Discrepancy-based Method for Hierarchical Distributed Optimization

Jonathan Gaudreault\textsuperscript{1,2}, Jean-Marc Frayret\textsuperscript{1,2}, Gilles Pesant\textsuperscript{1}
\textsuperscript{1}École Polytechnique de Montréal
\textsuperscript{2}FORAC Research Consortium, Université Laval
\{jonathan.gaudreault, gilles.pesant, jean-marc.frayret\}@polymtl.ca

Abstract

Distributed Constraint Optimization is increasingly used for problem solving by multiple agents. However, there are situations where the system is made up of heterogeneous agents, for which the context, the structure, and the business rules define the interactions that are possible between them. As an example, supply chains are made up of interdependent business units having some form of customer-supplier hierarchical relationships. The coordination space for these hierarchical situations can be described as a tree. Therefore, we propose a distributed algorithm (MacDS) that performs discrepancy-based search which is known to perform well for centralized problems. The proposed algorithm is complete and aims at producing good solutions in a short amount of time. It allows concurrent computation and is tolerant to message delays. It has been evaluated using real industrial supply chain problems, for which it showed good performance.

1. Introduction

Many generic algorithms have been proposed in recent years to solve Distributed Constraint Optimization Problems (DCOP). In [1], the authors underline that those generic algorithms do not always take advantage of the characteristics of a particular class of problems, e.g. specific constraint structures or domain specific heuristics. Moreover, there are situations, typical in distributed supply chain planning, where agents are responsible for solving heterogeneous problems that have producer-consumer type interdependencies [2].

This paper studies distributed problems that are hierarchical in nature. As for classic DCOP, there is an objective function that agents collectively try to minimize. However, unlike DCOP:

- The problem is partitioned into subproblems and there is a sequence in which they must be solved, imposed by the business domain;
- The agents are constrained when solving a subproblem by the decisions of the previous ones;
- The global objective can be formulated as a function of the variables of one subproblem (most of the time the last one).

These characteristics are more generally shared by hierarchical production planning problems [3]. Schneweiss describes several of these problems in [4].

More specifically, this paper studies the case of industrial supply chain where agents represent factories offering services to the others agents [5]. An external client announces a call for bids for a product and the work of each plant is needed to produce and deliver the final good (Figure 1). Different alternatives are possible regarding the parts to use, the manufacturing processes to follow, the scheduling of operations and the choice of transportation. The partners want to put together a common production plan (e.g. what to do, where and when).

The common objective function represents the client's interest, e.g. minimize lateness. If the external client rejects the first proposal, it is urgent to propose alternative solutions before it accepts a proposition from another consortium. Our goal is therefore to produce good solutions quickly.

Figure 1: Simple supply chain

However, the factories of the consortium may be competing against each other for other projects. Therefore, privacy is an important issue and each factory wants to plan its own activities. Each agent has a local vision of the overall problem. It can only think about its own production, inputs and outputs. The agents producer-consumer type of interdependence defines implicitly a resolution sequence; a factory cannot computes its needs for raw material and send them to its supplier without knowing what it is asked to produce, neither can it plan its production before knowing what supply is granted by the supplier. Figure 2 illustrates the sub-problem structure and the solving sequence required to handle their interdependence.
This two-phase business coordination process is needed because it may be impossible for an agent to satisfy all the needs of other agents. For example, some deliveries may be planned to be late or some products can be replaced by substitutes.

Finally, because agents do not know the alternative options of the agents they supply, a bound on the objective function cannot be computed until the last subproblem is solved (e.g., for an agent, late supply deliveries do not mean it cannot satisfy its own customer on time using alternative production processes).

In the following sections, we will formally describe the problem and show that few classical DCOP methods apply in this context. We will then propose MacDS, a concurrent method allowing the agents to systematically explore the solution space, but aiming at producing good solutions first by using a backtracking strategy based on the computation of discrepancies. The algorithm will then be evaluated for both random data and real industrial problems.

2. Problem definition

A Hierarchical Distributed Constraint Optimization Problem (HDCOP) is defined over a set of variables \( X = \{X_1, \ldots, X_n\} \). Each variable \( X_i \) can take a value from a set \( D_i \). The set of variables \( X \) is partitioned into disjoint subsets (subproblems). Each variable is part of a subproblem \( S(X_i) \in S = \{S_1, \ldots, S_n\} \). Each variable and subproblem is owned by an agent \( A(X_i) \in A= \{A_1, \ldots, A_k\} \). Subproblems must be solved in the order defined by \( S \). Each subproblem \( S_i \) may be constrained by previous decisions, as stated by a predicate \( C_j(\bigcup S_i) \). Agents wish to minimize a function \( F(S_q) \) where \( S_q \in S \).

We suppose that agents know algorithms to solve their subproblems and they produce the alternative solutions in an order defined by this algorithm.

In that context, we can represent the global solution space as a tree, where each level \( j = 1, \ldots, n \) corresponds to subproblem \( S_j \). Each node on that level represents an instance of that subproblem (defined by previous decisions). Each arc is an alternative and feasible solution to that subproblem. Arcs are ordered according to the local algorithm used by the agent. Each leaf is a global solution, therefore an alternative proposition to the external customer (see Figure 3).

2.1 DCOP

More generally, a Distributed Constraint Optimization Problem (DCOP) [6] is defined over a set of variables \( X = \{X_1, \ldots, X_n\} \). Each variable \( X_i \) can take a value from a set \( D_i \). Each variable is owned by an agent \( A(X_i) \in A = \{A_1, \ldots, A_k\} \). There is a set of relations \( C = \{C_1 \ldots C_n\} \) where \( C_j \) is a function \( C_j(Y_1, \ldots, Y_q) : D_1 \times \ldots \times D_q \rightarrow \mathbb{R}^+ \) defined for a subset of variables \( \{Y_1, \ldots, Y_q\} \in X \). The agents want to minimize the sum of all \( C_j \in C \).

The HDCOP problem we defined previously could be reformulated as a classic DCOP. In order to keep the notion of subproblems, we need to consider that each subproblem in the original HDCOP formulation is now a variable in the DCOP. The domain of each new variable is the Cartesian product of the original variables. The original variables are no longer considered. Unfortunately, this representation does not permit the modeling of the required solving sequence. We also miss to represent the fact that each time a variable must be valued a complex subproblem must be solved. Finally, for the problems we studies, an agent has the right to propose the alternative solutions in a sequence defined by its own local algorithm. In contrast, most DCOP algorithms specify the rule an agent must use to value its variable/sub-problem; agent must mandatory choose the value that minimize the sum of the relations \( C_j \) it is involved in. For these reasons, we will prefer the HDCOP definition.

2.2 Algorithms for Classical DCOP

The simplest algorithm is Synchronous Backtracking (SyncBT) [7]. It mimics chronological backtracking in the tree representing the solution space (for our HDCOP, the tree presented in Figure 3). Agents solve the subproblems in sequence. The messages transmitted by agents
represent current partial assignments (CPA). In the case of a dead end, or when a global solution is found, a
chronological backtrack occurs. Of course, the method is
complete. SyncBT can be applied as is for the introduced
HDCOP. Synchronous Branch-and-Bound (SyncBB) improves
SyncBT by calculating a bound on the objective in
each node. The value is used to prune the tree and
guide the value ordering [8]. For a hierarchical problem
where the objective is represented by a function of the
variables of the last subproblem and the agents lack a
good representation of the other subproblems, the bound
would be equal to zero at each node. SyncBB is then
equivalent to SyncBT. In Asynchronous Forward-
Bounding (AFB) [9], when an agent assigns a value it
transfers the CPA to every following agent. Agents then
compute bounds concurrently.

Some methods allow the agent to assign a value to
variables asynchronously, as soon as it supposes this
change could improve the global solution. Distributed
local search does this but it is incomplete [8,10].
Asynchronous Distributed Optimization (ADOPT) [6,11]
was the first method both asynchronous and complete but
the algorithm needs agents to be able to calculate good
bounds and change their values accordingly. It therefore
violates our assumption about each agent using a
specialized local solver, producing solutions in a sequence
of its choice.

Another approach is distributed dynamic programming.
The best known algorithm is DPOP [12] for which different improvements have been proposed
over time. It makes the assumption that agent \( A(S_j) \) can
solve \( S_j \) before \( S_{j+1} \) is solved. \( A(S_j) \) is asked for the best
solution for \( S_j \) for any potential solution of \( S_{j+1} \). This
supposes that \( A(S_j) \) has a good representation of the
domain for \( S_{j+1} \) and can solve \( S_j \) to optimality in
reasonable time.

3. Multi-agent Concurrent Discrepancy
Search (MacDS)

The search tree for the introduced Hierarchical
Distributed Optimization Problem (HDCOP) is equivalent
to one for a centralized problem given a variable ordering
heuristic and a value ordering heuristic. In such a
centralized context, chronological backtracking is most of
the time outperformed by search methods based on the
computation of discrepancies, both for satisfaction and
optimization problems [13,14]. These strategies have the
characteristic of not relying on bound calculation.

This section introduces a new algorithm (MacDS) that
performs distributed search based on discrepancies. It is
complete (exploring the same search space as SyncBT)
but aims at producing good solutions in a short amount of
time. It allows concurrent computation, uses
asynchronous communication, an asynchronous timing
model [15] and is tolerant to random message
transmission delays. It also takes advantage of situations
where subproblem solving times vary from one agent to
another.

3.1 Centralized Discrepancy-based Methods

Limited Discrepancy Search (LDS) was the first method
based on discrepancies [13]. The main idea is that the
leaves of the tree (solutions) do not have the same
expected quality; that it decreases with the number of
times one branches to the right when going from the root
to the leaf (i.e. the number of discrepancies). The
rationale is that a move to the right is a move against the
value ordering heuristic. LDS aims to first visit the leaves
with the fewest discrepancies. Another effect of LDS is
that the solutions visited in a given period of time will be
from more different part of the tree than those produced
using chronological backtracking. It is this interesting
characteristic we seek in our distributed algorithm. In the
original description by Harvey, LDS was a search
procedure, but the idea can be used to specify a node
selector: when backtracking conditions occurs, the search
engine must select the node for which the next unvisited
child has the fewest discrepancies [16]. This is why LDS
can be described as a backtracking policy.

LDS has been applied with success to optimization
problems in [14]. For n-ary trees, they proposed to count
the discrepancies as follows: the \( i \)-th arc followed at a
given level counts as \( i - 1 \) discrepancies. Over time, other
discrepancy-based methods have been proposed [16,17].
Discrepancy-based methods are integrated into
commercial solvers as, for example, ILOG Solver.

3.2 Concurrent Search Algorithms

Classic algorithms for DCOP and DisCSP (Distributed
Constraint Satisfaction Problems) exploit two different
approaches to achieve concurrency [18]. The first is the
asynchronous approach used by ADOPT, as described
previously. The second is called Concurrent Search
Algorithm (CSA) by [18]. They exploited it in the
ConcDB algorithm (for DisCSP). With this approach,
each global solution is constructed sequentially by the
agents, but agents collectively work on many solutions. A
concurrent algorithm must specify how and when a new
“path” must be explored. Our algorithm makes use of this
form of concurrency.

3.3 Proposed Algorithm (MacDS)

We will first describe the algorithm informally using a
simple example. Agents \( A = \{B,C,D\} \) should solve the
problem made up of subproblems \( S = \{B,C,D\} \). Each
global solution is constructed sequentially by the agents,
by solving the subproblems as ordered in the vector. The
first agent (B) uses its local solver. Once it has a first local
solution, it sends it to agent C in a message named “B0”
(first local solution for subproblem B). Agent C then do
the same and transmits the message “B0;C0” (first local solution of C according to the first local solution of B) to the next agent. But, as soon an agent has sent a solution to the next one, it started looking for an alternative decision and then sends it asynchronously to the next agent. Figure 4(i) shows a state where agent B is working on a second local solution. Agent C is working on a second local solution based on the first one of agent B. Agent D is working on a third local solution. In Figure 4(ii), agent C have come up with a second local solution (named “B0;C1”) and sent it to agent D. When receiving this message, agent D must ask itself whether it is better to produce a third solution based on its left node (that is, “B0;C0;D2”) or produce a first solution based on its right node (“B1;C0;D0”). This decision must be based on discrepancies (the names of the local solutions contain information to compute them). If we want to apply an LDS policy, the agent will choose the right node (Figure 4(iii)) because it will generate a solution having fewer discrepancies.

![Figure 4: Part of a MacDS execution trace](image)

Essentially, each agent manages a list of tasks, supplied asynchronously by the previous one. Tasks are prioritized locally according to the discrepancies of the next message the task would generate. The policy used to compare them defines a backtracking strategy that will be enforced collectively by all agents. It is possible to implement different known strategies, such as LDS, Depth-bounded Discrepancy Search (DDS) [17], or even chronological backtracking.

In the extreme case where a single agent owns every subproblem, then MacDS visits the nodes of the tree in the same order as a centralized search algorithm (applying the same backtracking strategy) would.

In a distributed context where each agent manages its own task list, each solution to the global problem will be obtained in no more time than would be necessary for the centralized algorithm (ignoring communication delays). Each agent is working as soon as there is a task in its list, but preempts its current work when a task with greater priority is added to the list.

The algorithm is complete since it explores the same solution space as SyncBT; it only changes the sequence in which nodes will be visited. In the case of communication breakdown (or in presence of random message transmission delays), the agent always works on the available task with the greatest priority, rather than remaining idle. Therefore, it is not mandatory for messages to arrive in the same order as they were sent.

3.3.1 Pseudocode

The following objects are manipulated by the algorithm:
- A message msg is a couple <d, p> where d represents the decisions for the previous subproblems and p is a vector of integers representing the ‘path’ leading to the corresponding node in the global tree. The element p[j] defines, for a level j, which arc should be followed when going from the root to that node.
- A list of tasks (tasks) contains the running and waiting tasks of the agent. A task is defined by d and p, by the number of local solutions produced to date for the task (i) and by a boolean indicating if the subproblem solver thinks there is no more solution (noMoreSol). A task corresponds to a node in the global tree.

Each agent runs many threads: one for each task plus a control thread. A single thread per agent is active at any moment. The control thread (Figure 5) is activated when the agent receives a message (WhenReceiveMsg) and when a task has just produced a new subproblem solution (WhenNewSolution). It then updates the task list and transfers control to the thread of the task with the greatest priority (ActivateTask).

![Figure 5: Control thread of the agent](image)

The pseudocode for the task threads is shown in Figure 6. When a task is created, its thread is idle. It must be activated by the control thread. When the task produces a new subproblem solution, it signals it to the control thread (SignalNewSolution) and goes idle (sleep). It is the control thread that sends the message to the agent that owns the next subproblem.
Run(task)
    task.noMoreSol ← false;
    task.sol ← NextSolution(task);
while (task.sol ≠ ∅)
    SignalNewSolution(task);
    Sleep();
    task.sol ← NextSolution(task);
    task.noMoreSol ← true;
    SignalNewSolution(task);

Figure 6: Thread implementing a task

CompareBT(p1, p2)
    depth ← Min(Card(p1), Card(p2));
    j ← 1;
    while (p1[j] = p2[j] && j ≤ depth) j++;
    if (j ≤ depth)
        if (p1[j] ≤ p2[j]) return p1;
        else return p2;
    else
        if (Card(p1) ≥ Card(p2)) return p1;
        else return p2;

CompareLDS(p1, p2)
    t1 ← Σ[j=1..Card(p1)] p1[j];
    t2 ← Σ[j=1..Card(p2)] p2[j];
    if (t1 < t2) return p1;
    else
        if (t2 < t1) return p2;
        else return CompareBT(p1, p2);

Figure 7: Two different backtracking policies

In the shown pseudocode, we suppose that tasks are ordered by decreasing priority. Figure 7 shows examples of comparator functions that can be used to maintain the list sorted. They identify, over a pair of task which one has the highest priority, according to a backtracking strategy. The first one (CompareBT) defines a chronological backtracking policy. The other one (CompareLDS) defines an LDS policy. The arguments of the functions are the ‘path’ in the global tree of the next local solution each task would generates (that is, the concatenation of task.p and task.i).

4. Evaluation

MacDS was evaluated with generated data and with real industrial problems.

4.1 Industrial Evaluation

We evaluated the algorithms using real industrial data with complex subproblems. The data were extracted from the company databases at different moments in 2005.

The case is a supply chain coordination problem in the forest products industry. The network has three facilities (Sawing, Drying and Finishing). The business coordination process is the one described in Figure 2. Each local solver was made available by FORAC, a consortium of companies and researchers. The Sawing agent plans its operations using a Mixed Integer Linear Programming model. The Drying agent uses Constraint Programming. The finishing agent uses a forward-scheduling heuristic, but uses Depth-bounded Discrepancy Search (DDS) [17] to produce alternative

Figure 8: Reduction of the objective function, according to computation time (in seconds) for cases (i), (ii), (iii) and (iv)
solutions. More information about the problems and the solvers can be found in [19].

These experiments were done in a distributed environment where each agent runs on a different computer. It allows measuring the real impact of concurrency in a situation where subproblems are hard and take a different amount of time to be solved. The difference in computation time between subproblems can vary five-fold. Production of alternative solutions varies from a few seconds to several minutes. In this context, communication complexity is not an issue [15].

We compared MacDS using an LDS policy (MacDS_LDS) to SyncBT (section 2.2). To measure which part of the gain was due to concurrency and which to the backtracking policy, we also tested a version of MacDS applying chronological backtracking (MacDS_BT) and also implemented SyncLDS (synchronous as SyncBT, but performing LDS).

For a given case, the first global solution of each compared algorithm is always the same. It is also this solution that would be obtained by the companies using the standard business coordination process (Figure 2). Consequently, we compared the algorithms according to the reduction of the objective function they achieved with additional computation time (0 to 3600 seconds).

Figure 8 shows the results for the four industrial cases studied (i,ii,iii,iv). For a very short computation time, SyncBT and MacDS_LDS give comparable results. This is because MacDS_LDS produces the same solution than SyncBT until the last agent receives a second task in its list. Then, MacDS_LDS starts to outperform SyncBT in a significant manner: while SyncBT persists in exploring only minor variations of the first solutions, MacDS_LDS explores different areas of the search tree. Case (ii) is an exception: SyncBT is the winner by 0.5% given a computation time greater to 1000 sec.

We can see the impact of the backtracking policy solely by comparing SyncBT and SyncLDS. The latter outperforms SyncBT, except for very short computation time.

To see the impact of concurrency, we compared MacDS_LDS with SyncLDS. For any solution quality reached by SyncLDS, MacDS produces an equal or better solution in an equal or shorter computation time. The average reduction of computation time for each case is as follows: 18.5%, 88.7%, 54.5% and 64.0%. Two reasons explain this. Concurrency makes each solution being produced in an equal or shorter amount of time, but it also gives agents the opportunity to explore more alternative solutions in a given amount of time.

MacDS_BT gave results indistinct from those of SyncBT. For that reason, they are not shown in Figure 8. The explanation is the following: for industrial cases, subproblems have many alternative solutions. When performing chronological backtracking, the system takes a long time to explore alternative solutions of the last subproblem only. In MacDS_BT, previous agents produce alternative solutions that are never exploited by the last agent in the given computation time.

4.2 Evaluation with Generated Data

We then evaluated MacDS simulating a wide range of problems. We randomly generated n-ary trees such that the probability that a leaf is the best solution is proportional with $\delta(p)$, where $p$ is the number of discrepancies of the leaf and $\delta$ is a parameter that varies from 0.1 to 1.0 (by steps of 0.1). When $\delta$ is maximal, all leaves have the same probability. This is a similar context in which LDS was originally evaluated for centralized problems [20].

Others parameters we experimented with were: number of subproblems ($\text{Card}(S)$), message transmission delay ($\tau$), subproblem solving time ($\alpha$), and the number of solutions for each subproblem ($n$). This experiment was conducted in a simulation environment similar to [6].

The performance measure used is the expected time needed to find the best solution.

MacDS_LDS is always the best of the four algorithms (caught up by MacDS_BT when $\delta=1.0$). For both

![Figure 9: Expected time to get best solution, according to: (i) message transmission time $\tau$ [with $\text{Card}(S)=4; n=10; \alpha=1; \delta=0.7$], (ii) time to solve subproblem $\alpha$ [with $\text{Card}(S)=4; n=10; \tau=0; \delta=0.7$] and (iii) total number of subproblems $\text{Card}(S) [n=3; \alpha=1; \tau=0; \delta=0.7]$]
backtracking policies (BT and LDS), the MacDS version always beats the synchronous one.

Figure 9 shows results when \( \delta = 0.7 \), which is representative of the average case. Subfigure (i) shows that message delay (\( \tau \)) has a linear impact for all four algorithms, but a much smaller one for both versions of MacDS. For them, expected time needed to find the best solution is equal to \( (\text{Card}(S) - 1) \tau \). Subfigure (ii) shows that subproblem solving time (\( \alpha \)) also has a linear impact, again smaller for the MacDS versions. Figure 9(iii) shows that the impact of the number of subproblems (\( \text{Card}(S) \)) is exponential.

5. Conclusion

We demonstrated the impact of distributed search methods for real industrial supply chains problems. Even a basic method (SyncBT) shows considerable reduction of lateness for a real network of companies.

We proposed MacDS, a distributed algorithm that performs search based on discrepancies and allows the agents to work concurrently. It outperforms SyncBT: (1) by applying backtracking policies based on discrepancies and (2) by allowing the agents to work concurrently, which reduces idle time for the agents.

For an equal solution quality, MacDS also shows considerable reduction of computation time, when compared to a non-concurrent synchronous algorithm applying the same backtracking policy (LDS).

In the near future, it would be interesting to measure if discrepancy-based search could be helpful in situations where bound computation is possible. Discrepancy-based backtracking coupled with pruning based on bounds has shown good results in a centralized environment [16].

8. References


