CIRRELT

Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

ADS: An Adaptive Search Strategy for Efficient Distributed Decision Making

Jonathan Gaudreault
Gilles Pesant
Jean-Marc Frayret
Sophie D'Amours

November 2008

CIRRELT-2008-49

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
Pavillon Palasis-Prince, local 2642
Québec (Québec)
Canada G1K 7P4
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

ADS: An Adaptive Search Strategy for Efficient Distributed Decision Making

Jonathan Gaudreault[1,2,*], Gilles Pesant[1,3], Jean-Marc Frayret[1,3], Sophie D'Amours[1,2]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Département de génie mécanique, Pavillon Adrien-Pouliot, Université Laval, Québec, Canada G1K 7P4

[3] Mathematics and Industrial Engineering Department, École Polytechnique de Montréal, C.P. 6079, succursale Centre-ville, Montréal, Canada H3C 3A7

Abstract. This paper concerns distributed decision-making in hierarchical settings. For this class of problems, the coordination space can be naturally modeled as a tree. A collective of agents can thus perform a distributed tree search in order to coordinate. Previous results have shown that search strategies based on discrepancies (e.g. LDS) can be adapted to a distributed context. They are more effective than chronological backtracking in such setting. In this paper we introduce ADS, an adaptive backtracking strategy based on the analysis of discrepancies. It enables the agents to collectively and dynamically learn which areas of the tree are most promising in order to visit them first. We evaluated the method using a real coordination problem in an industrial supply chain. This makes it possible for the team of agents to obtain high-quality solutions much more quickly than with previous methods.

Keywords. Supply chain coordination, multi-agent, distributed search, discrepancy, adaptive.

Acknowledgements. This work was funded by the FORAC Research Consortium and the Natural Sciences and Engineering Research Council of Canada (NSERC).

---

* Corresponding author: Jonathan.Gaudreault@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec,
Bibliothèque et Archives Canada, 2008

1 Introduction

Within the scope of this paper, we are interested in multiagent coordination in hierarchical settings. These are distributed optimization problems showing the following characteristics: (1) the global problem is naturally decomposed into subproblems, (2) there exists a sequence defined *a priori* in which the subproblems must be solved, (3) various agents are responsible for the subproblems, and (4) each subproblem is defined according to the solutions adopted for the preceding subproblems. Schneeweiss describes many such problems in [1]. *Organizational Distributed Decision Making* and *Supply Chain Coordination* are among the main application domains.

In such a context, the capacity of a team of agents to identify good solutions is related to the coordination mechanism used. The most commonly used mechanisms can be described as heuristics. The best known are certainly *upstream planning* and other related approaches [2]: subproblems are solved one after another, according to the predefined sequence. By doing so, agents obtain one and only one solution that may not be optimal from a global point of view.

In an optimization context, it is advisable to consider several alternative solutions. To this end, the coordination space can be modeled as a tree [3]. A team of agents can thus perform a distributed tree search in order to coordinate. The capacity of the team to find high-quality solutions quickly depends on the search strategy used. Section 2 provides more details on this concept.

In the present work, we put forward the hypothesis that there exist backtracking strategies that are particularly effective in a situation of hierarchical distributed problems. Section 3 introduces an adaptive search strategy called ADS. During the search process, agents collectively and dynamically identify the most promising areas of the tree in order to explore them first. This allows agents to systematically search the solution space (thus looking for the optimal solution) but aims at producing good solutions in a short amount of time. The algorithm is evaluated (Section 4) on a real coordination problem in an industrial supply chain, and for synthetic problems in order to allow more general conclusions. Finally, Section 5 situates our approach in relation to others that use related techniques, but in the context of classical global optimization.

2 Background

2.1 Distributed Decision Making in a Hierarchical Context

Figure 1 illustrates a coordination problem in the context of a supply chain. The protagonists (agents) are faced with a common optimization problem. The cooperation of each facility is needed to produce and deliver the products ordered by external customers. However, different alternatives are possible regarding the parts to use, the manufacturing processes to follow, the scheduling of operations and the choice of transportation. Therefore, supply chain partners must coordinate their local decisions (e.g. what to do, where and when), with the common objective of delivering the products ordered by an external customer with the least possible delay. This can be referred to as *collaborative production planning* (see [4]) or the *supply chain coordination problem* [5].

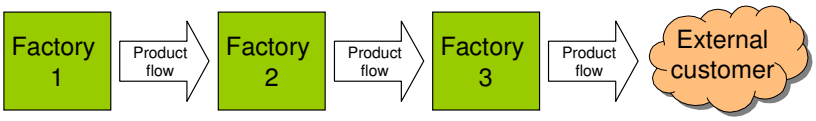


Figure 1. Product flow for a simple supply chain

The general assignment of responsibilities to agents and the decomposition of the overall problem into subproblems are usually determined by the business context or by a long-term agreement in force in the industry. This decomposition can be illustrated using a workflow diagram. As an example, Figure 2 illustrates the application of a mechanism based on the exchange of demand plans and supply plans to the preceding case. This mechanism is called *two-phase planning* [6]. Each unit is responsible for establishing two things: an ideal demand plan for raw material, and the actual production plan, produced once the

"answer" from the supplier is received. The sequence according to which agents intervene is shown in Figure 2. Because there are more subproblems than agents, here the term *hierarchy* refers to the sequence of subproblems, rather than a sequence of agents.

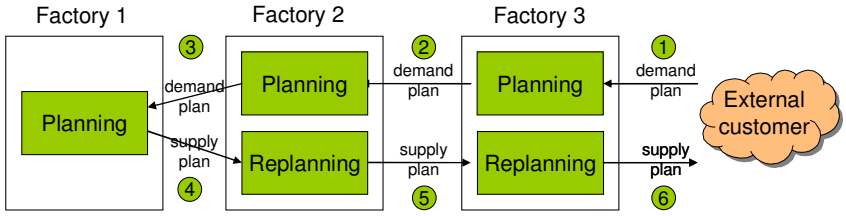


Figure 2. *Two-phase planning* protocol as coordination mechanism for the previous supply chain

In [1], Schneeweiss describes a general framework in order to model these kinds of problem hierarchies. According to this framework, the decision taken at a given level (*top*) is interpreted by the following level (*bottom*) as a parameter of the subproblem *bottom*. Stated otherwise, the subproblem *bottom* is defined as a function of the *top* decision. This can be generalized for situations with more than two subproblems. The subproblems are solved sequentially, from top to bottom, which is termed *upstream planning* [2,7]. This supposes that for each subproblem, the corresponding agent has access to a specific model/algorithm which allows solving the subproblem.

By doing so, the collective considers only one global solution which is unlikely to be optimal. This is why we termed these kinds of approaches *hierarchical coordination heuristics*. Of course, the better the top agent's ability to anticipate the reaction of the bottom agent is, the better the global solution will be (see [8] for a computational study). However, more general coordination mechanisms can be considered.

The following section recalls how hierarchical coordination heuristics can be generalized in order to consider a larger solution space.

2.2 Hierarchical Distributed Constraint Optimization (HDCOP)

The algorithms used in industry to make each local decision usually allow producing alternative solutions [4]. By considering each of these as an alternative proposition for the next agent, we can represent the coordination space as a non-binary tree of fixed depth (see Figure 3) [3].

This tree has one level per type of subproblem. For example, if we want to generalize the *two-phase planning* approach, each level of the tree will correspond to one of the boxes in Figure 2. Each node on a specific level represents an instance of that subproblem type (defined by decisions for previous subproblems). Each arc is an alternative and feasible solution to the subproblem. The number and order of these arcs depend on the local algorithm used by the agent. Each leaf of the tree (i.e., a solution of a subproblem of the last/lower level) is a solution to the global problem.

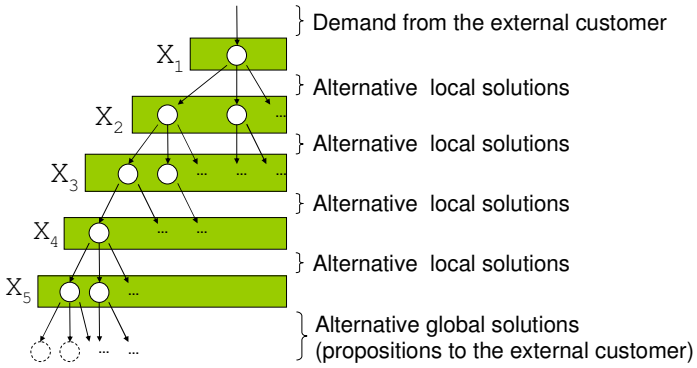


Figure 3. Coordination Space for HDCOP

The problem is then modeled as a *Hierarchical Distributed Constraint Optimization Problem* (HDCOP). We reproduce here the definition taken from [3]: The global problem is defined by a vector of subproblems: $X = [X_1, ..., X_M]$. Each subproblem X_i comes under the responsibility of an agent $\mathcal{A}(X_i) \in A = \{A_1, ..., A_N\}$. For each subproblem X_i the agent $\mathcal{A}(X_i)$ has a solver $\mathcal{S}^i$ producing a vector S^i of alternative solutions: $\mathcal{S}^i(X_i) \rightarrow S^i$ where $S^i = \left[S_1^i, \ldots, S_{|S^i|}^i\right]$. These local solutions are not known *a priori*; they are revealed one after another by the solver. Eventually one gets selected. We will denote it by S_*^i. Agents look for the vector $\left[S_*^1, ..., S_*^M\right]$ minimizing an objective function $\mathcal{F}\left(\left[S_*^1, ..., S_*^M\right]\right)$. Each subproblem X_i is defined by the chosen solutions for the previous subproblems: $X_i = \mathcal{G}^i\left(\left[S_*^j \middle| 1 \leq j < i\right]\right)$.

2.3 Distributed Search as Coordination Mechanism

The previous formulation for the coordination problem calls for its optimization using a standard tree search algorithm. Because of the nature of the coordination problem at hand, such an algorithm must satisfy three conditions. First, it must be distributed. Second, it must respect the business relationships between the partners. Finally, it must take into account that the tree is not fully defined *a priori*. That is, (1) the alternative solutions (arcs) of the local subproblems are not known before being produced by the local solver, and (2) the specific instances of each subproblem (node) are not known either, because they depend on the solution obtained for previous subproblems. The tree will therefore be 'revealed' progressively during the search process.

The simplest method for the agents to collectively explore this coordination space is to perform what Hirayama and Yokoo term *Synchronous Branch and Bound* (SyncBB) [9]. Agents solve their subproblems in sequence: the first one solves its subproblem and sends its decision to the second agent, and so on. In the case of a dead-end, or when an agent has considered all of its local solutions, this agent sends a message back to the previous agent, asking for an alternative proposition (this is termed *chronological backtracking*). This backtracking message contains the value of the best solution found so far. It is used during search to 'prune' the tree. The method is said to be *complete*, as it returns the optimal solution if given enough time.

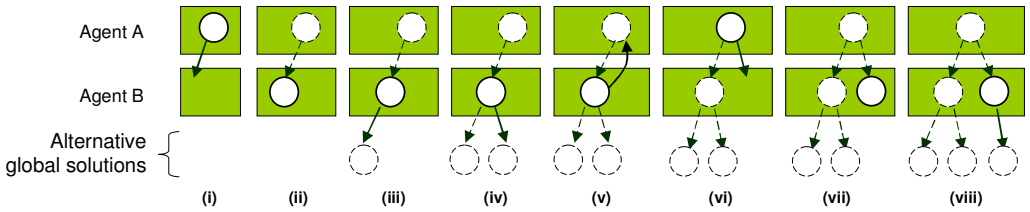


Figure 4. Example of execution trace for SyncBB

2.4 Discrepancy-based Backtracking Policies

In *centralized* contexts, chronological backtracking is often outperformed by other backtracking strategies, for example those based on the computation of *discrepancies* [10,11] such as *Limited Discrepancy Search* (LDS). The next section describes LDS. We then recall how this idea can be used in a distributed context.

2.4.1 LDS – Discrepancies in Centralized Contexts

Limited Discrepancy Search (LDS) was the first search method based on *discrepancies*. It was developed for *centralized* combinatorial problems and was introduced by Harvey and Ginsberg in [10,12]. The basic idea is based on the observation that for many problems, the leaves of the tree (solutions) do not all have the same expected quality. That is, the expected quality is related to the number of times one branches to the right when going from the root of the tree to that leaf (the *number of discrepancies* of the leaf). Figure 5 shows a simple tree and the number of discrepancies associated to each leaf.

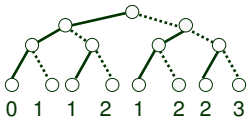


Figure 5. Binary tree (two arcs per node) with the associated number of discrepancies for each leaf. It corresponds to the number of times one branches to the right when going from the root to that leaf (dotted arcs on the figure).

Harvey measured that for many problems the expected quality of a leaf decreases when the number of discrepancies increases [12]. The explanation for this relation is as follows. In most trees, the arcs (alternative local solutions, in our case) are ordered according to a criterion. Therefore, each move 'to the right' is a move against that criterion. Consequently, he proposed an iterative procedure (LDS) that aims to first visit the leaves with fewest discrepancies. Another effect of applying LDS is that the solutions visited in a given period of time will be from more different parts of the tree than those produced using chronological backtracking. For non-binary trees, they proposed to count the discrepancies as follows: the *i*-th arc followed at a given level counts as $i-1$ discrepancies [11].

LDS was introduced as an iterative search procedure, but the same concept is often used to define a backtracking strategy (or policy) to be used within a generic centralized search engine [13]: The number of discrepancies can be computed for any node of the tree, not just for the leaves. When backtracking conditions occur (we encounter a global solution and aim for another) the solver 'backjumps' to the node already visited for which the next unvisited child has the fewest discrepancies.

2.4.2 SyncLDS - Discrepancies in Distributed Contexts

The simplest method for a team of agents to perform a distributed LDS is called SyncLDS [3]. The search is performed in a scheme similar to SyncBB, except that when a global solution is found, the agent detecting that condition sends a message to other agents asking each of them to identify the subproblem/node under its jurisdiction for which the next child has the smallest number of discrepancies. This can be seen as a kind of bid. The agent that detects the need for backtracking gives control to the agent with the smaller bid. The next global solution is reached from that point by solving the remaining sequence of subproblems.

We evaluated this method (SyncLDS) in [3]. It allowed considerable improvement of solution quality and computation time for a distributed supply chain problem in the forest products industry. The following is the main reason why it outperformed SyncBB: Once the first global solution is found, SyncBB tries enumerating every other alternative solution for the last subproblem without considering other options for the previous subproblems. By doing so, it persists in exploring only minor variations of the first global solution. In contrast with this, SyncLDS explores rapidly different areas of the tree.

In the same paper, we proposed another distributed implementation of LDS called MacDS. This implementation allows agents to work concurrently, which speeds up computation. However, in the remainder of this article we will not address the question of concurrency. Emphasis is put on the comparative evaluation of backtracking strategies only.

3 Proposed Method: Adaptive Discrepancy-based Search (ADS)

In the previous section, we recalled that SyncLDS outperforms SyncBB because it allows rapidly visiting different areas of the tree. This leads us to formulate the following hypothesis: If it were possible to identify interesting areas in a dynamic way, and to first concentrate our efforts on them, the effectiveness of the search process would be improved.

In this section, we introduce an adaptive search strategy called ADS. During the search process, agents collectively and dynamically identify the most promising areas of the tree in order to explore them first. It allows agents to systematically search the solution space (thus looking for the optimal solution) but aims at producing good solutions in a short amount of time.

The strategy exploits the fact that the tree is not binary. Since each subproblem (node) has many solutions (arcs), we have to backtrack many times to each node during the search process. Each time we will do so,

we will measure how beneficial it has been to produce an *i*-th solution for the corresponding subproblem (that is, how beneficial it has been to allow an additional discrepancy). We then seek to extrapolate for which node it would be more profitable to produce other local solutions, and how many should be generated before it becomes better to pass on to another node.

The basic concepts are introduced informally in Section 3.1. We will then introduce a model that allows extrapolating the contribution associated to performing an *i*-th discrepancy at a given node (3.2). Finally, we will define a backtracking strategy that uses this information (3.3), and a protocol that allows agents to implement it in a distributed scheme (3.4).

3.1 Main Idea

We will consider a fictive minimization problem represented by a non-binary tree. Figure 6i shows the part of the tree that is known after reaching the first global solution. Let us suppose this solution's quality (as measured by the objective function) is equal to 1.0. At that moment, three nodes are candidates for backtracking (**a**, **b** and **c**). Backtracking to any of them would lead to a new global solution having a number of discrepancies equal to 1. If we were to apply an LDS policy, we would not have a preference for any of them. Let us suppose we backtrack to each of them in turn. We then get three new solutions of quality 1.2, 0.6 and 0.9 (subfigure ii).

We now have 6 nodes (**a** to **f**) that are candidates for backtracking[1]. Again, these nodes are of equal interest from the point of view of an LDS policy (each would lead to a leaf having 2 discrepancies). However, we can see that for node **c**, the second arc led us to a leaf for which the solution quality is worse than for the first arc (1.2 > 1.0). For node **b**, the second arc produces an improvement of 0.4 (1.0 – 0.6). For node **a**, the improvement is 0.1 (1.0 – 0.9).

In short, although nodes **a**, **b** and **c** are equally appealing from an LDS point of view, when considering recent history it seems more interesting to produce a third solution from node **b**. As for nodes **d**, **e** and **f**, we have no information available but they could lead to promising areas. We thus have to make a choice between exploiting available information (and choosing node **b**) or exploring new areas. The following sections develop this idea.

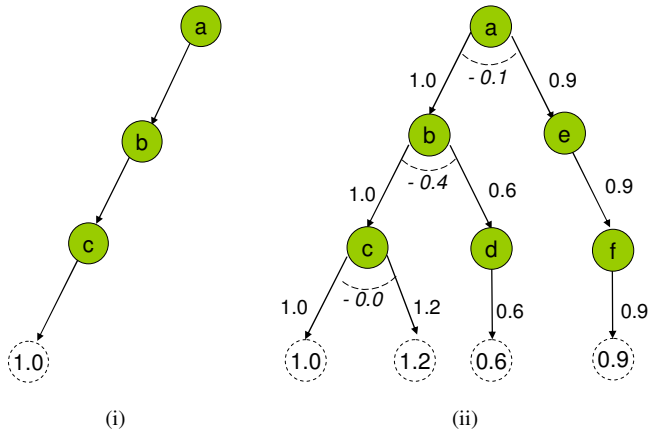


Figure 6. Illustrating the main idea

3.2 Extrapolate the Effect of Performing an i-th Discrepancy at a Given Node

Let us suppose that a search is in progress. An *i*-th global solution has just been found and we need to backtrack in order to explore alternative solutions. Let *nodeList* be the list of nodes *node* that are candidates for backtracking (i.e. nodes for which there are unexplored local solutions for their local subproblems). For each $node \in nodeList$, we will denote by n the number of local solutions already

[1] Nodes **a, b** and **c** are still candidates since the tree is non binary (there are more than two solutions for each subproblem)

explored (i.e. number of arcs exiting from this nodes already explored). We will define the following terms in the context of a specific *node*:

Definition 1 : $\text{Child}(i)$ corresponds to the node headed by the *i*-th arc of *node*. It is defined for $i = 0...n-1$. $\text{Child}(i)$ may be a leaf or an internal node. As an example, on Figure 6ii, for node **b** we have $n = 2$, $\text{Child}(0)$ corresponds to node **c** and $\text{Child}(1)$ to node **d**.

Definition 2: $\text{NextLeaf}(i)$ corresponds to the first leaf we would encounter performing a depth-first search in the subtree having $\text{Child}(i)$ as its root. If $\text{Child}(i)$ is a leaf, then $\text{NextLeaf}(i)$ returns this leaf. Otherwise we have $\text{Leaf}(i) := \text{Child}(i).\text{NextLeaf}(0)$.

Definition 3 : $\text{ArcValue}(i)$ corresponds to the quality of the first global solution that can be reached by following arc *i*. That is $\text{ArcValue}(i) := \text{NextLeaf}(i).score$.

As an example, each arc on Figure 6ii is labeled by its value. One might use other definitions for $\text{ArcValue}(\ )$ (e.g. best solution in the subtree). However, the definition we proposed has the following advantage. Each time we backtrack to a node, we can measure the quality of the followed arc as soon we reach the next leaf (that is before the next backtrack) and this value remains unchanged for the rest of the search, which limits model updates.

Definition 4 : $bestToDate[\ ]$ is a vector of size *n* such as $bestToDate[i] = \min_{j=0}^{i} \text{ArcValue}(i)$.

Figure 7 illustrates the relationship between $\text{ArcValue}(\ )$ and $bestToDate[\ ]$. It shows a node for which $n = 4$ (subfigure i). Each arc $i = 0...3$ is labeled by its value $\text{ArcValue}(i)$. Subfigure ii illustrates the apparent difficulty of extrapolating a value for $\text{ArcValue}(4)$. In contrast, $bestToDate[\ ]$ is monotonic non-increasing (iii). It seems easier to extrapolate a value for $bestToDate[4]$.

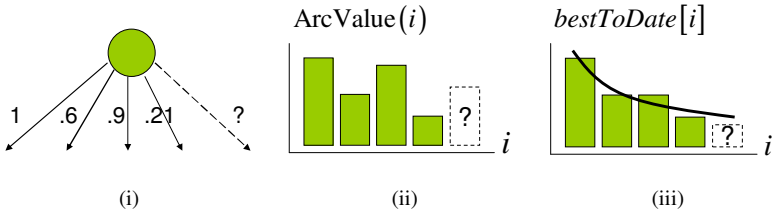


Figure 7. Relation between $\text{ArcValue}(\ )$ and $bestToDate[\ ]$

Definition 5: $\text{F}(\ )$. We will suppose a continuous function $\text{F}(\mathbb{R}^+) \to \mathbb{R}^+$ that approximates $bestToDate[\ ]$. It will be generated by an algorithm $\text{A}(\ )$, using the already known value for $bestToDate[\ ]$. Formally, $\text{A}(\ bestToDate[0...n-1]\ ) \to (\text{F}(\mathbb{R}^+) \to \mathbb{R}^+)$.

Our principal concern is not to find a function which explains well the values in the known range, but rather to extrapolate a new one outside the known range. Generally, extrapolation represents an additional challenge in comparison to interpolation (generation of new points inside the range) [14]. However, considering our definition for $bestToDate[\ ]$, algorithm $\text{A}(\ )$ can limit itself considering only monotonic functions. For example, if one puts forward the hypothesis that $bestToDate[\ ]$ decreases from its initial value $(\text{ArcValue}(0))$ by a constant rate β until it reaches an inferior limit α, algorithm $\text{A}(\ )$ must then

produce a function with the following form: $\mathrm{F}(i) := (\mathrm{ArcValue}(0) - \alpha)\, e^{-\beta i} + \alpha$. Figure 8 illustrates this situation. The role of algorithm $\mathrm{A}(\ )$ is then to estimate the value of parameters α and β for a given node. This could be done, for example, by achieving a least squares curve fitting with the *Gauss-Newton Algorithm* or the *Levenberg-Marquardt Algorithm* [14,15].

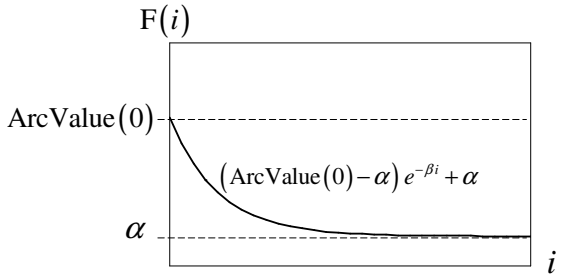


Figure 8. A first model for function $\mathrm{F}(i)$

Definition 6 : *improvement*$[\ ]$. Let us consider a node and two of its consecutive arcs, $i-1$ and i. They respectively lead to leaves having quality $arcValue[i-1]$ and $arcValue[i]$. The value of $improvement[i]$ corresponds to the improvement effected at the level of $bestToDate[\ ]$:

$$improvement[i] = bestToDate[i-1] - bestToDate[i]$$

Definition 7 : $\mathrm{Improvement}(\ )$. In a similar way, we will define the function $\mathrm{Improvement}(i)$ as being the expected improvement associated to the generation of an *i*-th local solution for the subproblem associated to current node, according to our approximation $\mathrm{F}(\ )$ for $bestToDate[\ ]$:

$$\mathrm{Improvement}(i) := F(i-1) - F(i)$$

For a given node *node* for which *n* local solutions have been generated, we are particularly interested in the value $\mathrm{Improvement}(n)$. This value is the extrapolation we seek, indicating whether an additional local solution (arc) is believed to lead to a global solution $(\mathrm{NextLeaf}(n))$ better than the one associated to the previous arcs. For simplification, we will write $\mathrm{Improvement}(\ )$ in place of $\mathrm{Improvement}(n)$ in the rest of the text.

3.3 Backtracking Strategy Exploiting the Model

The strategy we propose is the following. As in SyncBB and SyncLDS, the first global solution is obtained by solving the subproblems sequentially. For the first few backtracks, we select a node using an LDS policy. When we have backtracked at least once for each level of the tree, we can start using the model of the previous section in order to choose the node to backtrack to. The nodes will be compared according to their $\mathrm{Improvement}(\ )$ value[2].

By definition, $\mathrm{Improvement}(\ )$ can only be computed for nodes for which at least two arcs have been explored. Consequently, only the nodes meeting this condition are considered. We also suppose there is a threshold value ε beyond which the improvement is significant in the application domain. Besides this value, we suppose it is preferable to explore other nodes. In practice, we can use ε=0. When not enough nodes qualify according to the previous conditions (less than 2), we apply an LDS policy to select a node.

[2] We use $\mathrm{Improvement}(\ )$ rather than $\mathrm{F}(\ )$ as the weight criterion for the following reason. Using $\mathrm{F}(\ )$, a node having generated a good solution in the past would thence be preferred even if it started generating bad solutions.

Doing so increases the number of nodes which will qualify in the future, and represents an opportunity to discover new promising nodes. No node (even if its expected improvement is below ε) is discarded forever. The exploration of its remaining arcs is postponed until no other node seems more interesting. The method is complete (exploring the same search space as SyncBB or SyncLDS) but aims at producing good solutions in a short amount of time.

Each time a new global solution is found, the model must be updated. This dynamically modifies the priority given to the nodes, and by doing so, brings about the adaptive character of the strategy. Section 3.3.1 presents the pseudocode performing this update. Section 3.3.2 presents the pseudocode of a node selector applying the proposed strategy.

3.3.1 Updating the Model

Let us suppose that a global solution (leaf) has just been obtained. One must then update the vector $bestToDate[\ ]$ and the function $F(\ )$ for certain ancestors of this leaf. Figure 9 presents the pseudocode achieving this update. The function `UpdateModel` receives as arguments the quality of the leaf (`score`) and its unique identifier `p[]`. It is a vector of integers representing the path that would lead to the leaf in the corresponding global tree. The element `p[j]` defines, for a level `j`, which arc should be followed when going from the root to that leaf. The function `Card` returns the length of the vector. Finally, we recall that `nodeList` contains the nodes available for backtracking. Each node is defined by a tuple `<p[], bestToDate[], F()>`.

The main loop navigates along the path `p[]`, from the leaf to the root. For the leaf's parent node, we always update $bestToDate[\ ]$ and $F(\ )$. As for the upper nodes along the path, the update is not always necessary. Consider a node *node* and its *i*-th child *child* such as they are both on the path leading to *leaf*. If *leaf* is not the first leaf in the subtree rooted by *child*, then $leaf \neq node.\text{NextLeaf}(i)$ and this leaf's score is of no use in the computation of *node*'s $\text{ArcValue}(i)$ and $bestToDate[i]$ (according to Definitions 3 and 4). In this case, the nodes on the path that are above *node* do not need to be updated.

```
Procedure UpdateModel(p[], score)
  do
  {
     i := p[Card(p)-1];
     remove last element from p; // p is now the path of the parent
     node := select node in nodeList : (node.p = p); // node is that parent
     if (i = 0) node.bestToDate[0] := score;
     else node.bestToDate[i] := Min(node.bestToDate[i-1], score);
     node.F := A(node.bestToDate);
  }
  while ((Card(p) > 0) and (i = 0))
```

Figure 9. Updating $bestToDate[\ ]$ and $F(\ )$ when a global solution is found

3.3.2 Node Selector Implementing the Strategy

This section describes a node selector implementing the strategy. Figure 10 presents the pseudocode (see `SelectNode`). Since the model is unusable as long as a minimum of information has not yet been accumulated, the first backtracks are produced by virtue of an LDS policy. This is done by giving priority to the nodes for which the next global solution created will have a total number of discrepancies inferior to or equal to 1. We also prefer the LDS policy if too few nodes meet the ADS selection criteria (enforced by `FilterADS`). Nodes for which n is equal to zero also have priority as this corresponds to normal descent of the tree (when no backtracking is required). The function `CompareLDS` allows applying the LDS policy. It compares two nodes and returns the one of highest priority. The arguments of this function are: the path in the global tree of the next local solution each node would generate (thus, the concatenation of `p[]` and `n` as a new vector). In the case of equality, the equivalent of a chronological backtrack is applied to separate between the nodes (`CompareBT`).

```
Function SelectNode(nodeList)
  nodeListADS:= FilterADS(nodeList)
  if (Card(nodeListADS) < 2)
  or (∃ node in nodeList) : ((SumOfDisc(node)+node.n ≤ 1) or (node.n = 0))
     candidate := select node in nodeList according to function CompareLDS()
  else
     candidate := select node in nodeListADS : node.Improvement() is maximal
  return candidate

Function FilterADS(nodeList)
  return all node in nodeList : (node.n >= 2) and (node.Improvement() > ε)

Function SumOfDisc(node)
  return Σ(j=0..Card(node.p[])-1) node.p[j]

Function CompareLDS(p1, p2)
  t1 := Σ(j=0..Card(p1)-1) p1[j]
  t2 := Σ(j=0..Card(p2)-1) p2[j]
  if (t1 < t2) return p1
  else if (t2 < t1) return p2
  else return CompareBT(p1, p2)

Function CompareBT(p1, p2)
  depth := Min(Card(p1), Card(p2))
  j := 0
  while (p1[j] = p2[j] and j < depth) j := j+1
  if (j < depth)
       if (p1[j] ≤ p2[j]) return p1 else return p2
  else
       if (Card(p1) ≥ Card(p2)) return p1 else return p2
```

Figure 10. Node selector implementing the ADS strategy

3.4 Synchronous Adaptive Discrepancy-based Search (SyncADS)

This section introduces a protocol (SyncADS) allowing agents to perform distributed search while applying the previous adaptive backtracking strategy (ADS). Strictly speaking, the global tree exists nowhere, but the global solutions will be visited in the same order as if one was carrying out centralized search in the equivalent tree. As in SyncBB and SyncLDS, only one agent at a time is active. The transition from one agent to the other takes place by the exchange of messages that could be seen as the transmission of a privilege (or token). The term *synchronous* refers to the fact that an agent cannot select/change the solution for his local problem asynchronously (that is at any moment).

Each agent manages a list of nodes/subproblems under its authority (`nodeList`) and executes the pseudocode in Figure 11. The main procedure (`MsgProposition`) is activated when the agent receives a proposition from the previous one (or from the external customer). This proposition is denoted by a couple `<d,p[]>`. The element `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. Upon receiving this message, the agent creates a node corresponding to the new subproblem to solve and then adds it into `nodeList`. It then begins solving this instance of the subproblem (`Work`), finds a first solution and sends it to the next agent as a proposition (`send MsgProposition`). If there is no following agent, then we have on hand a solution for the global problem. The agent then updates its model (`UpdateBestToDate` and `F()`), informs its predecessors about the new solution quality (the message `MsgGlobalSolQuality` is propagated upward) and starts the cooperative backtracking mechanism (`CooperativeBacktracking`).

In function `CooperativeBacktracking`, each agent is asked to identify which node under its authority would be locally chosen (each agent selects it using the `SelectNode` function in Figure 10). Agents are also asked to count how many nodes qualify according to the ADS filtering criteria. Knowing that information, the calling agent can identify the node/subproblem with highest priority (using code similar to Figure 10 `SelectNode`). It then gives control to the agent responsible for that subproblem (`MsgBacktrack`). Please note that agents do not really send nodes as we do in pseudocode. They only need to communicate the vector `p[]` and the value `n`.

```
WhenReceive MsgProposition(<d,p[]>) do
  nodeList.add(<d, p[], n=0>)
  Work(node)

Procedure Work(node)
  proposition := NextSolution(node);
  if (proposition ≠ ∅)
     node.n := node.n+1
     if (Successor(node) ≠ ∅)
        send MsgProposition(<proposition, node.p[]+[node.n-1]>) to Successor(node)
     else
        UpdateBestToDate(node, node.n-1, proposition.score);
        node.F := A(node.bestToDate[])
        if (node.n = 1) send MsgQuality(score, node.p[]+node.n-1) to Predecessor(node)
        CooperativeBacktracking()
  else
     nodeList.remove(node)
     CooperativeBacktracking()

Procedure UpdateBestToDate(node, i, score)
  if (i = 0) node.bestToDate[0] := score
  else node.bestToDate[i] := Min(node.bestToDate[i-1], score)

WhenReceive MsgQuality(score, p[]) do
  node := select node in nodeList : p[] begins with node.p[]
  i := p[Card(node.p)-1]
  UpdateBestToDate(node, i, score)
  if (i = 0) and (Predecessor(node) ≠ ∅)
       send MsgQuality(score, p[]) to Predecessor(node)

Procedure CooperativeBacktracking()
  send MsgAskNbADSQualifiedNodes to Everybody // including itself
  nbQualifiedNodesADS := Σ answer from Everybody
  send MsgAskBestLocalNode() to EveryBody
  answers := all answer from EveryBody : (answer ≠ ∅)
  if (nbQualifiedNodesADS < 2)
  or (∃ node in answers) : ((SumOfDisc(node)+node.n ≤ 1) or (node.n = 0))
     candidate := select node in answers[] according to function CompareLDS()
  else
     candidate := select node in answers[] : node.Improvement() is maximal
  send MsgBacktrackADS(node) to Agent(answer)

WhenReceive MsgAskNbADSQualifiedNodes() do return Card(FilterADS(nodeList))

WhenReceive MsgAskBestLocalNode() do return SelectNode(nodeList)

WhenReceive MsgBacktrack(node) do Work(node)
```

Figure 11. Pseudocode for SyncADS

4 Evaluation

We will first apply the proposed approach to a real industrial supply chain problem in the forest products industry. We will evaluate the gains associated with the use of the proposed strategy, as well as the quality of the predictive model. Then we will evaluate the algorithm for synthetic problems, in order to generalize the obtained results.

4.1 Industrial Evaluation

We will use the same case and the same data as in [3]. It is a real coordination problem in an industrial supply chain producing softwood lumber. Figure 12 introduces the production units involved: (1) the sawmilling facility, where logs are cut into various sizes of rough pieces of lumber; (2) the drying facility, which reduces moisture level of the lumbers and (3) the finishing facility, where lumber is planed (surfaced), trimmed and sorted. We wish to synchronize activities between facilities (e.g. what to do, where and when). The objective is to minimize tardiness for delivery to the external customer.

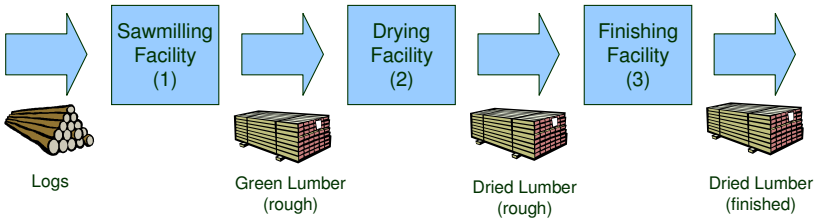


Figure 12. Lumber production supply chain.

Each facility uses a specialized local solver to plan its operations. The local solvers were previously developed by FORAC, a consortium of companies and researchers. The solvers have now been commercialized by the company *Synaptas*. In the base case, the facilities use *two-phase planning* as the coordination mechanism (see Section 2.1 and Figure 2). As the local solvers can produce alternative local solutions, this situation allows the use of distributed search as described in Section 2.2.

Sawmilling produces multiple types of lumber at the same time (co-production) from a single product type in input (divergence). Sawing operations are planned using a Mixed Integer Linear Programming model. It is designed to identify the right mix of log types and cutting patterns to use during each shift in order to control the output of the overall divergent production process. Many setup configurations are possible for the plant during each shift. Each configuration limits the log types and cutting patterns that can be used.

Wood drying operations planning is batch-oriented and aims at finding the type of rough lumber to put in the kiln dryers and the drying processes to implement. The problem is described thoroughly in [16]. Constraint programming [17] and *Depth-bounded Discrepancy Search* (DDS) [18] are used in order to produce alternative local solutions.

Finally, wood finishing operations planning aims at finding what dry lumber type and how much of it should be finished, taking into account setup time and sorting booth constraints. It is a divergent transformation process with co-production. It also uses constraint programming and DDS.

More information about this supply chain problem can be found in [6].

4.1.1 Methodology

Synchronous Branch and Bound (SyncBB) and SyncLDS were compared for the previous problem in [3]. Even for a large computation time, SyncBB persisted in exploring only minor variations of the first solutions (see Section 2.4.2). In contrast, SyncLDS allows better sampling of the solution space as LDS rapidly explores different areas of the tree. In consequence, the quality of the solutions found by SyncBB stops improving after a short computation time, while continuing to improve with SyncLDS - until it finally reaches a plateau within an hour's time. We will make the assumption that the part of the tree explored during this period of time is a good sample of the solution space. Here, our algorithm (SyncADS) will be compared to SyncLDS using the same trees.

We will also evaluate different predictive models $\mathrm{F}(\ )$ to be used for the extrapolation of $bestToDate[\ ]$. First, the model in Section 3.2 based on parameters α and β (hereafter model 1). We recall it makes the assumption that $bestToDate[\ ]$ decreases according to a constant rate β until it reaches a plateau α. We then have $\mathrm{F}(i) := (\mathrm{ArcValue}(0) - \alpha)\, e^{-\beta i} + \alpha$. Second, a simplified version where it is supposed that $bestToDate[\ ]$ decreases until reaching zero: $\mathrm{F}(i) := \mathrm{ArcValue}(0)\, e^{-\beta i}$ (model 2). Other models have been considered (e.g. second-degree polynomial) but preliminary testing showed their predictive capacity was rather poor although interpolation capacity was good.

The advantage of model 2 over model 1 (if it happens to allow obtaining good results) is that there is only one parameter to fit, and this can be done using simple linear regression. However, we used the *Levenberg-*

Marquardt Algorithm (LMA) in order to update all our models[3]. This standard non-linear least-squares minimization method [14] has the following advantage over the *Gauss-Newton Algorithm*. Although for both algorithms we need to provide initialization values for the parameters, LMA is less dependent on the quality of the initialization values. In our experiments, we used the following initial values: $\beta = 0.5$ and $\alpha = 0$.

4.1.2 Results

Table 1 compares time needed to get the best solution. It shows the reduction of computation time (%) achieved when using SyncADS instead of SyncLDS for the four industrial cases from [3]. Model 1 allowed an average reduction of **48.3%**, in comparison with **44.5%** for model 2.

Table 1. Computation time needed to get best solution – Reduction (%) made possible by SyncADS (vs SyncLDS)

Model	*Case #1*	*Case #2*	*Case #3*	*Case #4*	***Average***
1. $F(i) := (\text{ArcValue}(0) - \alpha)\, e^{-\beta i} + \alpha$	47.6 %	36.2 %	55.2 %	54.2 %	48.3 %
2. $F(i) := \text{ArcValue}(0)\, e^{-\beta i}$	42.9 %	30.9 %	50.2 %	54.2 %	44.5 %

Table 2 presents the average reduction of computation time needed to get solutions of intermediate quality. This indicator was proposed to verify that SyncADS allows obtaining solutions of intermediate quality (not just the best one) for a computation time equal or less than SyncLDS[4]. The reduction is smaller than for the previous indicator. Indeed, the more one is ready to accept poor solutions, the more the advantage of SyncADS over SyncLDS diminishes. For both algorithms it takes less time to find poor solutions than good solutions; SyncADS then has less time to learn and to distinguish itself over SyncLDS. We even have a result (case #2 with model 1) for which SyncADS is on average **7.7%** slower than SyncLDS for intermediate solutions, while being **36.2%** quicker to get the best solution. We will study this relationship between the performance of ADS and computation time further in Section 4.2.

Table 2. Average reduction (%) of the computation time needed to get solutions equal or better than SyncLDS

Model	*Case #1*	*Case #2*	*Case #3*	*Case #4*	***Average***
1. $F(i) := (\text{ArcValue}(0) - \alpha)\, e^{-\beta i} + \alpha$	28.7 %	-7.7 %	46.6 %	44.3 %	28.0 %
2. $F(i) := \text{ArcValue}(0)\, e^{-\beta i}$	24.0 %	12.7 %	42.9 %	43.6 %	30.8 %

4.1.3 Quality of the Models

We have seen in the previous section that SyncADS allowed better performance than SyncLDS for the industrial problem. In our opinion, two elements explain this result. First (hypothesis 1), the models proposed for $F(\ )$ model well $bestToDate[\ ]$ and allow good extrapolation of further values. Second (hypothesis 2), the curve/profile for $bestToDate[\ ]$ is relatively different from one node to another (i.e. there is some subproblems instances for which it is more worthwhile to generate alternative local solutions). Indeed, if this curve had been identical for all the nodes, it would be futile to choose the backtracking candidate on the basis of this; an LDS strategy would give equivalent results.

In order to verify hypothesis 1, we measured the gap between the forecasts of the models (i.e. the values returned each time $F(i)$ is called) and the actual values of $bestToDate[i]$ known *a posteriori*. We measured an average error of **1.6%** for model 1 and **3.6%** for model 2. The model with the best extrapolation is also the one that gave the best results in Section 4.1.2.

[3] We use the following implementation: *Levenberg-Marquardt.NET*, by Kris Kniaz. See http://kniaz.net

[4] This metric is calculated in the following way. Let us consider the execution of SyncLDS. The quality of the "best solution to the global problem found up to now" evolved over time. For any level of quality reached by SyncLDS, we evaluate the time necessary to SyncADS to reach a solution that is equal or better.

To verify hypothesis 2, we proceeded as follows. For each node we characterized its vector $bestToDate[\ ]$ using a single value β. We took the values in $bestToDate[\ ]$ (they are all known after the search) and best fit them. Figure 13 shows the distribution for β in the industrial data. The trend line shows the distribution is close to be exponential $\left(e^{-12.718x}\right)$, defined for the interval $[0,...,0.5]$. Let us recall that nodes for which β is close to zero corresponds to subproblems instances for which it has been unprofitable to generate alternative solutions (these are therefore very numerous in our industrial case). Nodes with a bigger β have more potential. The strategy seeks to dwell on these in priority.

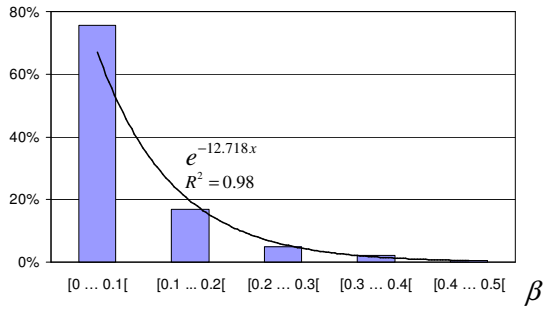


Figure 13. Distribution of the values β for the nodes of the studied trees

In a similar way, for each node we observed the last value in the vector $bestToDate[\ ]$ in order to estimate α. For our node population, values for α are distributed more or less uniformly between $\text{ArcValue}(0)$ and $0.5\times\text{ArcValue}(0)$.

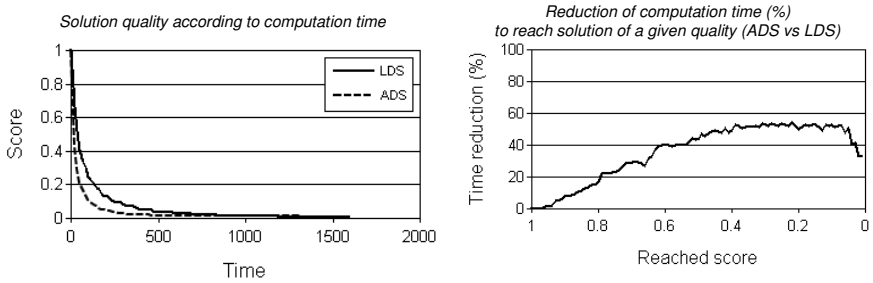
4.2 Evaluation with Synthetic Data

In the trees corresponding to our industrial coordination problems, there are some node/subproblems for which it is worth more to produce alternative solutions (these are the nodes for which we measured large β). The performance of our approach is strongly related to the distribution of the values for β. Here, we will generate new datasets that will be more or less favorable to our algorithm, in order to study some of its characteristics. We will also study the impact of the distribution for δ, and the impact of the number of subproblems (that is, the depth of the tree).

We will suppose minimization problems such as the first leaf of the tree corresponds to a solution with quality equal to 1. For each node, we randomly choose a value for parameter β using the probability distribution $\text{PR}(\beta = x) \coloneqq e^{-\gamma x}$. For our industrial cases we measured $\gamma = -12.718$ but we will try other values. In a similar way, the values for α will be randomly chosen using a uniform distribution defined between 0 and $\delta\times\text{arcValue}[0]$. Knowing values β and α for each node, we can calculate the quality of other leaves using model 1. The trees are generated dynamically during search. The total number of nodes will depend on the computation time allowed to the search.

To introduce our evaluation framework, we will first study the following case. We have trees corresponding to a hierarchy of 4 subproblems types, generated using $\gamma = 10$ (considering $0 \le \beta \le 0.5$) and $\delta = 0$. This case was chosen because it allowed producing solutions really close to 0 within reasonable computation time. Figure 14i shows the quality of the best solution found so far, according to computation time (measured as the number of visited nodes) for SyncLDS and SyncADS. We can see that for large computation time, both allow very good solutions. Subfigure (ii) presents the reduction in computation time allowed by SyncADS (in comparison with SyncLDS) for equal solution quality. As an example, obtaining a solution with a score equal to 0.4 (i.e. a reduction of the objective function of **60%**) takes approximately half the time using SyncADS. We can also point out that that the relative advantage of SyncADS decreases

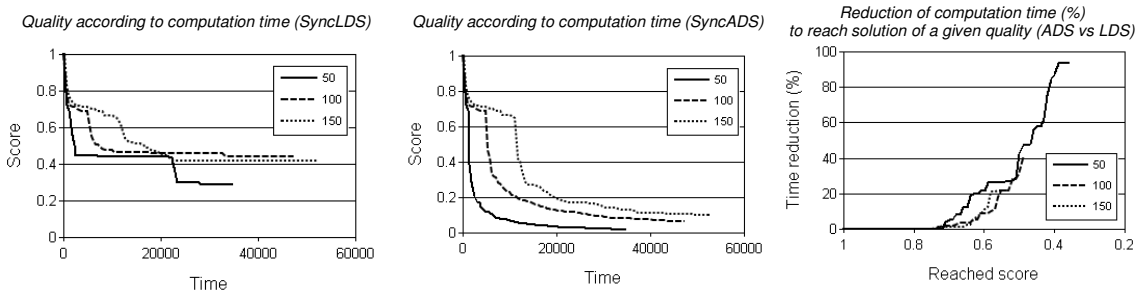
for scores very close to 0. For both methods the score tends to 0 for large computation time (subfigure i) and the relative advantage tends to diminish.



(i) (ii)

Figure 14. Comparison of SyncLDS and SyncADS for trees of depth=4, with [$\gamma = 10$; $0 \le \beta \le .5$; $\delta = 0$]

The next experiment (Figure 15) shows how the number of subproblems (that is, the depth of the tree) affects the performance. Subfigures (i) and (ii) shows solution quality according to computation time for SyncLDS and SyncADS (for depth=50, 100, 150). Subfigure (iii) shows the reduction in computation time permitted by ADS. Two details receive our attention. With ADS, the quality of the solutions improves more gradually and continuously according to computation time. However, at the beginning of the search the results for ADS are identical to those for LDS. This corresponds to the initialization phase of our algorithm, where backtracking is performed like LDS. We can see (subfigure ii) that we quickly reach a plateau (especially noticeable for depth=150). The end of this plateau corresponds to the end of the initialization phase.

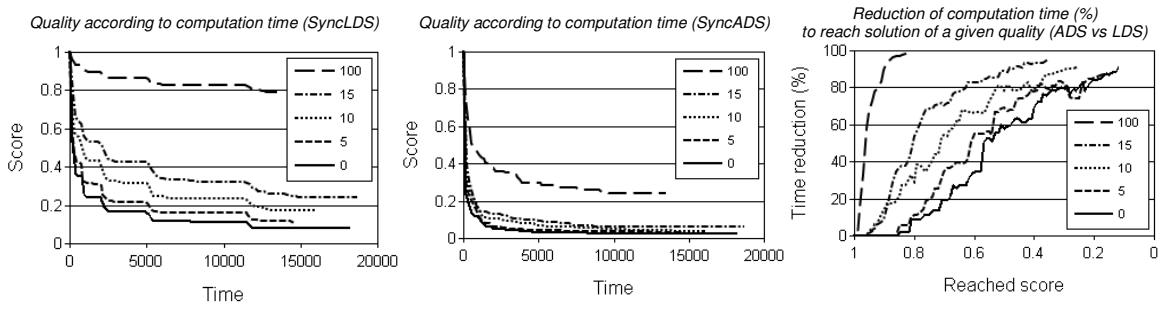


(i) (ii) (iii)

Figure 15. Impact of the number of subproblems (depth=50, 100, 150), with [$\gamma = 10$; $0 \le \beta \le .5$; $\delta = 0$]

The results shown in Figure 16 demonstrate the impact of parameter γ. Once again, we have assumed the values of β to be between 0 and 0.5 according to an exponential distribution $e^{-\gamma x}$. With $\gamma = 0$, the values β are chosen from a uniform distribution. The higher γ is, the fewer nodes there are with high β. Stated otherwise, the greater γ is, the fewer nodes there are for which it is profitable to produce a great number of discrepancies. It should thus be more difficult to find good solutions (hypothesis 1) and the search for and detection of 'profitable' nodes should be worthwhile (hypothesis 2). The subfigures (i) and (ii) confirm hypothesis 1; for a same strategy, the quality curves are less and less good as γ grows. Subfigure (iii) confirms hypothesis 2; the greater γ is, the more the ADS strategy has a significant advantage over the LDS strategy. These results empirically illustrate the following intuitive idea: the rarer the 'promising' nodes are, the more searching for and remaining with them is worthwhile. Even for $\gamma = 0$, we can observe an advantage of ADS over LDS. This is because there is still variability in the tree (and thus some nodes are more interesting than others) even if the β values are taken from a uniform distribution[5].

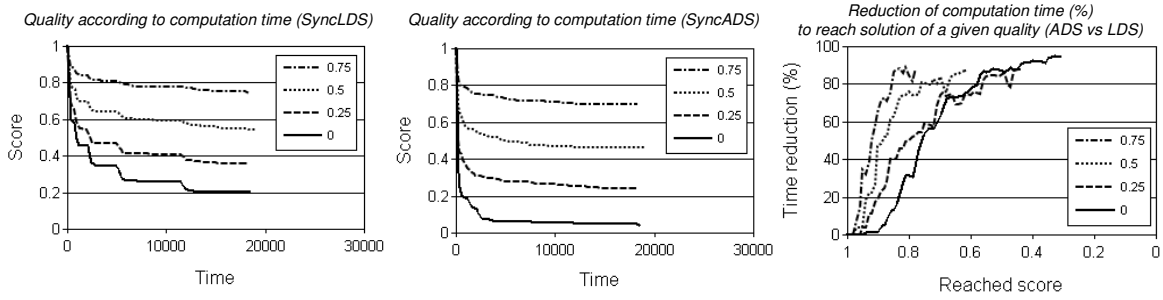
[5] If we use the same value β for any node in the tree, then ADS reports the same results as LDS (not shown on the chart)



(i) (ii) (iii)

Figure 16. Impact of parameter γ (0, 1, 10, 15, 100), with [$0 \le \beta \le .5$; depth=10; $\delta = 0$]

Finally, the last experiment illustrates the impact of parameter δ. Let us recall that the α values of the nodes (the value towards which $bestToDate[i]$ tends for the high i) are chosen between $arcValue[0]$ and $\delta \times arcValue[0]$ according to a uniform distribution. When we have $\delta = 0$, then for every node the value $\text{Next Leaf}(i)$ tends toward 0 for a high i. The higher δ is, the more variability there is in the tree and the more the learning becomes profitable. This is illustrated by the results in Figure 17.



(i) (ii) (iii)

Figure 17. Impact of parameter δ (0, 0.25, 0.50, 0.75), with [$\gamma = 10$; $0 \le \beta \le .5$; depth=10]

5 Related Work

This section situates our approach in relation to others that use related learning techniques in the context of classical global combinatorial problems. But first, let us recall the major difference between our context and the one of combinatorial optimization (other than the fact that our problem is a distributed one). For classical combinatorial problems, it is the solving algorithm that constructs the tree (by choosing the order in which variables are instantiated and the order in which values are tried). Learning can then be used in order to establish a strategy for variable ordering and value ordering. In the context of hierarchical decision making, the sequence of sub-problems (equivalent to the sequence of variables) is determined in advance and the sequence of local-solutions (equivalent to value ordering) depends on local criteria of each decision maker (see problem definition in Section 2.2). Our room for manoeuvre is limited to the backtracking strategy. However it is relevant to establish a parallel between these approaches.

Firstly, we distinguish between two major currents: (1) the *training* approach, and (2) the *adaptive* approach. The first approach (1) consists in training a system for the resolution of a particular family of problems. In its basic form, the system achieves what a practitioner would do manually, that is to configure a solver for a particular context [19]. For example, using a set of training problems, the system could determine which variable ordering heuristics and value ordering heuristics are best for a given family of problems. The ACE system [20] uses a similar but more advanced approach. After the training phase, it attributes weights to different heuristics according to their pertinence. Once in production, the system will have the different heuristics vote for the next variable to instantiate, taking their weight into account. The performance obtained can be better than that of individual heuristics. Other approaches can be used. As an

example, the system proposed in [21] studies a set of trees in order to identify the cuts that can be carried out on all these trees while still being assured that good solutions can be found. After training, these cuts are applied to the new problems submitted to the system.

The *adaptive* approach (2) concerns the development of systems that dynamically react and adjust during the resolution of a particular instance of a problem [22]. This approach is often put to good use for *constraint satisfaction problems* (CSP). Many apply what is called *Learning from failure*. When a constraint is violated during the descent of the tree, the conditions of that failure are analyzed with the view of making the most of this knowledge throughout the remainder of the search. For example the techniques of *nogood recording* and *clause learning* seek to avoid redoing combinations of variable/value affectations that are mutually inconsistent. Others try to learn during the search which variables are the most difficult to instantiate, in order to change dynamically the order of variables (e.g. *YIELDS* [23]). In *Impact Based Search* (IBS), the impact of variables is measured by observing how their instantiation reduces the size of the search space [24]. In [25] and [26], each time a constraint causes a failure, the priority of variables implicated in this constraint is increased. In [27], another approach for variable ordering is proposed, but for the context of *Weighted CSP*.

Regarding backtracking strategy, approaches where the system learns to evaluate the quality of nodes are of particular interest for us. Ruml has made an interesting proposal regarding this. While a basic LDS strategy gives the same importance to any discrepancy, BLFS [28] dynamically attributes different weights to discrepancies according to their depth. For a binary tree, BLFS will define two parameters for each level of the tree. One corresponds to the "cost" of branching to the left, and the other to the right. The value of a leaf is reckoned to be equal to the sum of the costs along the path from the root to this leaf. By knowing the value of a certain number of leafs, BLFS uses a linear regression in order to establish the value of the parameters. The model is not used in order to define a backtracking strategy. Instead, the algorithm proceeds to a series of successive descents in the tree. At each run, it tries to reach a leaf using a path that minimizes the costs. The branching choices are made stochastically in order to avoid always taking the same path. Ruml has achieved very good results with this algorithm (see [29]). The limits of this approach are the following. The branching factor must be the same for each node on the same level, and one can say nothing about the cost of a supplementary discrepancy at a given node as long as at least as much has been done at another node on the same level. Moreover, the impact of performing an *i*-th discrepancy at a given node is supposed to be the same for all other nodes on the same level. And this hypothesis was not met for our industrial problem.

6 Conclusion

We proposed an adaptive search strategy for efficient distributed decision making in hierarchical contexts (ADS). Agents collectively and dynamically identify the most promising areas of the tree in order to explore them first. It allows agents to systematically search the solution space (thus looking for the optimal solution) but aims at producing good solutions in a short amount of time.

We applied the method to a real industrial coordination problem in the Canadian forest industry. It reduced computation time needed to get the best solution by nearly half. As well, we have evaluated ADS for synthetic problems. It allowed evaluating the performance of the algorithm for a wide range of problems, according to how difficult it is to find nodes/subproblems leading to good global solutions. It also allowed evaluating the performance for coordination situation with more agents.

As future work, we plan to do a comparative evaluation of LDS and ADS strategies in a context where agents work concurrently (rather than sequentially). The algorithm called *Multi-agent Concurrent Discrepancy Search* (MacDS) [3] allows applying an LDS strategy in a concurrent context (each global solution is produced sequentially, but the agents work simultaneously on many solutions). It could be adapted in order to apply the ADS strategy. Such an algorithm would search in several areas of the tree at the same time, which would further improve the capacity of ADS to identify promising nodes.

References

[1] C. Schneeweiss. *Distributed Decision Making*, New York: Springer, 2003.

[2] R. Bhatnagar, P. Chandra and S. K. Goyal. "Models for multi-plant coordination", *European Journal of Operational Research*, vol. 67, 1993, pp. 141-160.

[3] J. Gaudreault, J. M. Frayret and G. Pesant. "Discrepancy-based Method for Hierarchical Distributed Optimization", *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, 2007.

[4] C. Kilger and B. Reuter. "Collaborative Planning" in: *Supply Chain Management and Advanced Planning*, H. Stadtler and C. Kilger Eds. New York: Springer, 2005, pp. 259-278.

[5] A. Fink. " Supply Chain Coordination by Means of Automated Negotiations Between Autonomous Agents" in: *Multiagent-Based Supply Chain Management*, B. Chaib-draa and J. P. Müller Eds. New York: Springer, 2006. pp. 450

[6] J. M. Frayret, S. D'Amours, A. Rousseau, S. Harvey and J. Gaudreault. "Agent-based Supply Chain Planning in the Forest Products Industry", *International Journal of Flexible Manufacturing Systems*, vol. 19, 2007.

[7] G. Dudek and H. Stadtler. "Negotiation-based collaborative planning between supply chains partners", *European Journal of Operational Research*, vol. 163, 2005, pp. 668-687.

[8] C. Schneeweiss and K. Zimmer. "Hierarchical coordination mechanisms within the supply chain", *European Journal of Operational Research*, vol. 153, 2004, pp. 687-703.

[9] K. Hirayama and M. Yokoo. "Distributed partial constraint satisfaction problem", *International Conference on Principles and Practice of Constraint Programming, LNCS #1330*, 1997, pp. 222-236.

[10] W. D. Harvey and M. L. Ginsberg. "Limited discrepancy search", *International Joint Conference on Artificial Intelligence*, 1995, pp. 607-613.

[11] C. Le Pape and P. Baptiste. "Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem", *Journal of Heuristics*, vol. 5, 1999, pp. 305-325.

[12] W. D. Harvey. "Nonsystematic backtracking search". Ph.D. thesis, Stanford University, California, 1995.

[13] J. C. Beck and L. Perron. "Discrepancy-Bounded Depth First Search", *Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems*, 2000, pp. 7-17.

[14] W. H. Press. *Numerical recipes the art of scientific computing*, Cambridge: Cambridge University Press, 2007.

[15] D. W. Marquardt. "An algorithm for least-squares estimation of nonlinear parameters", *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, 1963, pp. 431-441.

[16] J. Gaudreault, J. M. Frayret, A. Rousseau, and S. D'Amours. "Combined planning and scheduling in a divergent production system with co production", Université Laval, CENTOR, DT-2006-JMF-1, 2006.

[17] I. J. Lustig and J. F. Puget. "Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming", *Interfaces*, vol. 31, 2001, pp. 29-53.

[18] T. Walsh. "Depth-bounded discrepancy search", *International Joint Conference on Artificial Intelligence*, 1997, pp. 1388-1393.

[19] F. Hutter, D. Babic, H. H. Hoos and A. J. Hu. "Boosting verification by automatic tuning of decision procedures", *Formal Methods in Computer Aided Design*, 2007, pp. 27-34.

[20] S. L. Epstein, E. C. Freuder and R. J. Wallace. "Learning to support constraint programmers", *Computational Intelligence*, vol. 21, 2005, pp. 336-371.

[21] E. Breimer, M. Goldberg, D. Hollinger and D. Lim. "Discovering optimization algorithms through automated learning", *Graphs and Discovery. DIMACS Working Group Computer-Generated Conjectures from Graph Theoretic and Chemical Databases, 12-16 Nov. 2001*, 2005, pp. 7-25.

[22] R. Battiti, M. Brunato and F. Mascia. *Reactive Search and Intelligent Optimization*, Springer (In press), 2008.

[23] W. Karoui, M.-J. Huguet, P. Lopez and W. Naanaa. "YIELDS: a yet improved limited discrepancy search for CSPs", *Proceedings of the 4th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2007, pp. 99-111.

[24] P. Refalo. "Impact-based search strategies for constraint programming", *International Conference on Principles and Practice of Constraint Programming, LNCS #3258*, 2004, pp. 557-71.

[25] F. Boussemart, F. Hemery, C. Lecoutre and L. Sais. "Boosting systematic search by weighting constraints", *Proceedings of the 16th European Conference on Artificial Intelligence*, 2004, pp. 146-150.

[26] D. Grimes and R. J. Wallace. "Learning from failure in constraint satisfaction search", *2006 AAAI Workshop, Jul 16-20 2006*, 2006, pp. 7-14.

[27] N. Levasseur, P. Boizumault and S. Loudni. "A value ordering heuristic for weighted CSP", *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, 2007, pp. 259-62.

[28] W. Ruml. "Adaptive Tree Search". Ph.D. thesis, Harvard University, 2002.

[29] W. Ruml. "Heuristic Search in Bounded-depth Trees: Best-Leaf-First Search", *Working Notes of the AAAI-02 Workshop on Probabilistic Approaches in Search*, 2002.