stochastic search* (DSA) algorithms works similar to asynchronous backtrack. There are two main differences: there is no backtrack, and all value selection is based on a stochastic process [41].

Initially, all agents concurrently and randomly select an initial value and send it to their neighbors. After sending values, agents receive values from their neighbors and determine whether they change their internal values based on stochastic probabilities and degree of constraint violation (values that result in lower constraint violations are more likely to be selected / kept). The differences from one DSA to another are the strategy used to determine the stochastic probabilities, and the termination conditions used to stop the execution [41].

Algorithm 9. default

```
Distributed stochastic search main:
   agent_value ← select_random_value
   while not done
      if is_new_value(agent_value)          for each neighbor in neighbors_list
            send(agent_value,neighbor)
         new_values ← receive_values
         agent_view.update(new_values)
         check_termination
         update_value(agent_value)
```

2.3.10 Concurrent dynamic backtracking

Concurrent dynamic backtracking (ConcDB) is a search algorithm that seeks to exploit concurrency as much as possible. In this algorithm, all agents process *consistent partial assignments* (CPAs) by assigning their variable values that do not violate constraints with variables that already exist in the CPA. Unlike other distributed algorithms, ConcDB has no priority ordering, so each agent selects the destination of the new CPA randomly from the set of neighbors with unassigned values. If the agent cannot find a value that does not violate the constraints, it *backtracks* the CPA to the original sender [42].

The innovation of this algorithm lies in its *concurrent search*. The *initializing agent* creates 2 or more *search processes* (SPs), assigning different values from its domain to each respective SP, so each process is a search through different subspaces of the domain. The agent, then, sends a CPA message to two randomly selected, different agents, including the SP and a *step counter* into each message [42].

When receiving and sending CPAs, each agent keeps track of which assignments it has made to which SP to process potential backtrack messages, as well as which domain values are removed and why they are removed from the potential assignments to the CPA (an *eliminating explanation*). Additionally, every time the agent sends a CPA, including every time an agent sends one on a backtrack message, the *step counter* is increased by 1. An agent that receives a CPA with a *step counter* greater than a predefined *step limit* has to *split* its domain into 2 or more new *search processes*, just as an *initializing agent* would [42].

If an agent removes all of its potential assignments to the CPA, the it sends a *nogood* message based on the eliminating explanations of that invalid assignment. The recipient of this message selects a new value to assign to the CPA, if able, along with a new SP identifier, creates a new *unsolvable* message that is propagated to the originator of the SP that generated the CPA, and shares the new SP identifier with all agents that had previously processed the CPA with the old SP that was marked as *unsolvable* [42].

Algorithm 10. Concurrent dynamic backtracking

```
main:
   if agent is first_agent
      initialize_SPs
   while not done
      if msg.type = split
         perform_split
      if msg.type = stop
         done ← true
      if msg.type = backtrack or CPA
         receive_CPA
      if msg.type = unsolvable
         mark_unsolvable(msg.SP)
assign_CPA:
   CPA ← assign_local_value(msg.CPA)
   if is_consistent(CPA)
      if is_full(CPA)
         return_solution(CPA)
         stop
      else
         send(CPA,next_agent)
   else
      do_backtrack
do_backtrack:
   origin_SP.split_set.delete(CPA.ID)
   if origin_SP.split_set.is_empty
      if agent is first_agent
         CPA ← no_solution
         if(active_CPAs.is_empty)
            return_solution(null)
            stop
      else
         send(backtrack,inconsistent_assignment,last_assignee)
   else
      mark_fail(CPA)
stop:
   send(stop,all_agents)
   done ← true
```

Algorithm 10. Concurrent dynamic backtracking (cont'd)

```
assign_CPA:
    CPA ← msg.CPA
    if first_received(CPA.ID)
        create_SP(CPA.ID)
    if CPA.generator = agent_id
        CPA.steps ← 0
    else
        CPA.steps ← CPA.steps + 1
        if CPA.steps == steps_limit
            splitter_id ← select_splitter
            CPA.steps ← 0
            send(split,splitter_id)
    if msg.type = backtrack
        remove_last_assignment
    assign_CPA
```

```
perform_split:
    if not_backtracked(CPA)
        var ← select_split_var
        if var ≠ null
            create_split_SP(var)
            create_split_CPA(SP.ID)
            origin_SP.split_set.add(CPA.ID)
            assign_CPA
        else
            send(split,next_agent) initialize_SPs:
    for i ← 1 to domain_size
        CPA ← create_CPA(i)
        SP[i].domain ← first_var[i]
        create_SP(CPA.ID)
        assign_CPA
```

```
mark_unsolvable(SP):
    SP.unsolvable ← true
    send(unsolvable,SP.next_agent)
    for each split in SP.origin.split_set        split.unsolvable ← true
        send(unsolvable,split.next_agent)
```

```
Algorithm 10. Concurrent dynamic backtracking (cont'd)
check_SPs(inc_assignment):
  for each sp in all_SPs where sp ≠ current_SP
    if sp.contains(inc_assignment)
      send(unsolvable,sp.next_agent)
      remove_last_assignment(last_sent_CPA)
      CPA ← last_sent_CPA
      rename_SP(sp)
      assign_CPA
receive_CPA:
  CPA ← msg.CPA
  if msg.SP.unsolvable
    terminate(msg.SP)
  else
    if first_received(CPA.ID)
      create_SP(CPA.ID)
    if CPA.generator = agent_id
      CPA.steps ← 0
    else        CPA.steps ← CPA.steps + 1
    if CPA.steps = steps_limit
      splitter_id ← CPA.generator
      send(split,splitter_id)
    if msg.type = backtrack
      check_SPs(CPA.inconsistent_assignment)
      remove_last_assignment(last_sent_CPA)
      CPA ← last_sent_CPA
    if sp.split_ahead
      send(unsolvable,sp.next_agent)
      rename_SP(sp)
  assign_CPA
```

2.3.11 Speculative distributed constraint logic programming

The field of DisCSP focuses mostly on numerical constraint satisfaction. However, centralized constraint solving has other paradigms and languages that solve different problem domains. *Constraint logic programming* extends *logic programming* with constraint satisfaction concepts. Constraints and domains are represented as part of *rules* composed of atoms that are either constraints or *queries*. The interpreter sequentially analyzes and checks each element in the *goal* of the program, a logical expression composed of multiple atoms. On finding a query, the interpreter consults with other rules that have the form of the query, substituting the unknown information with data from the other queries. If the query has multiple results, only one is returned at a time. When it finds a constraint, it is added ot a *constraint store* that keeps track of all constraints, and checks whether the latest queries produce valid variable assignments according to all constraints found so far. If the interpreter finds that the constraints are not satisfied, it *backtracks* in order to obtain the next result from previous queries. If the constraint store is satisfied, execution continues. The program terminates when there are no more atoms to check in the goal and all constraints are satisfied, which means a solution has been found, or when all queries have been exhausted without finding an assignment that satisfies the constraints in the store, returning a failure, or no solution [8].

In a distributed version of this process, the executing agents has only a partial set of queries from the entire problem. When an agent encounters a query that cannot be resolved by itself, the query is forwarded to an agent that can return an answer. Originally, the asking agent has to wait for an answer before progressing with its execution, otherwise it would not be able to fulfill its own queries and validate its constraints. In *speculative distributed constraint logic programming* [4], the program assigns *default values* to the unknown variables involved in external (*askable*) queries, and continues its execution normally. When the agent receives the answer to a query, it revises existing information.

The main objects used for this model are *process* and *answer entry*. Processes correspond to alternative computations, generating a new one whenever a new line of computation is encountered, by assigning default values, splitting cases or receiving an answer. Each process has a designated goal, and the process is finished successfully when there are no more atoms and constraints to process in the goal, or with a failure when the default constraints contradict the recently returned answers. Answer entries keep track of answers of previously-asked queries; each answer has an id used to distinguish between revisions to previous answers, or an entirely new answer, which in turn creates a new process. When a new process is created, a *default process* is kept suspended in order to reconstruct the original, while the newly created process is executed normally. This means that for every computation decision point, two new processes are created: the *default* suspended process, and the process that will be executed. This creates a *computation tree*; however, the main advantage of this algorithm is that not all processes are kept in memory, only the leaves of the computation tree [4].

Algorithm 11. Speculative distributed constraint logic programming

```
main:
  default_proc ← new_process()
  default_proc.body ← rules
  proc_list.add(default_proc)
  process_reduction(msg)
  while proc_list != :
    msg ← receive_msg()
    if msg.type = query_init:
      init_q ← msg.query
      default_proc ← new_process()
      default_proc.body ← init_q
      proc_list.add(default_proc)
      process_reduction(msg)
    else if msg.type = query:
      process_reduction(msg)
    else if msg.type = answer:
      fact_arrival(msg)
```

Algorithm 11. Speculative distributed constraint logic programming (cont'd)

```
fact_arrival(msg):
  ans ← answers.get(msg.query,msg.ans_id)
  if ans = null:
    ans ← answers.add_ans(msg.query,msg.ans_id,msg.constr,)
    for each def_ans in answers.get_default_answers(msg.query):
      for each proc in proc_list where proc.id is in def_ans.process_list:
        if proc.finished and proc.constraint != (proc.constraint and msg.constr):
          send(answer,(proc.query,proc.id,proc.constraint and msg.constr),msg.id)
        if proc.is_ordinary:
          proc.wait_list.add(msg.query,def_ans.id)
          proc.answer_list.remove(msg.query,def_ans.id)
          if consistent(msg.constr and proc.constr):
            newProc ← new_process()
            newProc.constraint ← msg.constr and proc.constr
            newProc.goal_st ← proc.goal_st
            newProc.wait_list ← proc.wait_list
            newProc.answer_list ← proc.answer_list
            newProc.answer_list.add(msg.query,ans.id)
            newProc.answer_list.remove(msg.query,def_ans.id)
            proc_list.add(newProc)
            ans.process_list.add(newProc.id)
    orig_ans ← answers.get(msg.query,true,o)
    for each proc in proc_list where proc.id in orig_ans.process_list and consistent(msg.constr and proc.constr an
      newProc ← new_process()
      newProc.constraint ← msg.constr and proc.constr and not ans.constr
      newProc.goal_st ← proc.goal_st
      newProc.wait_list ← proc.wait_list
      newProc.wait_list.remove(msg.query,o)
      newProc.answer_list ← proc.answer_list
      newProc.answer_list.add(msg.query,ans.id)
      proc_list.add(newProc)
      ans.process_list.add(newProc.id)
  else:
    ans.constr ← msg.constr
    ans.proc_list ← msg.proc_list
    for each proc in ans.proc_list:
      if proc.finished and proc.constraint != (proc.constraint and msg.constr):
        send(answer,(proc.query,proc.id,proc.constraint and msg.constr),msg.id)
      if proc.is_ordinary:
        if consistent(ans.constr and proc.constr):
          proc.constr ← ans.constr and proc.constr
        else:
          proc_list.remove(proc)
          ans.proc_list.remove(proc)
    orig_ans ← answers.get(msg.query,true,o)
    for each proc in proc_list where proc.id in orig_ans.process_list and consistent(msg.constr and proc.constr an
      newProc ← new_process()
```

Algorithm 11. Speculative distributed constraint logic programming (cont'd)

```
process_reduction(msg):
   proc ← proc_list.select(msg.query,wait_list = null)
   if proc != null:
      if proc.goal_st = null:
         send(answer,(init_q,proc.id,proc.constraint),msg.id)
         proc.query ← init_q
         proc.finished ← True
      else:
         atom ← select_atom(proc.goal_st)
         if not atom.is_askable:
            for every rule in rules:
               if atom = rule.head and consistent(proc.constraint and rule.constraints)
                  newProc ← new_process()
                  newProc.constraint ← proc.constraint and constraint(atom = rule.head) and rule.constraints
                  newProc.goal_st ← rule.body.union(proc.goal_st).remove(atom)
                  newProc.answer_list ← proc.answer_list
                  for every ans in proc.answer_list:
                     ans.process_list.add(newProc.id)
                  proc_list.add(newProc)
            for every ans in proc.answer_list:
               ans.process_list.remove(proc.id)
            proc_list.remove(proc)
         else if atom.is_askable:
            if answers.get_ordinary_answers(atom) =
               for each rule in rules where rule.is_default and consistent(proc.constraint and rule.constraint):
                  newProc ← new_process()
                  newProc.constraint ← proc.constraint and rule.constraint
                  newProc.goal_st ← proc.goal_st.remove(atom)
                  newProc.answer_list ← proc.answer_list
                  newProc.answer_list.add(atom,rule.id)
                  ans ← answers.get(atom,rule.id)
                  if ans != null:
                     ans.process_list.add(newProc.id)
                  else:
                     answers.add_ans(atom,ans.id,ans.constraint,newProc)
                  for every ans in proc.answer_list:
                     ans.process_list.add(newProc.id)
                  proc_list.add(newProc)
            else:
               for each o_ans in answers.get_ordinary_answers(atom) where consistent(proc.constraint and o_ans.con
                  newProc ← new_process()
                  newProc.constraint ← proc.constraint and o_ans.constraint
                  newProc.goal_st ← proc.goal_st.remove(atom)
                  newProc.answer_list ← proc.answer_list
                  newProc.answer_list.add(atom,o_ans.id)
                  for every ans in proc.answer_list:
                     ans.process_list.add(newProc.id)
```

Algorithm 11. Speculative distributed constraint logic programming (cont'd)

3 Distributed constraint optimization

3.1 Distributed constraint optimization problem

DisCSP problems are solved when all agents find a value in their domains that does not violate their constraints. However, not all problems have domains and constraints that can result in a set of values that does not violate the constraints. Not all problems are that well conditioned, and in some cases the best possible solution lies in *minimizing* the number of violated constraints. The need to solve DisCSP problems that seek to minimize the cost of constraint violations is the motivation behind the *distributed constraint optimization problem* (DCOP) was developed [12].

A DCOP has a set of variables and associated *cost functions* distributed among multiple agents, such that each agent holds one or more variables and their respective *cost functions*, which may involve unknown variables from other agents. Unlike DisCSP, the DCOP model assumes that the communication between agents occurs to satisfy the value needs for the cost functions in all agents. This means that a cost function can be private for an individual agent, communicating only the values of the variables necessary to compute the cost function. Most DCOP algorithms assume a communication model in which each agent holds a single variable and one or more cost functions that determine the communication channels that exist between agents. Extending this model to a more general one is trivial.

Definition 3. A *distributed constraint optimization problem* (DCOP) contains a set of n variables

$$X = x_1, \dots, x_n$$

with respective domain values

$$D = D_1 \times \dots \times D_n$$

distributed among m agents, such that each agent assigns the value of 1 or more variables, and each agent holds a set of o cost functions $f_k(x_i, \dots, x_j)$ where the agent knows some variables in $x_i, \dots, x_j$, with the objective of minimizing the sum of all cost functions from all agents.

3.2 Algorithms for DCOP

This section describes known algorithms to solve DisCSP. Unless otherwise specified, all these algorithms makes the following assumptions:

- All agents hold a single variable

- All agents have cost functions that may or may not involve other agents' variables
- The objective is to minimize the sum of the costs of all agents

3.2.1 Synchronous branch and bound

Just like synchronous backtracking in DisCSP, *synchronous branch and bound* (SynchBB) is a straightforward adaptation of a non-distributed algorithm into a distributed environment. *Branch and bound* is an optimization algorithm that *branches* the space of the domain, limiting the space of the search, and uses an *upper bound* of an *objective / cost function* to determine whether a domain / assignment is an improvement over previous assignments.

As a *synchronous* algorithm, SynchBB is strictly sequential, so the agents have a total order relation between them. Agents communicate by sending *paths* composed of domain assignments, with the first agent sending its initial value only, as well as a known upper bound for the objective function, which is often the sum of all constraint violations in the set of agents [7].

When an agent receives a path, it evaluates the path in addition with the next value of its own variable domain (the first value, if it is the first path received), obtaining the cost of selecting that value. If the cost evaluation is less than the current upper bound, that cost valuation becomes the new upper bound, and the agent sends an updated path with its selected value, along with the new upper bound, to the next agent in the order. If a value selection results in a value that does not improve the upper bound, the agent sequentially tries more values from its domain until it finds one that improves the cost. If the agent cannot find such a value, then the agent sends a *backtrack* message to the previous neighbor in the order [7].

Upon receiving a *backtrack* message, an agent selects the next value in its domain, following the same procedure as when receiving the path from the previous agent in the order: the agent finds for a value+path combination that improves the upper bound, sending the improved path to the next agent in the order, or sending a backtrack message if no such evaluation can be found [7].

Algorithm 12. Synchronous branch and bound

```
main:     if agent is first_agent
      agent_value ← select_first_value
      upper_bound ← select_upper_bound
      previous_path ← null
      counter ← 0
      send(value,path(agent_id,agent_value,counter),upper_bound,next_agent)
   while not done
      if msg.type = value
         previous_path ← msg.path
         upper_bound ← msg.ub
         next ← get_next(domain)
         send_value
      if msg.type = backvalue
         next_counter ← msg.path.get_next_value
         upper_bound ← msg.ub
         next ← get_next(domain.remove_previous(agent_value))
         send_value
```

```
send_token:     if next ≠ null
      if last_agent
         next_to_next ← next
         while next_to_next ≠ null
            if new_path.get_max_nv < upper_bound
               upper_bound ← new_path.get_max_nv
               best_path ← new_path
            if upper_bound = 0
               done ← true
               terminate
            next_to_next ← get_next(domain.remove_previous(next_to_next))
         send(backvalue,previous_path,upper_bound,previous_agent)
      else
         send(value,new_path,upper_bound,next_agent)
   else if agent is first_agent
      done ← true
      terminate
   else
      send(backvalue,previous_path,upper_bound,previous_agent)
```

Algorithm 12. Synchronous branch and bound (cont'd)

```
get_next(value_list):   if value_list = null
      return null
   else
      val ← value_list.pop
      new_path ← null
      counter ← 0
      if check(previous_path)
         return val
      else
         return get_next(value_list)
```

```
check(path):   if path = null
      new_path.add(agent_id,agent_value,counter)
      return true
   else
      (p_id,p_value,next_counter) ← path.pop
      if not consistent((agent_id,agent_value),(p_id,p_value))
         counter ← counter + 1
         if counter ≥ upper_bound or next_counter + 1 ≥ upper_bound
            return false
         else
            new_path.add(p_id,p_value,next_counter + 1)
            return check(path)
      else
         new_path.add(p_id,p_value,next_counter)
         return check(path)
```

3.2.2 ADOPT

Asynchronous distributed optimization (ADOPT) is the first distributed, complete and asynchronous algorithm for DCOP. The method assumes that the agents follow a depth-first search tree structure, with every agent having one parent agent and one or more children agents. All agents exchange three types of messages: *VALUE* messages containing variable assignments are sent down the tree; *COST* messages containing the cost information of each agent and its children are sent up the three; *THRESHOLD* messages are sent from a parent agent to change the *backtrack threshold* of its children. In addition, the algorithm uses an interval-based mechanism for termination, by keeping track of a lower and upper bounds on the cost, ending the search when the difference between them is zero [16].

All agents begin by concurrently choosing a value for their variable. Next, all non-leaf agents send *VALUE* messages to their children. An agent that receives a *VALUE* message stores it in its *context*, a partial solution that contains information about an agent's higher neighbors. After receiving a parent value, the agent calculates the cost of those value assignments in addition to the cost received from its children (through *COST* messages) in the form of an *upper bound* and a *lower bound*, creates two new bounds based on its own domain and the received values, and sends its own *COST* message containing the agent's context, and calculated lower and upper bounds. Leaf nodes always have lower and upper bounds equal to their values [16].

An agent's backtrack threshold is used to change its value. If the calculated lower bound on the cost of the agent's value assignment is greater than the threshold, then the agent attempts to change its value to one that produces a reduced lower bound. The threshold must be updated if no such value exist, which is to say, the agent determines that the interval cost of its subtrees does not contain the threshold. When an agent is forced to change the threshold due to the sum of the costs from its children, it also sends this sum as a *THRESHOLD* message; the child agent that receives this message uses it to rebalance and distribute amongst its children satisfying a series of rules. This means each parent agent changes its children's thresholds to avoid overestimating the cost of the subtrees, as well as reconstruct threshold-abandoned solutions [16].

On its own, ADOPT is a very elegant algorithm with serious communication deficiencies, requiring many messages to ensure completeness. Many improvements have been made to the algorithm, resulting in new methods. *ADOPT-ng* [28] changes the communication model to enhance the *cost* message by incorporating *nogood* information, incorporates the *add-link* message found in ABT, and eliminates the need for a total order in the agents. *BnB-ADOPT* [33] is a variant of ADOPT that incorporates branch-and-bound and depth-first search techniques into ADOPT to improve the performance of the algorithm, in particular by pruning search nodes that cannot possibly improve the cost value.

Algorithm 13. Asynchronous distributed optimization (ADOPT)

```
main:
  threshold ←
  curr_context ← null
  for all v in domain and agent in children
    lb[v,agent] ← 0
    t[v,agent] ← 0
    ub[v,agent] ← infinity
    context[v,agent] ←
  agent_value ← minimize_LB(domain)
  do_backtrack
  while not done
    if msg.type = threshold
    if msg.context.compatible(curr_context)
      threshold ← msg.t
      maintain_t_invariant
      do_backtrack
    if msg.type = terminate
      terminate ← true
      curr_context ← msg.context
      do_backtrack
    if msg.type = value
      if not terminate
        curr_context.add(msg.id,msg.value)
        for all v in domain and agent in children
          if not context[v,agent].compatible(curr_context)
            lb[v,agent] ← 0
            t[v,agent] ← 0
            ub[v,agent] ← infinity
            context[v,agent] ←
        maintain_t_invariant
        do_backtrack
```

Algorithm 13. Asynchronous distributed optimization (ADOPT) (cont'd)

```
    if msg.type = cost
      c_val ← msg.context[agent_id]
      msg.context.remove(agent_id,c_val)
      if not terminate
        for all (id,val) in msg.context where not neighbors.contains(id)
          curr_context.add(id,val)
        for all v in domain and agent in children
          if not context[v,agent].compatible(curr_context)
            lb[v,agent] ← 0
            t[v,agent] ← 0
            ub[v,agent] ← infinity
            context[v,agent] ←
      if context.compatible(curr_context)
        lb[c_val,msg.id] ← msg.lb
        ub[c_val,msg.id] ← msg.ub
        context[c_val,msg.id] ← msg.context
        maintain_child_t_invariant
        maintain_t_invariant
      do_backtrack
```

do_backtrack

```
  if threshold = UB
    agent_value ← minimize_UB(domain)
  else if LB[agent_value] > threshold
    agent_value ← minimize_LB(domain)
  for each agent in neighbors where agent.priority < agent_priority
    send(value, (agent_id,agent_value),agent)
  maintain_alloc_invariant
  if threshold == UB
    if terminate or isRoot
      curr_context.add(agent_id,agent_value)
      for each agent in children
        send(terminate,curr_context,agent)
      terminate
  send(cost,(agent_id,curr_context,LB,UB),parent_agent)
```

3.2.3 Optimal asynchronous partial overlay

The *optimal asynchronous partial overlay* (OptAPO) algorithm adds an interesting concept to the process of solving a DCOP: mediation. Each agent contains an *agent view* that stores the names, values, domains and constraints from the agent's neighbors, a *good list* with the names of all other agents that have direct or indirect constraints with the current agent, and a dynamic *priority* based on the size of the *good list*. A larger *good list* means that the agent has more knowledge about the problem, so it gets assigned a higher priority. Priority is used to determine the agent that will mediate with other agents [14].

All agents start by selecting a value, and sending an *init* message to their neighbors. This message contains the variable, priority, current value, domain and constraints of the sender. A recipient of an *init* message adds the information to its *agent view*, and adds the variable name to its *good list* if the received variable has direct or indirect constraints with any of the other variables in the *agent view*. After receiving all *init* messages, each agent proceeds to calculate the minimum of the local subproblem defined by the constraints in the *good list* by the sum of their constraint violation, using the values in the *agent view* [14].

The expected minimum of a local subproblem is always initialized to zero. If the calculated minimum is greater than the expected minimum, the agent starts a *mediation session*, which can be passive or active, depending on the agent's priority. If the agent has the highest priority among its neighbors with suboptimal relationships, its session is active, otherwise, it is passive. An active mediator can only participate in one active mediation at a time, and seeks to update both the expected and the calculated minimum of its subproblem; a passive mediator can participate in multiple mediation processes, and only seeks to understand and update its expected minimum [14].

An active mediator uses its knowledge of the domains in lower priority agents to determine the values that improve the calculated local minimum; then, the mediator sends *value?* messages to all lower priority agents so they revise their values. However, if the mediator is unable to find an assignment that improves the calculated local minimum, it sends an *evaluate?* message to all agents on its *good list*, containing the variables and constraints from its *agent view* [14].

An agent that receives an *evaluate?* message can reply with one of two different messages: an *evaluate!* message containing variables and constraints unknown to the mediator that sent the *evaluate?* message, if the agent is not part of an active mediation; or, if the agent is part of an active mediation, it will send a *wait!* message. The mediator that receives a *wait!* message excludes the sender from the mediation session [14].

After the mediator receives all *evaluate!* or *wait!* messages from its *good list*, it does a branch-and-bound search on the subproblem of the *good list* using the received information to determine the new expected minimum. After finding the value assignments for this new minimum, the mediator sends *value?* messages to all agents that need to revise their local values, and finishes the mediation session [14].

Algorithm 14. Optimal asynchronous partial overlay (OptAPO)

```
main:
  agent_val ← select_random_value
  minimum ← 0
  priority ← sizeof(neighbors)
  med_type ← active
  med ← none
  good_list.add(agent_val)
  for each agent in neighbors
    send(init,(agent_id,priority,agent_val,med_type,dom,constr,path),agent)
  init_list ← neighbors
  while not done
    if msg.type = init
      ag_view.add(msg.contents)
      if good_agent.is_neighbor(msg.id) where good_list.contains(good_agent)
        good_list.add(msg.id)
        for each agent in ag_view where not good_list.contains(agent)
          if agent.is_neighbor(msg.id)
            good_list.add(agent)
        priority ← sizeof(good_list)
      if not init_list.contains(msg.id)
        send(init,(agent_id,priority,agent_val,med_type,dom,constr),msg.id)
      else
        init_list.remove(msg.id)
      check_ag_view
    if msg.type = value?
      ag_view.update(msg.contents)
      check_ag_view
    if msg.type = wait!
      counter ← counter - 1
      if counter = 0
        choose_solution
    if msg.type = evaluate!
      preferences.record(msg.id,msg.labeled_dom)
      counter ← counter - 1
      if counter = 0
        choose_solution
```

Algorithm 14. Optimal asynchronous partial overlay (OptAPO) (cont'd)

```
check_ag_view:
  if not init_list.is_empty or med ≠ null
    return
  v_constr ← constr.get_violated_constr
  new_med ← null
  if cost > minimum and neighbors.has_consistent_below(priority)
    new_med ← active
  else if cost > minimum
    new_med ← passive
  if new_med == active and not neighbors.has_active_above(priority)
    (new_min,new_value) ← minimize(dom)
    if new_min ≠ minimum and changes_in_lo_priority
      agent_val = new_value
      med ← null
      new_constr ← neighbors.get_optimal_neighbors
      for all agent in ag_view
        send(value?,(agent_id,priority,agent_val,med,new_constr), agent)
    else
      do_med(new_med)
  else if new_med = passive
    do_med(new_med)
  else if med ≠ new_med or (med = null and constr ≠ new_constr)
    med ← new_med
    for all agent in ag_view
      send(value?,(agent_id,priority,agent_val,med,new_constr), agent)
  else if med = null
    for all id_k in ag_view where id_k not in constr and id_k not in good_list
      for agent in path.to(id_k) where agent not in ag_view
        send(init,(agent_id,priority,agent_val,med,dom,constr,path),agent)
        init_list(agent)
  constr ← new_constr
```

3.2.4 Dynamic programming optimization protocol

Unlike most algorithms, the *dynamic programming optimization protocol* (DPOP) is less a search strategy and more a dynamic programming technique. DPOP consists of three stages:

1. In the *Pseudo-tree generation phase*, agents assign priorities in such a way that the resulting network represents a pseudo-tree, in which nodes can

have multiple children, there is one root node, all other nodes have one parent, and there is a number of leaf nodes with no children [23].

2. In the *UTIL propagation phase*, all agents transmit their optimal utilities based on their own list of values and the utilities received from children agents. The *UTIL* message propagation starts from the leaf nodes, and includes the cost of all possible assignments for the agent against all the received utilities, creating a multidimensional *utility matrix* [23].

3. The *VALUE propagation phase* begins after the root agent receives all *UTIL* messages and generates its utility matrix. Based on this matrix, the root agent selects a value that minimizes the cost of the problem, and sends *value* messages to its children to inform them of its decision. All agents repeat these steps, until the leaf agents receive and process their parents' *VALUE* messages [23].

The most notable aspect of this algorithm is that the number of messages is linear, and this is much smaller in comparison with other algorithms. However, a simple observation can also show the main weakness of this algorithm, which is also found in many dynamic programming problems: memory growth is exponential, and the *size* of the messages is proportional to the exponentially-expanding utility matrix. Problems that contain a considerable number of agents and possible values will have memory and communication problems not because of the number of messages, but the size of them. Newer variants of the algorithm, such as DPOP-ASP [10], are designed with the objective of reducing the memory size and complexity of the messages, with comparable time performance.

Algorithm 15. Dynamic programming optimization protocol (DPOP)

```
main:
  if agent is first_agent
    create_pseudotree
  if sizeof(children) = 0
    utility_parent ← compute_utility(parent,pseudo_parents)
    send(util,utility_parent,parent)
  while not done
    if msg.type = util
      utilities[msg.id] ← msg.utility)
      if agent_view.contains_all(children)
        if parent = null
          optimum ← choose_optimal(null,utilities)
          for all child_agent in children, pseudo_children
            send(value,optimum,child_agent)
        else
          utility_parent ← compute_utility(parent,pseudo_parents)
          send(util,utility_parent,parent)
    if msg.type = value
      agent_view.add(msg.id,msg.value)
      if agent_view.contains_all(parent, pseudo_parents)
        (agent_value, optimum) ← choose_optimal(agent_view,utilities)
        for all child_agent in children, pseudo_children
          send(value,(agent_value,optimum),child_agent)
        done ← true
```

3.2.5 No-commitment branch and bound

The *No-commitment branch and bound* algorithm (NCBB) uses a pseudo-tree structure to guide the search. The first step, then, is to arrange the priorities of the agents such that they have the properties of a tree, with each agent besides the *root agent* having exactly one parent. Each agent maintains a *costs* map, a list of *unexplored* trees and a list of values assigned to its subtrees (the *anncdVals* list). After the initial priority assignment, parent agents compute upper and lower bounds on their local solution using greedy search and transmit it to their descendants using a *SEARCH* message [5].

An agent that receives a *SEARCH* message begins searching for a solution by sending its value information to its children. What makes this algorithm interesting is that, depending on the previous costs of its children subtrees and the calculated upper bound on the optimum, it can send a different value to each subtree, exploring different regions of the search space. The *unexplored* and *anncdVals* keep track of which values have not been sent to which trees, and which trees have been sent which values, respectively. Additionally, every time an agent receives a value update from its parent, the child agent computes all possible lower bounds on the cost, one for each value in the child agent's domain, stores them in the *costs* map, and sends to its parent the best agent cost according to the selected assignment and the values sent by its children. This mechanism improves the pruning capabilities of the algorithm, and allows the higher priority agents to calculate tighter bounds on the optimal solution [5].

Algorithm 16. No-commitment branch and bound (NCBB)

```
main:
  if parent ≠ null
    update_context
  while not done
    do_search
    can_stop ← update_context
    if parent = null or can_stop
      done ← true
  costs[result_value] ← 0
  for all child_agent in children
    subtree_search(result_value, child_agent)
  for all child_agent in children
    send(stop,child_agent)

update_context:
  while not done
    receive(msg)
    if msg.type = search
      bound ← msg.bound
      return false
    if msg.type = value_msg
      agent_cont.add(msg.id,msg.value)
      lb_anc ← ancestors_min(agent_id,agent_cont,msg.id_index)
      lb_anc_2 ← ancestors_min(agent_id,agent_cont,msg.id_index - 1)
      new_lb ← lb_anc - lb_anc_2
      send(cost,new_lb,msg.id)
    if msg.type = stop
      return true
```

Algorithm 16. No-commitment branch and bound (NCBB) (cont'd)

```
subtree_search(val,child):
   for all descendant in descendants[child]
      send(value_msg,val,descendant)
   anncdVals[child] ← val
   for all descendant in descendants[child]
      receive(cost_msg)
      costs[val] ← costs[val] + cost_msg.cost
   if costs[val] > bound
      prune
      anncdVals[child] ←
      return false
   else
      new_bound ← bound - costs[val]
      send(search,new_bound,child)
      return true
```

```
Algorithm 16. No-commitment branch and bound (NCBB) (cont'd)
do_search:
  idle ← children
  cost ←
  unexpl ←
  anncdVals ←
  min_cost ← ancestors_min(agent_id,agent_cont,sizeof(ancestors))
  for all val in domain where agent_cost(val,agent_cont) ≤ bound + min_cost
    costs[val] ← agent_cost(val,agent_cont) - min_cost
  for all val in domain where costs[val] ≠ null
    unexpl[val] ← children
  while not unexpl.empty or not anncdVals.empty
    while not idle.empty
      child ← idle.pop
      val ← select_unexpl_value_for_child(child,unexpl)
      unexpl[val].remove(child)
      val_c ← select_unexpl_value_for_child(child,unexpl)
      if not subtreeSearch(val,child) and val_c ≠ null
        idle.add(child)
    if not anncdVals.empty
      receive(cost_msg)
      val ← anncdVals[cost_msg.id]
      anncdVals[cost_msg.id] ← null
      costs[val] ← costs[val] + cost_msg.cost
      if costs[val] > bound
        prune
      else if unexpl[val].empty and not anncdVals.contains_var(cost_msg.id)
        bound = costs[val]
        result_value = val
        prune
    val_c ← select_unexpl_value_for_child(child,unexpl)
    if val_c ≠ null
      idle.add(child)
  if parent ≠ null
    minimum ← minimize(costs)
    send(cost,minimum,parent)
```

3.2.6 Asynchronous forward bounding

Like all other branch-and-bound algorithms for DCOP, *asynchronous forward bounding* (AFB) uses the lower and upper bounds on the solution to guide the search. Its main communication mechanism is *current partial assignments* (CPAs), which are transmitted from higher to lower priority agents along with the CPA's cost, which is the sum of the constraint costs (violations). Additionally, each agent keeps track of a lower bound for its CPA, along with a global upper bound [6].

In general, AFB is very similar to the asynchronous forward-checking method to solve DisCSP, with one key difference: the value change / backtracking conditions is are based on the bounds on the global solution. An agent that sends a CPA, also sends a copies of it to future agents requesting their lower bounds on the cost, and using them to compute its own cost. If this lower bound is greater than the global upper bound, the agent tries to reassign its value so its cost is less than the global upper bound; if this is not possible, the agent then sends a backtrack message to its preceding agent [6].

Algorithm 17. Asynchronous forward bounding

```
main:
  bound ← inf
  if agent is first_agent
    agent_cpa ← generate_cpa
    assign_cpa
  while not done
    if msg.type = fb_cpa
      lb_estimate ← estimate_lb(msg.pa)
      send(fb_est,(lb_estimate,msg.pa),msg.id)
    is msg.type = fb_est
      estimates.add(msg.lb_estimate)
      if cpa.cost + sum(estimates) ≥ bound
        assign_cpa
    if msg.type = cpa_msg
      cpa ← msg.pa
      temp_cpa ← msg.pa
      if temp_cpa.contains(agent_id)
        temp_cpa.remove(agent_id)
      if temp_cpa.cost ≥ bound
        do_backtrack
      else
        assign_cpa
```

Algorithm 17. Asynchronous forward bounding (cont'd)

```
assign_cpa:
  estimates ←
  if cpa.contains(agent_id)
    cpa.remove(agent_id)
  new_value ← null
  for each value in domain
    if cpa.cost + agent_cost_function(value) < bound
      new_value ← value
  if value = null
    do_backtrack()
  else
    agent_value ← new_value
    cpa.add(agent_id,agent_value)
    if cpa.is_complete
      broadcast(new_solution,cpa)
      bound ← cpa.cost
      assign_cpa
    else
      send(cpa_msg,cpa,next_agent)
      for each agent in unassigned
        send(fb_cpa,(agent_id, cpa),agent)
do_backtrack:
  estimates ←
  if agent is first_agent
    broadcast(terminate)
  else
    send(cpa_msg,cpa,previous_agent)
```

3.2.7 Concurrent forward bounding

Concurrent forward bounding (ConcFB) is an algorithm that, unlike many other advanced DCOP methods, does not incorporate an asynchronous communication technique [20]. It is related to two DisCSP algorithms: *asynchronous forward backtracking* [15] and *concurrent dynamic backtracking* [42].

ConcFB uses a *synchronous forward bounding* (SFB) as its main search method. SFB is a synchronous version of asynchronous forward backtracking, with the difference being that in the asynchronous version, an agent adds its value to the *consistent partial assignment* (CPA) and sends it to the next agent in the order, along with copies for all the unassigned agents, without waiting for the feedback of the unassigned agents. In the synchronous version, the agent adds its value to the CPA, sends a copy of it to all unassigned agents (excluding the next agent in the order) and *waits* for their feedback. The agent sends the CPA only after receiving *all* feedback messages from the unassigned agents and revising its knowledge if needed [20].

Just like concurrent dynamic backtracking, in ConcFB an agent can choose to split the domain of its variable into multiple subproblems assigned into *search processes*, each one following its own SFB process, thus each subproblem is processed and solved asynchronously and concurrently. All unassigned agents that receive a copy of a CPA calculate a *lower bound* to the cost of their possible solutions, which is sent to the originator of the CPA copy. This agent compares this lower bound to the global *upper bound*, backtracking the search towards the previous agent in the order of the sum of the costs of the received lower bounds is greater than the upper bound. If an agent finds a new upper bound, this value is broadcast to all agents, who compare it to their last known upper bound and update it accordingly [20].

Algorithm 18. Concurrent forward bounding (ConcFB)

```
main:
  if agent is first_agent
    root_sp ← create_sp(root_id,domain)
    root_sp.splits ← create_split_set
    init_sp
  while not done
    if msg.type = cpa_msg
      sp_id ← create_sp(msg.sp_id,domain)
      sp_id.cpa ← msg.cpa
      sp_id.lb_list ← msg.lb_list
      sp_id.lb_list.remove(agent_lb)
      sp_list.add(sp_id)
      assign_cpa(sp_id)
    if msg.type = backtrack_cpa
      sp_id ← sp_list.get(msg.sp_id)
      current ← sp_id.current
      sp_id.domain.remove(current)
      if sp_id.domain.is_empty
        do_backtrack(sp_id)
      else
        assign_cpa(sp_id)
```

Algorithm 18. Concurrent forward bounding (ConcFB) (cont'd)

```
    if msg.type = lb_request
      agent_lb ← minimize_cost(msg.cpa,domain)
      send(lb_report,agent_lb,msg.agent_id)
    if msg.type = lb_report
      sp_id ← sp_list.get(msg.sp_id)
      sp_id.lb_list[msg.agent_id] ← msg.lb
      received_reports.add(msg.agent_id)
      if received_reports = unassigned
        if sp_id.cpa.cost + sp_id.current.cost + sum(sp_id.lb_list) < agent_ub
          cpa ← sp_id.cpa
          cpa.add(sp_id.current)
          send(cpa_msg,(cpa,sp_id.lb_list),next_agent)
        else
          sp_id ← sp_list.get(msg.sp_id)
          current ← sp_id.current
          sp_id.domain.remove(current)
          if sp_id.domain.is_empty
            do_backtrack(sp_id)
          else
            assign_cpa(sp_id)
    if msg.type = ub_update
      if msg.ub < agent_ub
        agent_ub ← msg.ub
      else
    if msg.type = terminate
      done ← true
  return agent_ub
```

init_sp:

```
  for i ← 1 to sizeof(domain)
    sp_id ← create_sp(i,domain[i])
    root_sp.splits.add(sp_id)
    assign_CPA(sp_id)
```

Algorithm 18. Concurrent forward bounding (ConcFB) (cont'd)

```
assign_cpa(sp_id):
  cpa ← sp_id.cpa
  new_val ← select_new_value(domain)
  current ← (agent_id,new_val)
  if new_val = null
    send(backtrack_cpa,cpa,previous_agent)
  else
    cpa.add(agent_id,new_val)
    cpa.cost ← cpa.cost + current.cost
    for each agent in unassigned
      send(lb_request,cpa,agent)
```

3.2.8 Divide and coordinate subgradient algorithm

Most literature on DCOP algorithms focuses on global / complete optimization search. The main drawback of these algorithms is the time it takes for them to find a solution. However, some recent algorithms choose to trade accuracy and obtain a good local optimum in exchange for speed. The *divide and coordinate* technique is one of these algorithms, based on a two-stage process: first, the agents *divide* the problem into local sub-problems that are solved individually by each agent, with agents potentially sharing variables; then, the agents *coordinate* by sending information about their assignments, identifying disagreements and making corrections and new problem subdivisions that improve the level of agreement between the agents. The algorithm alternates between *divide* and *coordinate* stages until all agents agree on their local solutions, or another termination condition is met [31].

It is important to note that during the *divide* stage, each agent can modify its local subproblem, as long as all subproblems compose into the original problem. The *divide and coordinate subgradient algorithm* employs Lagrangian decomposition and subgradient methods during its *divide* stage to obtain dual subproblems. During *coordination*, the agents attempt to reduce conflict by modifying the subgradient parameters. This version of the algorithm alternates between the two stages until the difference between the found solution and a predefined bound is close to zero, or a user-defined number of divide-and-coordinate iterations have passed without finding a solution [31].

Algorithm 19. Divide and coordinate subgradient algorithm (DaCSA)

```
main:
  bound ← inf
  lambda ← 0
  solution ← null
  best_value ← -inf
  cands ← null
  subproblem ← create_subproblem(vars,domain,utility_rels)
  while not termination_condition
    ub_subproblem ← modify_subproblem(subproblem, lambda)
    (curr_sol, curr_min) ← solve_subproblem(ub_subproblem)
    for each neigh in neighs
      send(value,(curr_sol[agent_id],curr_sol[neigh],curr_min,cands),neigh)
    received ←
    while received.id_list ≠ neighs
      received[msg.id] ← msg.contents
    step_size ← update_step_size
    lambda ← update_coord_params(lambda,step_size,curr_sol)
    if received.has_better_bound(bound)
      bound ← received.best_bound
    if received.has_better_sol(best_value)
      best_value ← received.best_value
      solution ← received.best_solution
    cands ← select_candidate_solutions(curr_sol[agent_id],cands)
  return (solution,best_value,bound)
```

3.2.9 Distributed upper confidence tree

The *distributed upper confidence tree* (DUCT) algorithm is an incomplete search method that can quickly find near-optimal solutions to DPOP problems. It incorporates elements that are similar to complete algorithms, such as requiring a pseudo-tree structure, but the idea behind its search pattern is very different. Instead of trying to systematically calculate the best possible local cost, agents maintain *confidence bounds* delimiting promising subspaces of the domain, selecting a *random sample* from this higher‘-confidence subspace [21].

At the beginning of this algorithm, the *root agent* selects a value for its variable and sends a *CONTEXT* message to its children. Each child agent randomly selects a value from its domain and sends a *CONTEXT* message to its children, repeating this process until the leaf nodes are reached. This creates a *search path* in which all variables are assigned. Children nodes calculate their cost based on their parent's context and their own assignment, and send back the information through *COST* messages, with the recipients adding their own costs to all their childrens' and sending *COST* messages further up the tree, until reaching the root node that repeats the process [21].

What differentiates this algorithm is that each agent also keeps track of the number of times each *value* has been selected, and the number of times a certain *context* has been received. These two values are used to calculate a confidence bound that reduces the search space into a promising subspace. It is a lower bound that is adjusted over time to both limit the search space to while not entirely discarding other subspaces [21].

Algorithm 20. Distributed upper confidence tree (DUCT)

```
main:
  if agent is root
    parent_finished ← true
    agent_value ← sample()
    agent_context.add(agent_id,agent_value)
    for each child in children
      send(context,agent_context,child)
  else
    parent_finished ← false
  while not done
    if msg.type = context
      if children.is_empty
        leaf_min ← minimize_constraint_sum(domain,msg.context)
        send(cost,(leaf_min,leaf_min),parent)
      else
        agent_value ← sample(msg.context)
        agent_context ← msg.context
        agent_context.add(agent_id,agent_value)
        for each child in children
          send(context,agent_context,child)
```

Algorithm 20. Distributed upper confidence tree (DUCT) (cont'd)

```
    if msg.type = f-context
      parent_finished ← true
      if termination_condition
        agent_context.add(agent_id,msg.context[agent_id])
        for each child in children
          send(f-context,agent_context,child)
      else
        agent_value ← sample(msg.context)
        agent_context ← msg.context
        agent_context.add(agent_id,agent_value)
        for each child in children
          send(context,agent_context,child)
    if msg.type = cost
      received.add(msg.id,msg.cost,msg.bound)
      if received.id_list = children
        agent_cost ← calculate_cost(agent_context,received.costs)
        agent_bound ← calculate_bound(agent_context,received.bounds)
        if parent_finished and termination_condition
          agent_context.add(agent_id,agent_value)
          for each child in children
            send(f-context,agent_context,child)
        else if parent_finished or agent_cost = inf
          agent_value ← sample(msg.context)
          agent_context ← msg.context
          agent_context.add(agent_id,agent_value)
          for each child in children
            send(context,agent_context,child)
        else
          send(cost,(agent_cost,agent_bound),parent)
```

3.2.10 D-Gibbs

The main drawback of DUCT is its memory requirement, as storing all contexts requires an exponential amount of memory. Based on the message model of DUCT, the *Distributed Gibbs* algorithm also constructs a pseudo-tree and uses random value selection; however, it takes inspiration from the Gibbs sampling method (a Markov chain algorithm used to approximate joint probability distributions) to determine the value of all agents without storing all the historical contexts along with their frequencies.

In D-Gibbs, all agents contain three values: the current value, the previous value, and the value corresponding to the best cost found so far. All agents also maintain a *context* with all the values of its neighbors, and a *time index* to indicate the number of iterations the agent has sampled. It also maintains two *delta* values: the difference between the current solution and the best solution from the previous iteration, and the difference the best solution of the current iteration and the best solution of the previous iteration.

At the beginning of the algorithm, all agents select their default values. The root samples its initial value based on a probability distribution, and sends a *VALUE* message to its neighbors. An agent that receives a *VALUE* message stores that value in their respective context, and, if the sender was the agent's parent, samples its value and propagates it to its neighbors with its own respective *VALUE* message, propagating these messages until the leaves of the tree sample their values. Leaf agents send a *BACKTRACK* message to their parents, and in turn they propagate their own *BACKTRACK* message until the root receives all responses from its neighbors, at which point the algorithm completes one iteration.

The delta values are transmitted as part of both the *VALUE* and *BACKTRACK* messages. An agent that receives a *VALUE* message and samples its value will calculate the difference between the current solution and the best solution from the previous iteration by adding to it its own difference in local quality. If the calculated difference is larger than the difference to the best solution between iterations, this value is replaced, along with the agent's own best value found. This update is transmitted to the rest of the agents through *VALUE* and *BACKTRACK* messages. With this, every time an improved solution is found, all agents receive the updated value by the next iteration. The algorithm terminates either after a given number of iterations, or when no improvements to the solution are found after a number of consecutive iterations.

Algorithm 21. Distributed Gibbs (D-Gibbs)

```
main:
  current_value ← init_value
  prev_value ← init_value
  best_value ← init_value
  agent_context.add(agent_id,current_value)
  for each n_agent in neighbors
    n_value ← create_assumption(n_agent)
    agent_context.add(n_agent.id,n_value)
  prev_diff ← 0
  best_diff ← 0
  iter ← 0
  best_iter ← 0
  if agent is root
    iter ← iter + 1
    do_sampling
  while not done
    if msg.type = value
      agent_context.update(msg.id,msg.value)
      if msg.id = parent
        wait_for_pseudoparents
        iter ← iter + 1
        if msg.best_iter = iter
          best_value ← current_value
        else if msg.best_iter = iter - 1 and msg.best_iter > best_iter
          best_value ← prev_value
        prev_diff ← msg.prev_diff
        best_diff ← msg.best_diff
        best_iter ← msg.best_iter
        do_sampling
        if agent is leaf
          send(backtrack,(prev_diff,best_diff),parent)
```

Algorithm 21. Distributed Gibbs (D-Gibbs) (cont'd)

```
    if msg.type = backtrack
      prev_diff_list[iter,msg.id] ← msg.prev_diff
      best_diff_list[iter,msg.id] ← msg.best_diff
      if prev_diff_list[iter].agents = children
        prev_diff ← sum(prev_diff_list[iter].values) - ...
          ... (sizeof(children) - 1) × prev_diff
        best_diff_new ← sum(best_diff_list[iter].values) - ...
          ... (sizeof(children) - 1) × best_diff
        if best_diff_new > best_diff
          best_diff ← best_diff_new
          best_value ← current_value
          best_iter ← iter
        if agent is root
          prev_diff ← prev_diff - best_diff
          best_diff ← 0
          iter ← iter + 1
          do_sampling
        else
          send(backtrack,(prev_diff,best_diff),parent)
```

do_sampling:

```
  prev_value ← current_value
  current_value ← get_random_sample
  prev_diff ← prev_diff + sum_gain(current_value,domain) - ...
    ... sum_gain(prev_value,domain)
  if prev_diff > best_diff
    best_diff ← prev_diff
    best_val ← current_val
    best_iter ← iter
  for each n_agent in neighbors
    send(value,(current_value,current_diff,best_diff,best_iter),n_agent)
```

4 Applications

The most common problem used to test DisCSP and DCOP algorithms are classic CSP problems such as *n-queens* or *graph coloring*. These CSP problems are extended into a distributed version in which the number of agents equals the number of variables, and while they provide a good initial benchmark, they can still be considered *toy problems* with little actual application in solving real-world problems.

This section section shows some real-world applications and benchmarks for DisCSP and DCOP found in the literature.

4.1 Sensor networks

This area includes multiple applications of both DisCSP and DCOP. A *sensor network* features an array of interconnected sensor units that has to coordinate to achieve a specific objective. The type of sensor network problem determines the variables in play, and whether the problem is a DisCSP or DCOP problem. For example, a group of static sensors used to keep track of airborne objects works only on a fixed area, under limited amount of power. Additionally, these sensors must coordinate to create a schedule, so their radio transmissions to each other receive no interference from other sensors [41].

There is at least one test bed for sensor networks, SensorDCSP [2]. This platform simulates a mobile-sensor problem: there are multiple sensors and mobile targets. The sensors are pair-wise disjoint and have two sets of constraints: visibility (can the sensor detect a mobile?), and compatibility (how close is the sensor to other sensors who can detect the mobile?). Thus, the objective of the SensorDCSP problem is to find *cliques* of 3 sensors that are detecting a mobile target. The most simple version of this problem, GSensorDCSP, uses a *sensor grid*, in which the agents observe only the area around their four quadrants [2].

4.2 Scheduling

Scheduling is a well-known problem in the CSP literature, and has been used extensively to create and test new algorithms. However, there exist what is known as *distributed scheduling* or distributed time-tables, in which a schedule or time-table is generated by the cooperation and negotiation of multiple agents. A perfect example of this problem is university course schedules: each department has different resources, requirements and restrictions, and ultimately they must communicate to produce a timetable by negotiating using public information and keeping their own private information. All of this without considering complications that arise from sharing resources between departments, such as shared courses and shared faculty [9].

Another prevalent case in the literature is *meeting scheduling*. In this problem, a group of agents needs to agree on a time for a meeting. Each agent maintains its own private schedule, but must make certain availability information known to other agents. All agents, then, must negotiate by trading potential meeting times until they can all reach an agreement [38].

4.3 Wireless network planning

Another area of application is in *wireless network planning*. One particular area focuses on *interference* found in wireless area networks, caused by transmission channel assignments. This is not a problem for networks in which all the wireless access points belong to the same network administrator; however, when there are multiple access points belonging to multiple administrators, the channel selection of the access points can cause interference and wireless service degradation. Each access point, then, serves as an agent, with every two pair of agents within the same range sharing a constraint on their wireless channels [18].

4.4 Vehicle routing / service delivery optimization

Vehicle routing is a classic applied CSP/COP. The particular version of this problem that can be solved using DCOP is called the *multiple-depot vehicle routing problem*. It is based on a delivery company that has subcontracted delivery operations to multiple subcontractors, and must coordinate the assignment of deliveries to ensure they will be done on time. However, each subcontractor also has to try to make more deliveries, maximizing their profits. Each contractor is physically separated from the others, has its own local optimal objective, and they all have inter-agent scheduling constraints [11].

5 Observations

5.1 Other partial / incomplete algorithms

Most algorithms in DCOP focus on *complete search*. That is, they try to find the *global minimum* of the cost. This process can be time consuming, and in many cases the problem could be considered solved with a *near-minimum*. The algorithms that focus on finding a *local minimum* are called *incomplete search* DCOP algorithms. The most effective of these have been outlined above; however, these are not the only ones. In general, incomplete search DCOP algorithms developed in the last ten years focus on *stochastic search methods* [41] and/or alternative measures of optimality such as *k-optimality* [22]. Other experimental algorithms are either focused on solving specific problems [13], or show interesting techniques with little in the way of concrete results [27].

5.2 Challenges

While the field itself emerged in the early 1990's, it saw very little growth until the 2000's, and even to this day it faces considerable challenges in its development.

5.2.1 Lack of innovation

Truly innovative methods are scarce in the field. Most are either distributed adaptations of existing CSP methods, or enhancements of other distributed algorithms. Different groups work on their respective algorithms, often reaching similar communication and coordination methods. Rarely do DisCSp and DCOP algorithms work outside of pseudo-tree networks.

5.2.2 Lack of benchmarking and comparisons

There are few attempts at creating benchmarks for DisCSP/DCOP. SensorDCSP [2] is likely the most established one, used to test multiple algorithms that have been developed since. Additionally, nearly all algorithms are compared against the most basic method in its class: DisCSP algorithms are always compared with ASB, while DCOP algorithms are compared with either SynchBB or the original ADOPT. There is no real sense of what could be considered the most advanced algorithm in the field, as all algorithms are presented as having improvements over old algorithms: ASB was published in 1992, SynchBB in 1997 and ADOPT in 2005.

5.2.3 Insufficient real world applications

The four applications mentioned are not the only ones, but are the most prevalent throughout the literature. The main problem is that few of them have actually found a real world application for general application methods, and the only tests they have carried out are through the use of *simulations*. Specific methods designed to solve individual, distributed problems are interesting but never achieve enough recognition or traction in the field [26].

6 Conclusions

The field of distributed constraint satisfaction contains multiple interesting ideas, rife with potential applications. However, one look at its history, development and techniques show some degree of stagnation: innovation is rare, and active applications even rarer. There is no consensus on what constitutes the best algorithms in the field, or even a robust set of benchmarks to measure the effectiveness of the algorithms, as found in other related fields, such as centralized CSP or optimization. However, as more decentralized networked technologies are developed, the field might yet see an unexpected resurgence, or change in priorities, with new research being motivated by newer potential applications.

References

[1] Krzysztof Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[2] Ramón Béjar et al. "Sensor networks and distributed CSP: communication, computation and complexity". In: *Artificial Intelligence* 161.1 (2005), pp. 117–147.

[3] L. Carlozo. *Why college students stop short of a degree.* `http://johngress.com/2012/03/27/why-college-students-stop-short-of-a-degree`.

[4] Martine Ceberio, Hiroshi Hosobe, and Ken Satoh. "Speculative constraint processing with iterative revision for disjunctive answers". In: *Computational Logic in Multi-Agent Systems*. Springer, 2006, pp. 340–357.

[5] Anton Chechetka and Katia Sycara. "No-commitment branch and bound search for distributed constraint optimization". In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM. 2006, pp. 1427–1429.

[6] Amir Gershman, Amnon Meisels, and Roie Zivan. "Asynchronous forward bounding for distributed COPs". In: *Journal of Artificial Intelligence Research* 34.1 (2009), p. 61.

[7] Katsutoshi Hirayama and Makoto Yokoo. "Distributed partial constraint satisfaction problem". In: *Principles and Practice of Constraint Programming-CP97*. Springer, 1997, pp. 222–236.

[8] Joxan Jaffar and Michael J Maher. "Constraint logic programming: A survey". In: *The journal of logic programming* 19 (1994), pp. 503–581.

[9] Eliezer Kaplansky and Amnon Meisels. "Negotiation among scheduling agents for distributed timetabling". In: *Proceedings of the 5th International Conference on Practice and Theory of Automated Timetabling, Pittsburgh*. Citeseer. 2004, pp. 517–520.

[10] Tiep Le et al. "ASP-DPOP: solving distributed constraint optimization problems with logic programming". In: *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2014, pp. 1337–1338.

[11] Thomas Léauté, Brammert Ottens, and Boi Faltings. "Ensuring privacy through distributed computation in multiple-depot vehicle routing problems". In: *Proceedings of the ECAI" 10 Workshop on Artificial Intelligence and Logistics (AILog" 10)*. EPFL-CONF-149797. 2010.

[12] Allan R Leite, Fabrício Enembreck, and Jean-Paul A Barthès. "Distributed constraint optimization problems: Review and perspectives". In: *Expert Systems with Applications* 41.11 (2014), pp. 5139–5157.

[13] Fabiana Lorenzi et al. "Improving recommendations through an assumption-based multiagent approach: An application in the tourism domain". In: *Expert Systems with Applications* 38.12 (2011), pp. 14703–14714.

[14] Roger Mailler and Victor Lesser. "Solving distributed constraint optimization problems using cooperative mediation". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society. 2004, pp. 438–445.

[15] Amnon Meisels and Roie Zivan. "Asynchronous forward-checking for DisCSPs". In: *Constraints* 12.1 (2007), pp. 131–150.

[16] Pragnesh Jay Modi et al. "ADOPT: Asynchronous distributed constraint optimization with quality guarantees". In: *Artificial Intelligence* 161.1 (2005), pp. 149–180.

[17] Pierre Monier, Sylvain Piechowiak, and Rene Mandiau. "A complete algorithm for DisCSP: Distributed Backtracking with Sessions (DBS)". In: *Second International Workshop on: Optimisation in Multi-Agent Systems (OptMas), Eigth Joint Conference on Autonomous and Multi-Agent Systems (AAMAS 2009), Budapest, Hungary*. 2009.

[18] Tânia L Monteiro et al. "A multi-agent approach to optimal channel assignment in wlans". In: *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*. IEEE. 2012, pp. 2637–2642.

[19] Paul Morris. "The breakout method for escaping from local minima". In: *AAAI*. Vol. 93. 1993, pp. 40–45.

[20] Arnon Netzer, Alon Grubshtein, and Amnon Meisels. "Concurrent forward bounding for distributed constraint optimization problems". In: *Artificial Intelligence* 193 (2012), pp. 186–216.

[21] Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. "DUCT: An upper confidence bound approach to distributed constraint optimization problems". In: *Proceedings of the National Conference on Artificial Intelligence*. Vol. 1. EPFL-CONF-197504. 2012, pp. 528–534.

[22] Jonathan P Pearce, Milind Tambe, and Rajiv Maheswaran. "Solving multiagent networks using distributed constraint optimization". In: *AI Magazine* 29.3 (2008), p. 47.

[23] Adrian Petcu and Boi Faltings. *A scalable method for multiagent constraint optimization*. Tech. rep. 2005.

[24] Patrick Prosser. "Hybrid algorithms for the constraint satisfaction problem". In: *Computational intelligence* 9.3 (1993), pp. 268–299.

[25] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[26] Ken Satoh, Philippe Codognet, and Hiroshi Hosobe. "Speculative Constraint Processing in Multi-agent Systems". English. In: *Intelligent Agents and Multi-Agent Systems*. Ed. by Jaeho Lee and Mike Barley. Vol. 2891. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 133–144. ISBN: 978-3-540-20460-2. DOI: 10.1007/978-3-540-39896-7_12. URL: http://dx.doi.org/10.1007/978-3-540-39896-7_12.

[27] Samaneh Hoseini Semnani and Kamran Zamanifar. "The power of ants in solving Distributed Constraint Satisfaction Problems". In: *Applied Soft Computing* 12.2 (2012), pp. 640–651.

[28] Marius C Silaghi and Makoto Yokoo. "ADOPT-ing: unifying asynchronous distributed optimization with asynchronous backtracking". In: *Autonomous Agents and Multi-Agent Systems* 19.2 (2009), pp. 89–123.

[29] Marius-Călin Silaghi and Boi Faltings. "Asynchronous aggregation and consistency in distributed constraint satisfaction". In: *Artificial Intelligence* 161.1 (2005), pp. 25–53.

[30] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. "Asynchronous search with aggregations". In: *AAAI/IAAI*. 2000, pp. 917–922.

[31] Meritxell Vinyals et al. "Divide-and-coordinate: DCOPs by agreement". In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems. 2010, pp. 149–156.

[32] X. Wang and M. Ceberio. "Fuzzy Measure Extraction for Predicting At-Risk Students". In: *Proceedings of 2nd World Conference on Soft Computing*. Baku, Azerbaijan, 2012.

[33] William Yeoh, Ariel Felner, and Sven Koenig. "BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm". In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems. 2008, pp. 591–598.

[34] William Yeoh and Makoto Yokoo. "Distributed problem solving". In: *AI Magazine* 33.3 (2012), p. 53.

[35] Makoto Yokoo. "Asynchronous weak-commitment search for solving distributed constraint satisfaction problems". In: *Principles and Practice of Constraint ProgrammingCP'95*. Springer. 1995, pp. 88–102.

[36] Makoto Yokoo. "Weak-commitment search for solving constraint satisfaction problems". In: *AAAI*. Vol. 94. 1994, pp. 313–318.

[37] Makoto Yokoo and Katsutoshi Hirayama. "Distributed breakout algorithm for solving distributed constraint satisfaction problems". In: *Proceedings of the Second International Conference on Multi-Agent Systems*. 1996, pp. 401–408.

[38] Makoto Yokoo, Koutarou Suzuki, and Katsutoshi Hirayama. "Secure distributed constraint satisfaction: Reaching agreement without revealing private information". In: *Principles and Practice of Constraint Programming-CP 2002*. Springer. 2002, pp. 387–401.

[39] Makoto Yokoo et al. "Distributed constraint satisfaction for formalizing distributed problem solving". In: *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*. IEEE. 1992, pp. 614–621.

[40] Makoto Yokoo et al. "The distributed constraint satisfaction problem: Formalization and algorithms". In: *Knowledge and Data Engineering, IEEE Transactions on* 10.5 (1998), pp. 673–685.

[41] Weixiong Zhang et al. "Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks". In: *Artificial Intelligence* 161.1 (2005), pp. 55–87.

[42] Roie Zivan and Amnon Meisels. "Concurrent search for distributed CSPs". In: *Artificial Intelligence* 170.4 (2006), pp. 440–461.

[43] Roie Zivan and Amnon Meisels. “Synchronous vs asynchronous search on DisCSPs”. In: *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*. 2003.