A Decomposition-Based Implementation of Search Strategies

LAURENT MICHEL and PASCAL VAN HENTENRYCK
Brown University

Search strategies, that is, strategies that describe how to explore search trees, have raised much interest for constraint satisfaction in recent years. In particular, limited discrepancy search and its variations have been shown to achieve significant improvements in efficiency over depth-first search for some classes of applications.

This article reconsiders the implementation of discrepancy search, and of search strategies in general, for applications where the search procedure is dynamic, randomized, and/or generates global cuts (or nogoods) that apply to the remaining search. It illustrates that recomputation-based implementations of discrepancy search are not robust with respect to these extensions and require special care which may increase the memory requirements significantly and destroy the genericity of the implementation.

To remedy these limitations, the article proposes a novel implementation scheme based on problem decomposition, which combines the efficiency of the recomputation-based implementations with the robustness of traditional iterative implementations. Experimental results on job-shop scheduling problems illustrate the potential of this new implementation scheme, which, surprisingly, may significantly outperform recomputation-based schemes.

Categories and Subject Descriptors: D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Languages Classifications—*constraint and logic languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*constraints; control structures*

General Terms: Design, Languages

Additional Key Words and Phrases: Constraint programming, search, combinatorial optimization

1. INTRODUCTION

Combinatorial optimization problems are ubiquitous in many practical applications, including scheduling, resource allocation, software verification, and computational biology to name only a few. These problems are computationally difficult in general (i.e., they are NP-hard) and their solving requires considerable

This research was supported in part by National Science Foundation (NSF) Awards DMI-0121495 and ACI-0121497.
Authors' present addresses: L. Michel, University of Connecticut, Computer Science & Engineering, 191 Auditorium Road, Storrs, CT 06269; P. Van Hentenryck, Brown University, Department of Computer Science, Box 1910, Providence, RI 02912; email: pvh@cs.brown.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2004 ACM 1529-3785/04/0400-0351 $5.00

expertise in optimization and software engineering, as well as in the application domain.

The constraint satisfaction approach to combinatorial optimization, which emerged from research in artificial intelligence (e.g., Fikes [1968], Lauriere [1978], Mackworth [1977], Montanari [1974], and Waltz [1972]) and programming languages (e.g., Colmerauer [1990], Van Hentenryck [1989], and Jaffar and Lassez [1987]), consists of a tree search algorithm using constraints to prune the search space at every node. Solving a problem in this approach amounts to specifying a set of constraints describing the solution set and a search procedure indicating how to search for solutions. Constraint programming (e.g., Colmerauer [1990], Jaffar and Lassez [1987], and Van Hentenryck [1989]) aims at simplifying this task by supporting rich languages to specify constraints and search procedures.

The search procedure in constraint satisfaction implicitly describes the search tree to explore. It does not specify how to explore it, which is the role of search strategies. Traditionally, the search tree is explored using depth-first search. However, in recent years, novel search strategies (e.g., Harvey and Ginsberg [1995], Korf [1996], Meseguer [1997], and Walsh [1997]) have raised much interest. In particular, limited discrepancy search (LDS) and its variations have been shown to achieve significant improvements in efficiency over depth-first search for some classes of applications. The basic idea underlying discrepancy search is to explore the search space in waves organized according to a good heuristic. The first wave (wave 0) simply follows the heuristic. Wave i explores the solutions which can be reached by assuming that the heuristic made i mistakes.

Because of their success on a variety of practical problems (e.g., Harvey and Ginsberg [1995], Laburthe and Caseau [1998], and Perron [1999]), search strategies became an integral part of constraint programming languages in recent years (e.g., Laburthe and Caseau [1998], Perron [1999], Schulte [1997], and Van Hentenryck et al. [2000]). In particular, the Oz programming system pioneered the specification of *generic search strategies* that can be specified independently of search procedures [Schulte 1997]. As a consequence, a search strategy can be applied to many search procedures and a search procedure can be "explored" by several strategies. Oz uses *computation states* to implement generic search strategies. Computation states are, in general, demanding in memory requirements but they have many appealling uses (e.g., Schulte [2000]). The search language SALSA [Laburthe and Caseau 1998] also contains a number of predefined generic search strategies. More recently, it was shown in Perron [1999] that many search strategies can be specified by two functions only: an *evaluation* function which associates a value with every node in the search tree and a *suspension* function which indirectly specifies which node to expand on next. The implementation technology in Perron [1999] is of particular interest. When applied to LDS, it features a recomputation-based scheme instead of the traditional iterative exploration initially proposed in Harvey and Ginsberg [1995]. More precisely, the implementation is organized around a queue which stores nodes to explore and uses backtracking and recomputation to restore the node popped from the queue. Nodes in the queue are represented

by their computation paths and, possibly, some other information. This implementation may yield significant improvements in efficiency over the iterative implementation for some applications.

This article is motivated by the observation that an increasing number of applications use sophisticated, highly dynamic, search procedures which may involve randomization (e.g., Nuijten [1994]), global cuts or nogoods (e.g., Hooker [2000] and Wolsey [1998]), and semantic backtracking (e.g., Refalo and Van Hentenryck [1996]) to name only a few. Randomized search procedures typically do not generate the same search tree (or subtree) when executed several times. Global cuts and nogoods are constraints that are generated dynamically during search, are valid for the whole search tree, and are used to prune the remaining search space. Semantic backtracking techniques use the constraint semantics to return, upon backtracking, to a state which is semantically equivalent (i.e., it has the same set of solutions) but may differ in its actual form (e.g., a different linear programming basis). This trend of using advanced search procedures is likely to continue and to grow, especially with the integration of constraint and integer programming (e.g., El Sakkout and Wallace [2000], Hooker [2000], Hooker et al. [2001], Refalo [1999], and Van Hentenryck [1999]) which opens new opportunities for heuristics. Unfortunately, these dynamic techniques raise fundamental challenges for recomputation-based implementations of search strategies. Indeed, recomputation relies on the ability to execute the search procedure in the exact same computation states which is much harder to achieve in presence of these features. For instance, randomized search procedures cannot be used during recomputation, which must be deterministic. Global cuts and nogoods change the constraint store dynamically; hence dynamic heuristics, which are prominently used in combinatorial optimization, may behave differently during recomputation due to these additional constraints. Similarly, semantic backtracking may change the behavior of heuristics based on linear relaxations (e.g., El Sakkout and Wallace [2000]). As a consequence, naive implementations based on recomputation become incomplete or may miss good solutions, while correct implementations may significantly increase the memory requirements and destroy the genericity of the implementation.

To remedy these limitations, this article presents a novel, decomposition-based, implementation of LDS in particular, and of search strategies in general. The key idea of the implementation is to view the search procedure as (implicitly) specifying a problem decomposition scheme (as in Freuder and Hubbe [1995]) instead of a search tree. By adopting this more "semantic" viewpoint, the implementation avoids the pitfalls of recomputation. In particular, it combines the runtime efficiency of recomputation-based schemes while retaining the robustness of traditional iterative implementations. Experimental results on job-shop scheduling problems demonstrate the potential of the new scheme. Surprisingly, decomposition-based implementations may significantly outperform recomputation-based schemes on these benchmarks.

The rest of this article is organized as follows: Since one of the objectives of the article is to illustrate the advantages and inconvenients of various approaches, it is necessary to present the algorithms at some level of detail.

Section 2 introduces the abstractions and notations used throughout the article to meet this goal. It defines various concepts, for example, constraint stores, goals, configurations, and computation paths, and illustrates them on depth-first search. Section 3 briefly reviews LDS, the search strategy used to illustrate the implementations. Section 4 describes the traditional iterative implementation of LDS and Section 5 presents the recomputation-based scheme. Section 6 introduces our novel, decomposition-based, implementation and Section 7 proves its correctness. Section 8 discusses how to represent the queue in recomputation-based and decomposition-based schemes. Section 9 generalizes the decomposition-based algorithm to strategies specified by evaluation and suspension functions. Section 10 presents the experimental results and Section 11 concludes the article. Note also that the article contains the first descriptions of limited discrepancy search, and search strategies in general, in an optimization setting.

2. BACKGROUND

This section describes the main abstractions and notations used throughout the paper to simplify the presentation. It also presents the hypotheses and properties used in the algorithms. Our goal is to present the algorithms at a high level of abstraction while giving enough detail to capture their time and space requirements and to discuss their robustness. As a consequence, the presentation of the algorithms require some machinery that would not be necessary otherwise. On the one hand, recursion is not used to save and restore constraint stores, since this would hide the memory and time requirements of the algorithms and the robustness issues. On the other hand, search procedures are not assumed to be described by static search trees, since the dynamic nature of search procedures is the primary difficulty in designing efficient implementations of limited discrepancy search and of search strategies in general. Instead, the algorithms use a dynamic branching function which constructs the search tree dynamically.

Our notations and abstractions are essentially borrowed from constraint logic programming but are simpler because of the nature of constraint satisfaction problems. Section 2.1 describes constraint stores and their operations, Section 2.2 defines the concept of solutions and optimal solutions, Section 2.3 shows how search procedures are expressed and Section 2.4 specifies their required properties. Section 2.5 illustrates the notations and abstractions by showing how to implement depth-first search. Section 2.6 discusses computation paths that are fundamental in implementations of LDS.

2.1 Constraint Stores

A constraint store is a finite sequence $\langle c_1, \ldots, c_n \rangle$ of constraints (or conjunction of constraints). Constraint stores are denoted by the letter σ, possibly subscripted or superscripted. Constraints, or conjunctions of constraints, are denoted by the letter c, possibly subscripted or superscripted. Given a constraint store $\sigma = \langle c_1, \ldots, c_n \rangle$, we use $\sigma \wedge c$ to denote $c_1 \wedge \cdots \wedge c_n \wedge c$.

Our algorithms use five main operations on constraint stores which are specified in Figure 1. Function SUCCESS(σ) returns true if σ is a solution, that is, σ

```
SUCCESS(Store σ) : Bool
   post:  SUCCESS(σ) iff σ is consistent and all variables are instantiated.

FAILURE(Store σ) : Bool
   post:  FAILURE(σ) implies σ is inconsistent.

TELL(Store σ,Constraint c)
   pre:   σ = ⟨c1, ..., cn⟩.
   post:  σ = ⟨c1, ..., cn, c⟩.

BACKTRACK(Store σ)
   pre:   σ = ⟨c1, ..., cn, c⟩.
   post:  σ = ⟨c1, ..., cn⟩.

EVAL(Expr f,Store σ) : int
   pre:   SUCCESS(σ).
   post:  EVAL(f,σ) is the value of f in σ.

BRANCH(Goal g,Store σ) : (Goal × Store) × (Goal × Store)
   pre:   ¬SUCCESS(σ) ∧ ¬FAILURE(σ);
          g is complete for σ.
   post:  BRANCH(g,σ) = ⟨(gl, cl), (gr, cr)⟩;
          σ ⇔ (σ ∧ cl) ∨ (σ ∧ cr);
          cl ∧ cr is inconsistent;
          gl is complete for cl and gr is complete for cr;
          g is the parent of gl and gr.

PARENT(Goal g) : Goal
   pre:   g is not the initial goal.
   post:  PARENT(g) returns the parent of g.
```

Fig. 1. Specification of the main operations.

is consistent and assigns a value to each of its variables. It returns false otherwise. Function FAILURE(σ) returns true if σ can be shown to be inconsistent. It returns false otherwise.[1] Procedure TELL(σ, c) adds constraint c to store σ. Procedure BACKTRACK(σ) removes the last constraint added to σ. Finally, procedure EVAL(f, σ) returns the value of expression f in solution σ.

2.2 Solutions and Optimal Solutions

The purpose of the algorithms described in this article is to find one or all solutions to a constraint store or to find the solution of a constraint store that minimizes an objective function. The solutions of a constraint store σ are given by

$$\text{SOL}(\sigma) = \{\sigma' \mid \sigma' \Rightarrow \sigma \ \& \ \text{SUCCESS}(\sigma')\}.$$

[1]Observe that, as traditional, FAILURE(σ) may return false even though σ is inconsistent.

A constraint store σ is satisfiable, denoted by SATISFIABLE(σ), if its solution set is non-empty, that is,

$$\text{SATISFIABLE}(\sigma) \equiv \text{SOL}(\sigma) \neq \emptyset.$$

The minimum value of f in σ, denoted by MIN(f,σ), is the smallest value taken by f in any solution of σ, that is,

$$\text{MIN}(f, \sigma) = \min_{\sigma' \in \text{SOL}(\sigma)} \text{EVAL}(f, \sigma').$$

A minimal solution of f in σ is a solution of σ which minimizes f, that is, a solution σ' such that

$$\text{EVAL}(f, \sigma') = \text{MIN}(f, \sigma).$$

In the rest of the article, we only present algorithms to determine the satisfiability of a constraint store and to find the minimum value of a function in a constraint store. Satisfiability is useful to introduce the main ideas behind the algorithms and their properties. Minimization is particularly important since it illustrates, in a simple way, many issues arising when introducing global cuts and nogoods in dynamic search procedures.

2.3 Search Procedures

Search procedures are used to find solutions to constraint stores. As mentioned earlier, it is important to consider dynamic search procedures, since they raise the main difficulties in implementing search strategies efficiently and are widely used in practice. Indeed, standard labeling procedures generally choose the next variable to assign dynamically (e.g., using the first-fail principle [Haralick and Elliot 1980]). They may also order the sequence of values dynamically, once again using information from the constraint store. Similarly, in job-shop scheduling, the choice of the machine to schedule and the choice of the task to rank on the selected machine is generally dynamic. In addition, some search procedures are randomized (e.g., Nuijten [1994]).

To capture the dynamic and randomized nature of complex search procedures while retaining a high level of abstraction, we assume that search procedures are specified by goals as, for instance, in constraint logic programming. In addition, to simplify the presentation, we only use one branching operation BRANCH(g,σ), which rewrites the goal g into two subgoals, possibly using information from the constraint store and randomization. The specification of Function BRANCH is also given in Figure 1 together with the specification of PARENT which is used by some of the algorithms. Informally speaking, function BRANCH(g,σ) rewrites g into a disjunction

$$(g_l \wedge c_l) \vee (g_r \wedge c_r),$$

where c_l and c_r partitions the set of solutions of σ. The goals g_l and g_r are assumed to be complete search procedures for $\sigma \wedge c_l$ and $\sigma \wedge c_r$, respectively. *Observe that the order of the disjunctions is important for the purpose of this article as the branching function recommends to explore the left disjunct before the right one.*

Most reasonable search procedures can be expressed in terms of branching functions. In a traditional labeling procedure, c_l and c_r may be constraints of the form $x = v$ and $x \neq v$, where x and v are the selected variable and value respectively. In job-shop scheduling, c_l and c_r are constraints of the form $rank = r$ and $rank > r$, where $rank$ is the variable representing the rank of a given task on a machine and r represents, say, the first rank available on the machine in the current constraint store. It is possible to generalize this operation further (e.g., to have n-ary branches or more general operations instead of constraints c_l and c_r) but this is not necessary for the purpose of this article.

2.4 Properties of Search Procedures

This article restricts attention to complete search procedures, that is, goals that can find all solutions to a constraint store. It also assumes that goals always terminate, that is, there is no infinite sequence of branchings. Finally, this article also assumes that, if a goal is complete for a constraint store σ, it is also complete for any constraint store $\sigma \wedge c$, where c is a constraint over a subset of the variables of σ. These three assumptions are satisfied by all "reasonable" search procedures for constraint satisfaction problems. Together, they also imply that a goal g is complete for a constraint store σ' obtained after a number of branchings from g and an initial constraint store σ. The rest of this section formalizes these concepts.

2.4.1 *The Operational Semantics.* The formalization uses an operational semantics based on a transition system (SOS semantics) [Plotkin 1981]. The configurations of the transition system represent the computation states; they are of the form $\langle g, \sigma \rangle$ and σ. Configurations of the form σ are terminal and represent solutions (i.e., SUCCESS(σ) holds). Transitions represent computation steps and a transition $\gamma \longmapsto \gamma'$ can be read as "configuration γ nondeterministically reduces to γ'." Transition rules are defined using the format

$$\frac{\begin{array}{l} < \text{condition } 1 > \\ \ldots \\ < \text{condition } n > \end{array}}{\gamma \ \longmapsto \ \gamma'}$$

expressing the fact that a transition from γ to γ' can take place if the conditions are fulfilled. There are only three transition rules in the operational semantics. The rule

$$\frac{\text{SUCCESS}(\sigma)}{\langle g, \sigma \rangle \longmapsto \sigma}$$

expresses that the computation terminates as soon as the constraint store is a solution. The rules

$$\frac{\begin{array}{l} \neg\text{SUCCESS}(\sigma) \wedge \neg\text{FAILURE}(\sigma) \\ \text{BRANCH}(g, \sigma) = \langle (g_l, c_l), (g_r, c_r) \rangle; \end{array}}{\langle g, \sigma \rangle \longmapsto \langle g_l, \sigma \wedge c_l \rangle}$$

$$\frac{\neg\textsc{Success}(\sigma) \wedge \neg\textsc{Failure}(\sigma) \quad \textsc{Branch}(g,\sigma) = \langle (g_l, c_l), (g_r, c_r)\rangle;}{\langle g, \sigma\rangle \longmapsto \langle g_r, \sigma \wedge c_r\rangle}$$

express that a configuration $\langle g, \sigma\rangle$ can be rewritten, as the result of branching, into $\langle g_l, \sigma \wedge c_l\rangle$ or $\langle g_r, \sigma \wedge c_r\rangle$ when σ is not a solution and is not shown inconsistent.

The operational semantics characterizes the set of solutions which can be reached from a goal g and a constraint store σ by applying the transition rules. It is expressed in terms of the transitive closure $\overset{*}{\longmapsto}$ of $\longmapsto$.

Definition 2.1 (*Operational Semantics*). Let g be a goal and σ be a constraint store. The set of successful computations of $\langle g, \sigma\rangle$, denoted by $\textsc{Sol}(g,\sigma)$, is defined as

$$\textsc{Sol}(g, \sigma) = \{\sigma' \mid \langle g, \sigma\rangle \overset{*}{\longmapsto} \sigma'\}.$$

2.4.2 *Hypotheses and Properties of Search Procedures.* As mentioned, this article restricts attention to complete search procedures, that is, goals that can find all solutions to a constraint store.

Definition 2.2 (*Completeness*). Let g be a goal and σ be a constraint store. Goal g is complete for σ if $\textsc{Sol}(g, \sigma) = \textsc{Sol}(\sigma)$.

It is natural to expect that, if g is complete for σ, then it is also complete for all constraint stores σ' such that $\sigma' \Rightarrow \sigma$, since adding more constraints to a store should preserve completeness of the search procedure. This justifies the following, stronger, notion of completeness.

Definition 2.3 (*Downward Completeness*). Let g be a goal and σ be a constraint store. Goal g is downward-complete for σ if g is complete for all σ' satisfying $\sigma' \Rightarrow \sigma$.

This article also restricts attention to terminating search procedures that is, once again, natural in constraint satisfaction.

Definition 2.4 (*Termination*). Let g be a goal and σ be a constraint store. Goal g is finite for σ if there exists no infinite sequence of transition steps

$$\langle g, \sigma\rangle \longmapsto \langle g_1, \sigma_1\rangle \longmapsto \cdots \longmapsto \langle g_n, \sigma_n\rangle \longmapsto \cdots$$

The algorithms described in this article receive, as inputs, a goal g and a constraint store σ such that g is downward-complete and finite for σ. They compute the set $\textsc{Sol}(\sigma)$ or, equivalently, the set $\textsc{Sol}(g, \sigma)$. The algorithms differ in the way they explore the transition steps, that is, how they chose to explore the configurations generated by branching. We now present two properties of complete search procedures. The branching property guarantees the completeness of subgoals while the restarting property makes it possible to use the original search procedure on intermediary stores.

PROPOSITION 2.5 (BRANCHING PROPERTY). *Let g be complete for σ and let* BRANCH$(g, \sigma) = \langle (g_l, c_l), (g_r, c_r) \rangle$. *Then g_l (respectively, g_r) is complete for $\sigma \wedge c_l$ (respectively, $\sigma \wedge c_r$).*

PROOF. By specification of BRANCH, we have $\sigma \equiv (\sigma \wedge c_l) \vee (\sigma \wedge c_r)$ and hence SOL(σ) = SOL$(\sigma \wedge c_l) \cup$ SOL$(\sigma \wedge c_r)$. Since $c_l \wedge c_r$ is inconsistent, SOL$(\sigma \wedge c_l) \cap$ SOL$(\sigma \wedge c_r) = \emptyset$. Since SOL$(g, \sigma)$ is complete, we have SOL$(g, \sigma \wedge c_l)$ = SOL$(\sigma \wedge c_l)$ and SOL$(g, \sigma \wedge c_r)$ = SOL$(\sigma \wedge c_r)$ and the result follows. □

PROPOSITION 2.6 (RESTARTING PROPERTY). *Let g be downward-complete for σ and let $\langle g, \sigma \rangle \longmapsto \langle g_1, \sigma_1 \rangle \longmapsto \cdots \longmapsto \langle g_n, \sigma_n \rangle$ be a sequence of transition steps. Then,* SOL(g_n, σ_n) = SOL(g, σ_n).

PROOF. By definition of the transition steps, $\sigma_n \Rightarrow \sigma$. Since g is downward-complete for σ, then SOL(g, σ_n) = SOL(σ_n). The result follows from the completeness of g_n for σ_n. □

2.5 Depth-First Search

Figure 2 shows how to implement a depth-first strategy to detect satisfiability of a constraint store. Function DFSSATISFY expects a goal g and a constraint store σ and returns SATISFIABLE(σ). Its implementation checks whether σ is a solution or can be shown inconsistent, in which cases it returns the appropriate result. Otherwise, it applies the branching function to obtain (g_l, c_l) and (g_r, c_r). DFSSATISFY adds constraint c_l to the store σ and performs a recursive call with g_l and the new store. If the recursive call succeeds, then DFSSATISFY returns successfully. Otherwise, it backtracks to remove c_l from the constraint store and explores the right branch. *Observe that* DFSSATISFY *maintains a single constraint store which is critical in obtaining good memory performance on large applications.* The correctness of DFSSATISFY follows directly from Proposition 2.5.

Figure 2 also shows how to implement the traditional depth-first branch and bound strategy of constraint programming [Van Hentenryck 1989] to minimize an objective function f subject to a constraint store σ. Function DFSMIN expects a goal g, a constraint store σ, and an objective function f. It returns DFSMIN(f,σ). The implementation of DFSMIN first declares an integer f^* that maintains the value of the best solution found so far and initializes it to ∞. It then calls procedure DFSMINIMIZE(g, σ, f, f^*). DFSMINIMIZE updates f^* if σ is a solution. Otherwise, if σ is not a failure, it applies the branching function and explore both disjuncts recursively. Observe that the TELL operation

TELL$(\sigma, c_l \wedge f < f^*)$

dynamically adds not only c_l but also the constraint $f < f^*$. This optimization constraint restricts the search to the solutions of σ, which improves the value of the current best solution (which is dynamically adjusted). A similar observation is valid for the right disjunct as well.

2.6 Computation Paths

A computation path represents the sequence of branching decisions from the root of the search tree to a given node or, more precisely, from an initial

```
Bool DFSSatisfy(Goal g,Store σ)
{
   if Success(σ) then return true;
   else if Failure(σ) then return false;
   else {
      <(g_l, c_l),(g_r, c_r)> := Branch(g,σ);
      Tell(σ,c_l);
      if DFSSatisfy(g_l,σ) then return true;
      Backtrack(σ);
      Tell(σ,c_r);
      if DFSSatisfy(g_r,σ) then return true;
      Backtrack(σ);
      return false;
   }
}

int DFSMin(Goal g,Store σ,Expr f)
{
   f* := ∞;
   DFSMinimize(G,σ, f, f*);
   return f*;
}
void DFSMinimize(Goal g,Store σ,Expr f,int f*)
{
   if Success(σ) then
      f* := Eval(f,σ);
   else if not Failure(σ) then {
      <(g_l, c_l),(g_r, c_r)> := Branch(g,σ);
      Tell(σ,c_l ∧ f < f*);
      DFSMinimize(g_l,σ, f, f*);
      Backtrack(σ);
      Tell(σ,c_r ∧ f < f*);
      DFSMinimize(g_r,σ, f, f*);
      Backtrack(σ);
   }
}
```

Fig. 2. The implementation of depth-first search.

configuration $\langle g, \sigma\rangle$ to a configuration $\langle g_n, \sigma_n\rangle$ (respectively, σ_n) such that

$$\langle g, \sigma\rangle \stackrel{*}{\longmapsto} \langle g_n, \sigma_n\rangle \quad \text{(respectively, } \sigma_n\text{)}.$$

Computation paths have been found useful for many aspects of logic and constraint programming. They are used in parallel implementations of logic programming (e.g., Shapiro [1987] and Clocksin and Alshawi [1988]) and in incremental constraint satisfaction systems (e.g., Van Hentenryck [1990] and

```
PathRetriever(Goal g,Store σ,Path p) : Goal × Store
{
   if p = ⟨⟩ then return ⟨g, σ⟩;
   else {
      <(g_l, c_l),(g_r, c_r)> := Branch(g, σ);
      if p = ⟨left, ...⟩ then {
         Tell(σ, c_l);
         return PathRetriever(g_l, σ);
      else {
         Tell(σ, c_r);
         return PathRetriever(g_r, σ);
      }
   }
}
```

Fig. 3. A function to follow a computation path.

Van Hentenryck and Le Provost [1991]). It is thus not surprising that they are also valuable in the implementation of search strategies [Perron 1999].

In this article, a branching decision is represented by either *left* or *right* to denote whether the left or right disjunct was selected.

Definition 2.7 (*Computation Path*). A computation path is a sequence $\langle d_1, \ldots, d_n \rangle$ where $d_i \in \{left, \ right\}$ $(1 \leq i \leq n)$.

It is easy to instrument the operational semantics given earlier to collect computation paths that uniquely identify each configuration. It suffices to consider configurations of the form $\langle g, \sigma, p \rangle$ or $\langle \sigma, p \rangle$, where p is a computation path and to update the transition rules to obtain, say,

$$\frac{\neg\textsc{Success}(\sigma) \wedge \neg\textsc{Failure}(\sigma) \quad \textsc{Branch}(g, \sigma) = \langle (g_l, c_l), (g_r, c_r) \rangle;}{\langle g, \sigma, p \rangle \longmapsto \langle g_l, \sigma \wedge c_l, p :: left \rangle}$$

where $\langle d_1, \ldots, d_n \rangle :: d$ denotes the computation path $\langle d_1, \ldots, d_n, d \rangle$. The other transition rules can be instrumented similarly.

Given its computation path, it is also easy to retrieve a configuration starting from the initial configuration. Figure 3 shows a simple function to perform this task. Several algorithms presented in this paper use similar ideas.

3. LIMITED DISCREPANCY SEARCH

Limited Discrepancy Search (LDS) [Harvey and Ginsberg 1995] is a search strategy relying on a good heuristic for the problem at hand. Its basic idea is to explore the search tree in waves and each successive wave allows the heuristic to make more mistakes. Wave 0 simply follows the heuristic. Wave 1 explores the solutions which can be reached by assuming that the heuristic made one mistake. More generally, wave i explores the solutions that can be reached by assuming that the heuristic makes i mistakes.

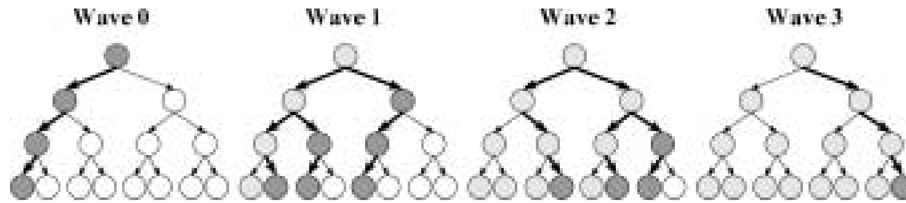

Fig. 4. The successive waves of LDS.

In our context, the heuristic always selects the left disjunct in any branching decision. Hence, wave 0 simply follows a computation path of the form ⟨*left*, ... , *left*⟩. Wave 1 explores, explicitly or implicitly, all the solutions with one discrepancy compared to the heuristic, that is, all solutions whose computation paths contain one occurrence of *right*. More generally, wave i explores, explicitly or implicitly, all the solutions whose computation paths contain i occurrences of *right*. For instance, for a uniform search tree of depth 3, LDS would explore the following waves:

Wave 0: {⟨*left*, *left*, *left*⟩}.
Wave 1: {⟨*left*, *left*, *right*⟩, ⟨*left*, *right*, *left*⟩, ⟨*right*, *left*, *left*⟩}.
Wave 2: {⟨*left*, *right*, *right*⟩, ⟨*right*, *left*, *right*⟩, ⟨*right*, *right*, *left*⟩}.
Wave 3: {⟨*right*, *right*, *right*⟩}.

Figure 4 illustrates these waves graphically. By exploring the search tree according to the heuristic, LDS may reach good solutions (and thus an optimal solution) much faster than depth-first and best-first search for some applications. Its strength is its ability to explore diverse parts of the search tree containing good solutions which are only reached much later by depth-first search. This ability of jumping to different regions of the search tree is also what makes its implementation more challenging.

4. AN ITERATIVE IMPLEMENTATION

Iterative LDS is a simple implementation of LDS obtained by slightly generalizing DFS. The key idea is to explore the search tree in waves and to relax the requirement of wave i which is now allowed to explore solutions with at most i discrepancies (instead of exactly i discrepancies). Figure 5 depicts the implementation of iterative LDS. Function ILDSSATISFY expects a goal g and a constraint store σ and returns SATISFIABLE(σ). Its implementation is a loop that calls function ILDSEXPLORE with an increasing number of allowed discrepancies, starting at 0 and stopping when the upper bound on the number of discrepancies `MaxDiscrepancies` is reached.[2] Function ILDSEXPLORE expects a goal g, a constraint store σ, and a number d of allowed discrepancies. It returns true iff there exists a solution of σ that can be reached by g in at most d discrepancies. Its implementation closely resembles DFSSATISFY. The main differences are the

[2]Observe that it is not difficult to modify ILDSSATISFY and ILDSMIN to remove this upper bound. It suffices to determine during a wave if some transition steps would have been allowed, had more discrepancies been given.

```
Bool ILDSSatisfy(Goal g,Store σ) {
   for(d := 0; d <= MaxDiscrepancies; d++)
      if ILDSExplore(g, σ ,d) then return true;
   return false;
}
Bool ILDSExplore(Goal g,Store σ,int d) {
   if d >= 0 then {
      if Success(σ) then return true;
      else if Failure(σ) then return false;
      else {
         <(g_l, c_l), (g_r, c_r)> := Branch(g, σ);
         Tell(σ, c_l);
         if ILDSExplore(g_l, σ ,d) then return true;
         Backtrack(σ);
         Tell(σ, c_r);
         if ILDSExplore(g_r, σ ,d-1) then return true;
         Backtrack(σ);
         return false;
      }
   } else return false;
}

int ILDSMin(Goal g,Store σ,Expr f) {
   f* := ∞;
   for(d := 0; d <= MaxDiscrepancies; d++)
      ILDSMinimize(g,σ , f , f*,d);
   return f*;
}
void ILDSMinimize(Goal g,Store σ,Expr f,inout int f*,int d) {
   if d >= 0 then {
      if Success(σ) then
         f* := Eval( f ,σ);
      else if not Failure(σ) then {
         <(g_l, c_l), (g_r, c_r)> := Branch(g, σ);
         Tell(σ, c_l ∧ f < f*);
         ILDSMinimize(g_l, σ, f, f*,d);
         Backtrack(σ);
         Tell(σ, c_r ∧ f < f*);
         ILDSMinimize(g_r, σ, f, f*,d-1);
         Backtrack(σ);
      }
   }
}
```

Fig. 5. An iterative implementation of LDS.

condition `d >= 0` which tests whether sufficiently many discrepancies are left and the recursive call for the right disjunct which decrements the number of allowed discrepancies. Figure 5 also presents the function ILDSMIN which is a similar generalization of DFSMIN for the minimization problem.

The correctness of ILDSSATISFY follows directly from the fact that its last call to ILDSEXPLORE, that is,

ILDSEXPLORE(g, σ, `MaxDiscrepancies`)

is equivalent to ILDSSATISFY(g, σ). The correctness of ILDSMIN follows a similar argument. The last call to ILDSMINIMIZE implicitly explores the same search tree as DFSMINIMIZE. Of course, it is likely that only a small portion of that tree is actually explored, since f^* is supposed to be optimal or near-optimal at that computation stage.

Besides the simplicity of its implementation, iterative LDS has two main benefits. On the one hand, iterative LDS inherits the memory efficiency of DFS. On the other hand, iterative LDS is very robust. Since its last iteration mimicks DFS, the branching function may be dynamic or randomized. Minimization does not raise any difficulty either and the search procedure can be easily enhanced to accommodate nogoods and global cuts. Observe however that, when the branching function is dynamic or randomized, wave i does not necessarily explore the configurations encountered in wave $i-1$. This does not affect correctness of course, since the last iteration explores, implicitly or explicitly, the whole search space.

The main inconvenience of iterative LDS is, of course, its execution time. By relaxing the requirements on the waves, iterative LDS may explore several times the same search regions. Korf [1996] proposed a clever implementation of LDS to avoid exploring the same regions when the maximum depth is known. However, the new implementation is not robust with respect to randomization, dynamic search procedures with global cuts, since there is no guarantee that the same choices will be made. We now study implementations aiming at addressing this limitation.

5. A RECOMPUTATION IMPLEMENTATION

We now present RLDS, an implementation of LDS based on recomputation. RLDS can be seen as an instantiation of the generic search strategy in Perron [1999]. The presentation in this section is a significant refinement of the search strategy algorithm presented in Van Hentenryck et al. [2000] in order to make backtracking and recomputation as explicit as possible and, more generally, to highlight all the issues involved in a recomputation implementation.

RLDS organizes the search around an active configuration and a queue of configurations. The active configuration represents the configuration that RLDS currently explores, while the queue represents configurations to be explored subsequently. The configurations in the queue are not stored explicitly but are represented by their computation paths. Each time a branching occurs on the active configuration, RLDS pushes the right disjunct onto the queue and the left disjunct becomes the active configuration. Whenever the active

configuration fails or succeeds, RLDS pops from the queue a configuration with the smallest number of discrepancies. This configuration becomes the new active configuration and RLDS proceeds as before with this new configuration and the new queue. A key step in implementing RLDS is to restore the new active configuration. Let

$$pc = \; < d_1, \ldots, d_n, d^c_{n+1}, \ldots, d^c_{n+i} >$$

be the computation path of the current active configuration, let

$$po = \; < d_1, \ldots, d_n, d^o_{n+1}, \ldots, d^o_{n+j} >$$

be the popped computation path, and let

$$< d_1, \ldots, d_n >$$

be the longest common prefix between these two paths. RLDS restores the configuration uniquely identified by *po* in two steps:

(1) it backtracks to the configuration whose path is $< d_1, \ldots, d_n >$;
(2) it recomputes the suffix $< d^o_{n+1}, \ldots, d^o_{n+j} >$.

Observe that restoring a configuration amounts to walking the shortest path between the two configurations in the search tree.

Figure 6 describes the implementation of RLDS for satisfiability. RLDSSATIFY expects a goal g and a constraint store σ and returns SATISFIABLE(σ). Its implementation creates an empty queue and calls RLDSEXPLORE. Function RLDSEXPLORE expects a goal g, a store σ, the computation path p of the configuration $\langle g, \sigma \rangle$, and a queue Q. It explores all the solutions that can be reached from $\langle g, \sigma \rangle$ and the configurations in Q. Its implementation first tests whether σ is a solution, in which case it succeeds. If σ is a failure, RLDSEXPLORE must explore the configurations in Q and thus calls RLDSEXPLOREQUEUE. Otherwise, RLDSEXPLORE applies the branching function, pushes the right disjunct onto the queue, and explore the left disjunct with the new queue. Function RLDSEXPLOREQUEUE(g, σ, pc, Q) explores the configurations in Q, starting from configuration $\langle g, \sigma \rangle$, whose path is pc. If the queue is empty, it returns false. Otherwise, it pops from the queue a path *po* with the smallest number of discrepancies, restores the configuration uniquely identified by *po*, and calls RLDSEXPLORE recursively with the new configuration and the new queue. Procedure RLDSRESTORE restores the configuration. It assumes that $< d_1, \ldots, d_n >$ is the longest common prefix of *po* and *pc*. It first backtracks to the configuration identified by this common prefix, restoring the constraint store σ and the goal g. It then *recomputes* the branching decisions $< d^o_{n+1}, \ldots, d^o_{n+j} >$ by following this suffix of *po*.

The main benefit of RLDS is to avoid exploring the same regions of the search tree several times. As a consequence, it may be substantially faster than ILDS in practice. However, RLDS also has a number of limitations. First, RLDS is more demanding in memory: it is necessary to store the computation paths of the configurations in the queue. This is a typical space/time tradeoff and the right choice probably depends on the application at hand. More important perhaps is the lack of robustness of this implementation.

```
Bool RLDSSatisfy(Goal g,Store σ) {
   Queue Q := ∅;
   return RLDSExplore(g, σ ,<>, Q);
}
Bool RLDSExplore(Goal g,Store σ ,Path p,Queue Q) {
   if Success(σ) then return true;
   else if Failure(σ) then return RLDSExploreQueue(g, σ, p, Q);
   else {
      <(g_l, c_l), (g_r, c_r)> := Branch(g, σ);
      Q.push(p :: right);
      Tell(σ, c_l);
      return RLDSExplore(g_l, σ, p :: left, Q);
   }
}
Bool RLDSExploreQueue(Goal g,Store σ ,Path pc,Queue Q) {
   if Q.isEmpty() then return false;
   po := Q.getMinDiscrepancies();
   <g, σ> := RLDSRestore(g, σ, pc, po);
   return RLDSExplore(g, σ, po, Q);
}

<Goal,Store> RLDSRestore(Goal g,Store σ ,Path pc,Path po){
   let pc = <d_1, ..., d_n, d^c_{n+1}, ..., d^c_{n+i} >;
   let po = <d_1, ..., d_n, d^o_{n+1}, ..., d^o_{n+j} >;
   for(k = 1; k <= i; k++) {
      g := Parent(g);
      Backtrack(σ);
   }
   for(k = 1; k <= j; k++) {
      ⟨(g_l, c_l), (g_r, c_r)⟩ := Branch(g, σ);
      if d^o_{n+k} = left then {
         g := g_l;
         Tell(σ, c_l);
      } else {
         g := g_r;
         Tell(σ, c_r);
      }
   }
   return ⟨g, σ⟩;
}
```

Fig. 6. A recomputation implementation of LDS: satisfiability.

RLDS, in its basic form, is not correct for *randomized branching functions*. The problem arises in function RLDSRESTORE that uses branching to return to a right disjunct. Since function BRANCH is randomized, it may not return the same disjuncts as those which led to the enqueued configuration, even if the constraint store σ is the same. As a consequence, recomputation may lead to a different configuration and introduce incompleteness. The problem could be remedied, as suggested in Van Hentenryck [1990] in the context of incremental search, by storing more information inside computation paths. For instance, RLDS could store and restore the state of all relevant random streams to ensure that BRANCH returns the very same disjuncts. This solution is not entirely satisfactory, however, since it requires to know the internal details of the search procedure and hence reduces the generic nature of the implementation.

The robustness problem is not limited to randomization. It also concerns dynamic branching functions based on the state of the constraint store *whenever the search procedure uses global cuts and/or nogood information*. In fact, the problem is readily illustrated when trying to implement minimization, since depth-first branch and bound relies on a very simple form of global cuts. Consider the "natural" generalization of RLDSSATIFY to minimization which consists of replacing the TELL operations, for example,

$$\text{TELL}(\sigma, c_l)$$

by

$$\text{TELL}(\sigma, c_l \wedge f < f^*)$$

as was the case in ILDS. This "natural" generalization is incorrect, since f^* may have changed between the time po was pushed on the queue and the time it was popped. As a consequence, function BRANCH may return different disjuncts since the constraint store may not be the same. For instance, BRANCH may choose to assign a different variable or schedule a different machine. Once again, the solution consists of adding information inside the computation path to ensure the correctness of the recomputation process. In particular, for a correct minimization, it is sufficient to use computation paths consisting of pairs (d, f), where d represents the branching decision and f represents the value of f^* before the decision. This is in fact the solution adopted in Perron [1999, page 352].

Figure 7 depicts the implementation of RLDSMIN. The functions RLDSMIN, RLDSMINIMIZE, and RLDSMINIMIZEQUEUE are essentially similar to RLDSSATIFY, RLDSEXPLORE, and RLDSEXPLOREQUEUE. Observe that both the branching decision and the current value of f^* are pushed on the queue and concatenated to computation paths. Not surprisingly, the main novelty is function RLDSRESTOREMIN which uses the augmented computation paths. The critical issues are the TELL instructions, which restore the same constraint on the objective function. *Observe that the implementation of* RLDSMIN *also illustrates the difficulty in dealing with global cuts and nogoods. To be correct, any implementation of* RLDS *must store and restore the state of the global cuts and of the nogoods inside the computation paths*. Similar observations apply to any technique that may affect the state of the constraint store such as semantic backtracking.

```
int RLDSMin(Goal g,Store σ,Expr f) {
   Queue Q := ∅;
   f* := ∞;
   return RLDSMinimize(g, σ, f, f*,<>, Q);
}
int RLDSMinimize(Goal g,Store σ,Expr f,int f*,Path p,Queue Q) {
   if Success(σ) then {
      f* := Eval(f, σ);
      return RLDSMinimize(σ, f, f*, p, Q);
   } else if Failure(σ) then return RLDSMinimizeQueue(g, σ, f, f*, p, Q);
   else {
      <(g_l, c_l), (g_r, c_r)> := Branch(g, σ);
      Q.push(p :: (right, f*));
      Tell(σ, c_l ∧ f < f*);
      return RLDSMinimize(g_l, σ, f, f*, p :: (left, f*), Q);
   }
}
int RLDSMinimizeQueue(Store σ,Expr f,int f*,Path pc,Queue Q) {
   if Q.isEmpty() then return f*;
   po := Q.getMinDiscrepancies();
   <g, σ> := RLDSRestoreMin(g, σ, pc, po);
   return RLDSMinimize(g, σ, f, f*, po, Q);
}
<Goal,Store> RLDSRestoreMin(Goal g,Store σ,Expr f,int f*,Path pc,Path po) {
   let pc = <(d_1, f_1), ..., (d_n, f_n), (d^c_{n+1}, f^c_{n+1}), ..., (d^c_{n+i}, f^c_{n+i}) >;
   let po = <(d_1, f_1), ..., (d_n, f_n), (d^o_{n+1}, f^o_{n+1}), ..., (d^o_{n+i}, f^o_{n+i}) >;
   for(k = 1; k <= i; k++) {
      Backtrack(σ);
      g := Parent(g);
   }
   f^o_{n+i+1} := f*;
   for(k = 1; k <= j; k++) {
      <(g_l, c_l), (g_r, c_r)> := Branch(g, σ);
      if d^o_{n+k} = left then {
         g := g_l;
         Tell(σ, c_l ∧ f < f^o_{n+k+1});
      } else {
         g := g_r;
         Tell(σ, c_r ∧ f < f^o_{n+k+1});
      }
   }
   return <g, σ>;
}
```

Fig. 7. A recomputation implementation of LDS: Minimization.

In summary, RLDS is an implementation of LDS that may produce substantial improvements in computation times over LDS. Its main limitation is a lack of robustness due to the recomputation process, which requires to restore the exact same computation states for the branching function. Depending on the application, this may impose significant overheads in memory space and may destroy the genericity of the algorithm.

6. A DECOMPOSITION IMPLEMENTATION

DLDS is an implementation of LDS motivated by our desire to improve the robustness of RLDS, while preserving its efficiency. As discussed in the previous section, the main difficulty in RLDS is the recomputation process that must restore the exact same computation states to reach the popped configuration.

The fundamental idea of DLDS *is to view the search procedure as a problem decomposition scheme and to reason in terms of subproblems instead of computation paths*. Viewing search procedures as decomposition schemes is, of course, not a new idea (see, e.g., instance, Freuder and Hubbe [1995]). *The novelty here is that this viewpoint can be naturally exploited to obtain an efficient and robust implementation of* LDS *and many search strategies.* Recall that the branching function returns a disjunction

$$(g_l \wedge c_l) \vee (g_r \wedge c_r)$$

and that the main goal of the algorithm is to search for solutions to the two subproblems defined by these disjuncts. The basic idea behind DLDS is thus to push subproblems in the queue instead of configurations. In other words, DLDS manipulates paths of constraints $\langle c_1, \ldots, c_n \rangle$ which, when added to the original constraint store σ, defines a subproblem.

The overall structure of DLDS resembles the architecture of RLDS. DLDS is also organized around a queue and an active configuration and has a similar overall design. However, it has three main differences:

(1) The queue does not store configurations but subproblems. These subproblems are represented as sequences of constraints.
(2) When restoring a subproblem, DLDS does not use the branching function. Instead, it simply restores the subproblems by backtracking and TELL operations.
(3) The restored subproblem is explored using the original goal, which is correct by Proposition 2.6 (the restarting property).

Figure 8 depicts the implementation of DLDSSATIFY. Function DLDSEXPLORE has an additional argument (compared to RLDSEXPLORE) representing the original goal. Observe the concatenation operation in function DLDSEXPLORE that manipulates paths of constraints instead of computation paths. The main simplification however arises in function DLDSRESTORE which restores the subproblem

$$po = \, < c_1, \ldots, c_n, c^o_{n+1}, \ldots, c^o_{n+j} >$$

```
Bool DLDSSatisfy(Goal g_o,Store σ_o) {
   Queue Q := ∅;
   return DLDSExplore(g_o, σ ,<>, Q, g_o, σ_o);
}

Bool DLDSexplore(Goal g,Store σ,Path p,Queue Q,Goal g_o,Store σ_o) {
   if Success(σ) then
      return true;
   else if Failure(σ) then
      return DLDSExploreQueue(σ, p, Q, g_o,Store σ_o);
   else {
      <(g_l, c_l), (g_r, c_r)> := Branch(g, σ);
      Q.push(p :: c_r);
      Tell(σ, c_l);
      return DLDSExplore(g_l, σ, p :: c_l, Q, g_o, σ_o);
   }
}

Bool DLDSExploreQueue(Store σ,Path pc,Queue Q,Goal g_o,Store σ_o) {
   if Q.isEmpty() then return false;
   po := Q.popMinDiscrepancies();
   σ' := DLDSRestore(σ, pc, po);
   return DLDSExplore(g_o, σ', po, Q, g_o, σ_o);
}

Store DLDSRestore(Store σ,Path pc,Path po,Store σ_o) {
   let pc = <c_1, ..., c_n, c^c_{n+1}, ..., c^c_{n+i} >;
   let po = <c_1, ..., c_n, c^o_{n+1}, ..., c^o_{n+j} >;
   for(k = 1; k <= i; k++)
      Backtrack(σ);
   for(k = 1; k <= j; k++)
      Tell(σ, c^o_{n+k});
   return σ;
}
```

Fig. 8. A decomposition implementation of LDS: Satisfiability.

from the subproblem

$$pc = \ <c_1, \ldots, c_n, c^c_{n+1}, \ldots, c^c_{n+i} >$$

where $< c_1, \ldots, c_n >$ represents the longest common prefix in *pc* and *po*. Restoring the subproblem *po* simply consists of i backtracks and j Tell operations.

DLDS is complete even for randomized branching functions because of the branching and restarting properties (Propositions 2.5 and 2.6). Indeed, the queue now stores subproblems and the branching function is only used in function DLDSExplore to decompose these subproblems. The branching function

```
int DLDSMIN(Goal g,Store σ,Expr f) {
   Queue Q := ∅;
   f* := ∞;
   return DLDSMINIMIZE(g, σ, f, f*,<>, Q, g);
}

int DLDSMINIMIZE(Goal g,Store σ,Expr f,int f*,Path p,Queue Q,Goal g_o) {
   if SUCCESS(σ) then {
      f* := EVAL(f, σ);
      return DLDSMINIMIZEQUEUE(σ, f, f*, p, Q, g_o);
   } else if FAILURE(σ) then
      return DLDSMINIMIZEQUEUE(σ, f, f*, p, Q, g_o);
   else {
      <(g_l, c_l), (g_r, c_r)> := BRANCH(g, σ);
      Q.push(p :: c_r);
      TELL(σ, c_l ∧ f < f*);
      return DLDSMINIMIZE(g_l, σ, f, f*, p :: c_l, Q, g_o);
   }
}

int DLDSMINIMIZEQUEUE(Store σ,Expr f,int f*,Path pc,Queue Q,Goal g_o) {
   if Q.isEmpty() then return f*;
   po := Q.popMinDiscrepancies();
   σ := DLDSRESTOREMIN(σ, f, f*, pc, po);
   return DLDSMINIMIZE(g_o, σ, f, f*, po, Q, g_o);
}

Store DLDSRESTOREMIN(Store σ,Expr f,int f*,Path pc,Path po) {
   let pc = <c_1, ..., c_n, c^c_{n+1}, ..., c^c_{n+i} >;
   let po = <c_1, ..., c_n, c^o_{n+1}, ..., c^o_{n+j} >;
   for(k = 1; k <= i; k++)
      BACKTRACK(σ);
   for(k = 1; k <= j; k++)
      TELL(σ, c^o_{n+k} ∧ f < f*);
   return σ;
}
```

Fig. 9. A decomposition implementation of LDS: Minimization.

is not used to restore subproblems as was the case in RLDS. Similarly, DLDS naturally accommodates minimization, global cuts and nogoods, as well as techniques like semantic backtracking which do not guarantee the same computation states upon backtracking. The implementation of DLDS for minimization is depicted in Figure 9. The main changes consist in upgrading the TELL operations, for example,

$$\text{TELL}(\sigma, c_l)$$

DLDSRESTORE(Store σ,Path pc,Path po,Store σ_o): Bool
pre: $\sigma = \sigma_o \wedge pc$;
post: $DLDSrestore(\sigma, pc, po, \sigma_o) = \sigma_o \wedge po$.

DLDSEXPLOREQUEUE(Store σ,Path p,Queue Q,Goal g_o,Store σ_o) : Bool
pre: $\sigma = \sigma_o \wedge p$;
g_o is downward-complete for σ_o;
post: succeeds if $\text{SOL}(\langle g_o, \sigma' \rangle) \neq \emptyset$ for some $\sigma' \in Q$.

DLDSEXPLORE(Goal g,Store σ,Path p,Queue Q,Goal g_o,Store σ_o) : Bool
pre: $\sigma = \sigma_o \wedge p$;
g_o is downward-complete for σ_o;
g is complete for σ;
post: succeeds if $\text{SOL}(\langle g, \sigma \rangle) \neq \emptyset$ or $\text{SOL}(\langle g_o, \sigma' \rangle) \neq \emptyset$ for some $\sigma' \in Q$.

DLDSSATISFY(Goal g_o,Store σ_o) : Bool
pre: g_o is downward-complete for σ_o;
post: succeeds if $\text{SOL}(\langle g_o, \sigma_o \rangle) \neq \emptyset$.

Fig. 10. Specification of the main subfunctions.

to include the objective function constraint, for example,

$$\text{TELL}(\sigma, c_l \wedge f < f^*)$$

As a consequence, DLDS combines the runtime efficiency of RLDS with the robustness of ILDS.

Before discussing the implementation issues of RLDS and DLDS, it is useful to conclude this section by mentioning a variation of DLDS. Indeed, from the above presentation, it may seem natural to extend DLDS to store the goals g_l and g_r in addition to c_l and c_r. This would allow the subproblems to be solved by their corresponding goals and not the original goal. This certainly is feasible and correct when g_l and g_r are easily available and can be compactly represented. In many constraint languages, however, goals are only available indirectly through continuations and this variation is thus hard to implement efficiently in a generic fashion. In addition, as the experimental results indicate, it may actually be a good idea to restart with the initial goal.

7. CORRECTNESS

We now prove the correctness of the decomposition-based implementation. We focus on satisfiability, since the proof for minimization is similar. In this section, given $\sigma_1 = \langle c_1, \ldots, c_n \rangle$ and $\sigma_2 = \langle c_{n+1}, \ldots, c_m \rangle$, we use the notation $\sigma_1 \wedge \sigma_2$ to denote the constraint store/path $\sigma = \langle c_1, \ldots, c_n, c_{n+1}, \ldots, c_m \rangle$. Figure 10 gives the specifications of the main subfunctions of the algorithm. We now prove the correctness of their implementations with respect to the specifications.

LEMMA 7.1 (CORRECTNESS OF DLDSEXPLOREQUEUE). *The implementation of* DLDSEXPLOREQUEUE *satisfies its specification.*

PROOF. The result obviously holds when the queue is empty. Otherwise, let po be an element of Q and $Q' = Q \setminus \{po\}$. Since $\sigma = \sigma_o \wedge pc$, it follows from the specification of DLDSRESTORE that $\sigma' = \sigma_0 \wedge po$. From the specification of DLDSEXPLORE, it follows that DLDSEXPLOREQUEUE(σ, pc, Q, g_o) succeeds iff DLDSEXPLORE(g, σ', po, Q, g_o) succeeds. □

LEMMA 7.2 (CORRECTNESS OF DLDSEXPLORE). *The implementation of* DLDSEXPLORE *satisfies its specification.*

PROOF. The result obviously holds for the base cases. Consider now the recursive case. It follows from the specification of BRANCH that $\sigma \Leftrightarrow (\sigma \wedge c_l) \vee (\sigma \wedge c_r)$. Since g is complete for σ, by Proposition 2.5, g_l is complete for $\sigma \wedge c_l$ and g_r is complete for $\sigma \wedge c_r$. Hence, $\text{SOL}(\langle g, \sigma\rangle) = \text{SOL}(\langle g_l, \sigma \wedge c_l\rangle) \cup \text{SOL}(\langle g_r, \sigma \wedge c_r\rangle)$. Moreover, since g_o is downward-complete for σ_o, it follows that g_o is downward-complete for σ. As a consequence, by Proposition 2.6, $\text{SOL}(\langle g_r, \sigma \wedge c_r\rangle) = \text{SOL}(\langle g_o, \sigma \wedge c_r\rangle)$. The results now follows from Lemma 7.1, since the recursive call is of the form DLDSEXPLOREQUEUE(g_l, $\sigma \wedge c_l$, $p :: c_l$, $Q \cup \{p :: c_r\}$, g_o, σ_o). □

THEOREM 7.3 (CORRECTNESS OF DLDSSATISFY). *The implementation of* DLDS SATISFY *satisfies its specification.*

PROOF. The proof is a direct consequence of Lemma 7.2. □

8. IMPLEMENTATION ISSUES

The main implementation issue in RLDS and DLDS concerns the memory management of the queue. Recall that the queue in these algorithms contains sequences. In the case of RLDS, the queue contains sequences of pairs (d, f) for minimization, where d is a branching decision and f is the value of f^* before the decision. Additional information, such as global cuts and random values, may need to be stored together with (d, f) for more advanced applications. For DLDS, the sequence contains constraints.

The queue in these algorithms can be implemented with a trie structure [Knuth 1998], a compact data structure for sets of sequences. A trie factorizes common prefixes in a collection of sequences. In its simple form, a trie is a tree whose nodes store a sequence element. A node n at depth i represents the sequence $\langle e_0, \ldots, e_i\rangle$ of the $i+1$ elements encountered on the path from the root to n. In the case of RLDS, the nodes only contain the value f, since the branching decision is implicitly represented in the trie. Of course, more information needs to be stored in presence of randomization, global cuts, and similar techniques. In the case of DLDS, the nodes only contain the constraints c_l and c_r encountered during branching. The trie is a compact data structure, when the sequences have long common prefixes, which is typically the case in search procedures.

It is easy to see why tries are appealling, even for RLDS. Consider, for instance, a complete tree of depth d. The search tree has about 2^{d+1} nodes. For RLDS, the trie would require three words inside each node: the value f and two pointers to its children. Without a trie, RLDS would have to store, for all the nodes, the value f as well as the computation paths. Even if the path is

coded by a bit sequence, it would still require $\lceil d/32 \rceil$ words. In addition, it will be necessary to store the length of the path and to link the sequences. As a consequence, such an explicit representation will always require more space in this example, even when d is small. For DLDS, the trie is obviously better than an explicit representation.

9. GENERALIZATION TO GENERAL SEARCH STRATEGIES

As shown in Perron [1999] (see also Van Hentenryck et al. [2000]), many search strategies can be expressed in terms of two functions: an evaluation function that assigns a value to every configuration and a suspension function that decides whether to suspend the current configuration and to switch to another configuration in the queue. In LDS, the evaluation function returns the number of discrepancies of the configuration, while the suspension function never suspends.

DLDS can easily be adapted to this broader framework. It suffices to assume the existence of a function SUSPEND(g,σ, Q) that returns true if configuration $\langle g, \sigma \rangle$ must be suspended in the context of queue Q and to insert the instruction

```
if SUSPEND(g, σ, Q) then {
   Q.push(p);
   return EXPLOREQUEUE(σ, p, Q, g0);
}
```

before the final `else` in the DLDSEXPLORE function. Figure 11 depicts the generalization for deciding satisfiability.

10. EXPERIMENTAL RESULTS

We now report experimental results on the algorithms presented in this article to show the practicality of our new implementation. The purpose of the experimental results is to show that DLDS can be implemented with reasonable overheads in memory and in time compared to RLDS, which would make it a good candidate for circumstances where robustness is a main issue.

All algorithms were implemented in C++ (under LINUX on an AMD 800 Mhz processor) and tested on a traditional set of job-shop scheduling problems. (We use the abbreviations A5=ABZ5, A6=ABZ6, L19=LA19, L20=LA20, MT=MT10, O1=ORB1, O2=ORB2, O3=ORB3, O4=ORB4, and O5=ORB5 in the tables.)

Our implementation is based on the edge-finder algorithm presented in Baptiste and Pape [1995]. In particular, the algorithm uses the edge finder for pruning and its search procedure iterates the following step: select the machine with the smallest global slack and rank the machine. To rank a machine, the algorithm selects a task that can be scheduled before all unranked tasks on the machine. If r is the smallest rank available on that machine and if $rank$ is the variable representing the rank of the selected task, the constraints c_l and c_r in a branching decision can be thought of as $rank = r$ and $rank \neq r$. The purpose of the algorithm is to minimize makespan, that is, the total duration of the project. As a consequence, the experiments compare algorithms

```
Bool SATISFY(Goal g,Store σ) {
   Queue Q := ∅;
   return EXPLORE(g, σ ,<>, Q, g);
}

Bool Explore(Goal g,Store σ,Path p,Queue Q,Goal g_o) {
   if SUCCESS(σ) then
      return true;
   else if FAILURE(σ) then
      return EXPLOREQUEUE(σ, p, Q, g_o);
   else if SUSPEND(g, σ, Q) then {
      Q.push(p);
      return EXPLOREQUEUE(σ, p, Q, g_0);
   } else {
      <(g_l, c_l), (g_r, c_r)> := BRANCH(g, σ);
      Q.push(p :: c_r);
      TELL(σ, c_l);
      return EXPLORE(g_l, σ, p :: c_l, Q, g_o);
   }
}

Bool EXPLOREQUEUE(Store σ,Path pc,Queue Q,Goal g_o) {
   if Q.isEmpty() then return false;
   po := Q.popMinEvaluation();
   σ := RESTORE(σ, pc, po);
   return EXPLORE(g_o, σ, po, Q, g_o);
}

Store RESTORE(Store σ,Path pc,Path po) {
   let pc = <c_1, ..., c_n, c^c_{n+1}, ..., c^c_{n+i} >;
   let po = <c_1, ..., c_n, c^o_{n+1}, ..., c^o_{n+j} >;
   for(k = 1; k <= i; k++)
      BACKTRACK(σ);
   for(k = 1; k <= j; k++)
      TELL(σ, c^o_{n+k});
   return σ;
}
```

Fig. 11. A decomposition implementation of search strategies: Satisfiability.

ILDSMIN, RLDSMIN, and DLDSMIN. Both RLDS and DLDS use tries to represent the queue compactly. In addition, the results also compare implementations of these algorithms where the number of discrepancies is increased by more than one in each wave. For RLDS and DLDS, these variations can be implemented as an instance of the general algorithm where the suspension function postpones the active configuration when its number of discrepancies exceeds the smallest number of discrepancies in the queue plus the increment. The algorithm can

Table I. CPU Times in Seconds of the Implementations

	A5	A6	L19	L20	MT	O1	O2	O3	O4	O5
DFS	29.1	4.8	43.1	12.4	48.6	317.3	60.7	2397.1	60.3	671.6
ILDSMIN	116.8	26.3	225.7	62.9	459.2	84.1	120.2	930.8	540.1	65.2
RLDSMIN	72.8	19.3	72.0	37.4	176.0	75.4	62.8	394.3	168.7	101.6
DLDSMIN	28.0	9.7	34.7	16.7	81.3	26.3	30.3	174.6	169.6	30.3
ILDS(2)	71.4	14.8	129.6	38.9	253.2	50.3	65.0	532.6	301.0	48.0
RLDS(2)	57.6	12.2	55.3	26.0	107.0	49.7	42.4	315.7	107.6	87.4
DLDS(2)	24.6	8.0	31.0	13.9	72.3	24.1	26.8	172.9	153.6	39.3
ILDS(3)	50.9	11.2	91.6	30.8	180.2	38.2	34.3	374.6	213.9	42.9
RLDS(3)	53.9	8.8	63.2	22.8	89.7	33.2	30.7	244.4	101.0	94.5
DLDS(3)	27.4	7.6	33.8	15.7	49.7	21.4	20.8	163.0	132.8	44.2
ILDS(4)	40.9	8.7	85.2	25.3	136.9	30.6	40.0	433.5	187.1	67.3
RLDS(4)	40.4	7.4	46.5	20.4	66.5	35.7	31.4	422.8	84.6	154.7
DLDS(4)	24.1	7.7	28.8	12.2	53.3	18.6	21.0	1139.5	126.7	34.3

Table II. Memory Space in Kilobytes of the Implementations

	A5	A6	L19	L20	MT	O1	O2	O3	O4	O5
ILDSMIN	15	15	15	15	15	15	15	15	15	15
RLDSMIN	1535	255	1149	518	1980	622	910	4080	1533	1389
DLDSMIN	822	263	910	355	1189	367	614	3058	2427	1980
ILDS(2)	15	15	15	15	15	15	15	15	15	15
RLDS(2)	1033	231	1085	415	1564	590	806	4135	1349	1772
DLDS(2)	942	247	998	439	1173	431	574	4878	2275	4064
ILDS(3)	15	15	15	15	15	15	15	15	15	15
RLDS(3)	1253	183	1250	375	1293	447	646	3285	1309	2235
DLDS(3)	1245	207	1053	399	718	471	503	4447	1780	3105
ILDS(4)	15	15	15	15	15	15	15	15	15	15
RLDS(4)	870	143	982	327	934	503	630	6299	1085	3824
DLDS(4)	1045	231	1149	495	934	566	638	31658	1988	3640

then be optimized to avoid pushing elements in the queue by using backtracking to explore the various waves. This implementation reduces the memory requirements but it may also limit the benefits of LDS, since the algorithm may now explore configurations of a later wave before configurations of an earlier wave. As a consequence, it should be used with special care.

Tables I and II report the runtime and memory performance of the algorithms. Table I reports the CPU time in seconds for the various algorithms and benchmarks. Table II reports the memory requirements in kilobytes. Table III gives the ratios RLDS/ILDS, DLDS/ILDS, DLDS/RLDS for the CPU times and Table IV reports the ratio DLDS/RLDS for memory space. Figures 12, 13, 14, and 15 depict these results graphically. The experimental results are surprising. As mentioned, our goal in implementing the algorithm was to confirm that DLDS was a viable alternative to RLDS. Indeed, given the data structures used

Table III. CPU Times Ratios of the Implementations

	A5	A6	L19	L20	MT	O1	O2	O3	O4	O5
RLDSMIN/ILDSMIN	0.6	0.7	0.3	0.6	0.3	0.9	0.5	0.4	0.3	1.5
DLDSMIN/ILDmin	0.2	0.3	0.1	0.2	0.1	0.3	0.1	0.1	0.3	0.4
DLDSMIN/RLDSmin	0.4	0.5	0.4	0.9	0.4	0.3	0.4	0.4	1.0	0.3
RLDS(3)/ILDS(3)	1.0	0.7	0.6	0.7	0.5	0.8	0.9	0.6	0.4	2.2
DLDS(3)/ILDS(3)	0.5	0.6	0.3	0.5	0.2	0.5	0.6	0.4	0.6	1.0
DLDS(3)/RLDS(3)	0.5	0.8	0.5	0.6	0.5	0.6	0.6	0.6	1.3	0.4

Table IV. Memory Space Ratios of the Implementations

	A5	A6	L19	L20	MT	O1	O2	O3	O4	O5
RLDSMIN/DLDSMIN	0.5	1.0	0.7	0.6	0.6	0.5	0.6	0.7	1.5	1.4
RLDS(3)/DLDS(3)	0.9	1.1	0.8	1.0	0.5	1.0	0.7	1.3	1.3	1.3

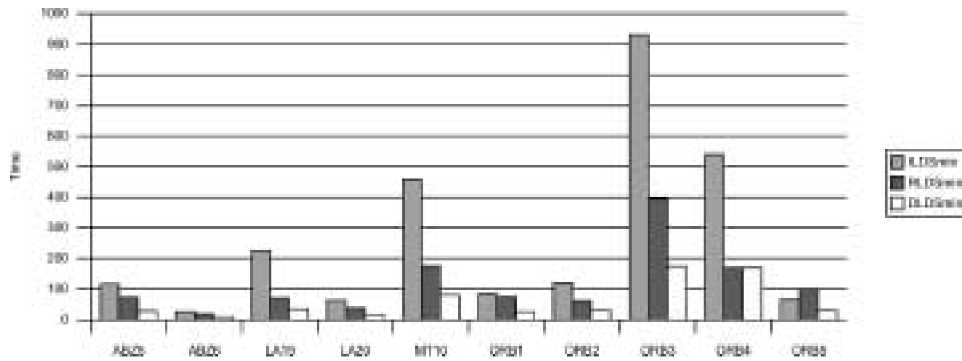

Fig. 12. Time comparison of ILDSMIN, RLDSMIN, and DLDSMIN.

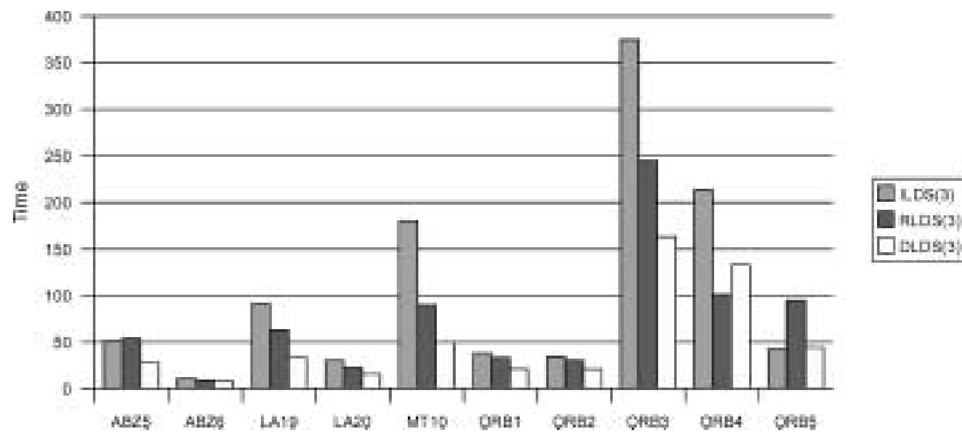

Fig. 13. Time comparison of ILDS(3), RLDS(3), and DLDS(3).

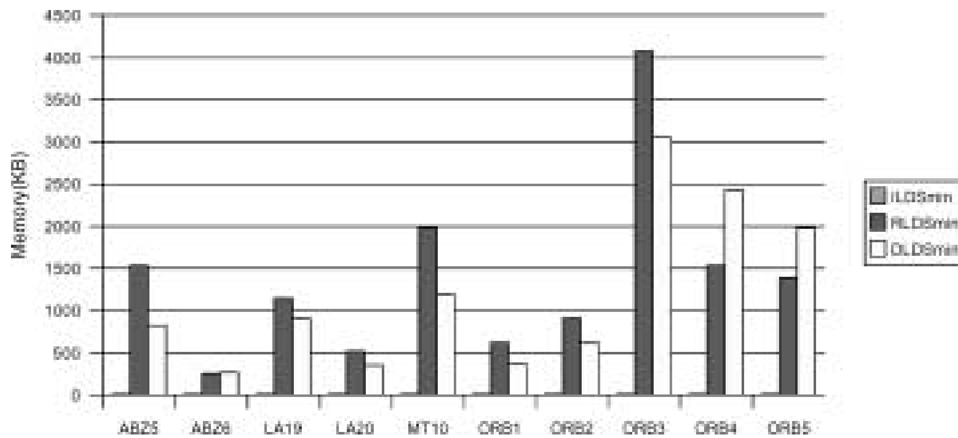

Fig. 14. Memory comparison of ILDSMIN, RLDSMIN, and DLDSMIN.

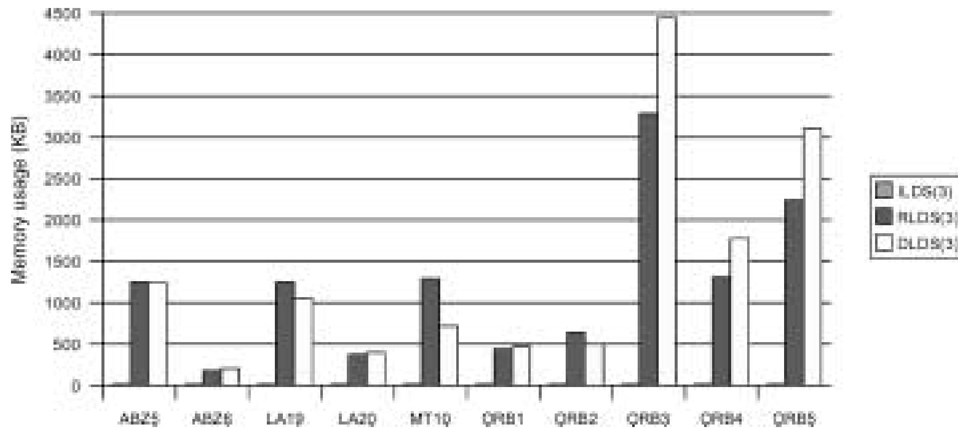

Fig. 15. Memory comparison of ILDS(3), RLDS(3), and DLDS(3).

in the implementation, we expected DLDS to take about four times the space as RLDS (the price for robustness), while we expected them to run at about the same speed.

10.1 The Main Algorithms

The results on the main algorithms, that is, ILDSMIN, RLDSMIN, and DLDSMIN, show that DLDSMIN outperforms RLDSMIN on this set of benchmarks, generally by a factor 2. DLDSMIN is sometimes about 3 times as fast as RLDSMIN and is almost never slower (it is slightly slower on ORB4). More surprising perhaps is the memory consumption of DLDSMIN compared to RLDSMIN. It is often the case that DLDSMIN requires less memory than RLDSMIN and DLDSMIN requires only 1.58 times the space of RLDSMIN in the worst case (ORB4 once again). DLDSMIN requires about three megabytes of memory in the worst case (instead of about 4 megabytes for RLDSMIN). Observe also that DLDSMIN is significantly faster

than ILDSMIN: It is about four times faster in the average and at least twice as fast on these benchmarks.

How to explain these results? *The good performance of* DLDSMIN *on this set of benchmarks comes from its restarting strategy which explores subproblems using the original procedure.* The restarting strategy applies the heuristic to subproblems that differ from the original problem in two aspects: the new decomposition constraints and the optimization constraint (if a solution has been found). By restarting the search procedure, DLDSMIN uses a more informed heuristic which leads to a smaller search space and thus to a better time and memory performance. Observe also that ILDS has some aspects of a restarting strategy since each new wave inherits a new optimization constraint when a solution was found in the previous wave and the heuristic may exploit this additional information to guide the search.

It is important to stress that restarting strategies are not novel in combinatorial optimization: they were advocated in Salkin [1970] and were part of the CHIP system since its inception [Van Hentenryck 1989]. The main idea is to restart the search procedure each time a new solution is found. As a consequence, the heuristic in the search procedure may make use of this additional information to guide the search more precisely. Restarting strategies are not used very often because of the heavy recomputation costs and runtime overheads they may induce. DLDS avoids this time overhead and hence it may be interesting to reconsider restarting strategies in light of the present experimental results.

10.2 The Generalized Algorithms

The results on the generalized algorithms, where discrepancies are increased by more than 1 at every wave, are also interesting. These algorithms are denoted ILDS(n), RLDS(n), and DLDS(n). For ILDS, choosing $n > 1$ may reduce the overhead of recomputation. For RLDS, such a choice may reduce the memory requirements, since backtracking may be used to explore the successive waves, thus reducing the number of pushes on the queue. Of course, choosing $n > 1$ may also endanger some of the benefits of LDS, since the algorithm moves closer to DFS. The experimental results indicate that DLDS(3) is still significantly faster than RLDS(3) and the memory requirements of the algorithms are reasonably close. DLDS(3) is also significantly faster than ILDS(3). The choice $n = 4$ illustrates the pitfalls of increasing the discrepancies too much. Indeed, ILDS(4), RLDS(4), and DLDS(4) are all slower, sometimes significantly, on ORB4, because they did not follow the heuristic closely enough. The high figures on time and memory for DLDS(4) come from the large search space to explore in these cases.

It is interesting to observe that the benefits of choosing $n > 1$ is not that significant for DLDS and is certainly much less significant than for ILDS and RLDS. In particular, DLDS is almost always faster than ILDS(n) and RLDS(n) for any n. This is a nice property of DLDSMIN. It essentially removes the need to explore larger waves, that may produce significant slowdowns. This is in contrast with RLDS, where increasing the wave sizes may increase execution time and reduce memory significantly.

10.3 Summary

In summary, the experimental results certainly confirm the viability of DLDS as an alternative to RLDS. In fact, as a side-effect of our desire to improve the robustness and the applicability of RLDS, DLDS almost always outperforms RLDS in execution time and has similar memory requirements on this set of benchmarks. These results were a surprise to us and they indicate the potential interest of restarting strategies that makes it possible to accommodate new dynamic information into the heuristic. The experimental results also indicate that DLDSMIN does not benefit significantly from increasing the wave sizes, which is an important property to avoid the pitfalls typically associated with larger wave sizes.

10.4 Discussion

It is important to conclude this section by a few observations. The main motivation behind this article was not to advocate restarting as a heuristic but rather to propose a new implementation technique for search strategies that accommodates advanced features such as randomization and global cuts easily and efficiently. The above results show that a decomposition-based implementation is indeed an attractive alternative in this context. We were not expecting that the decomposition approach would outperform the existing implementation schemes on this traditional set of benchmarks. Clearly, the use of restarting is beneficial in this respect, but it is essentially an unanticipated side-effect. One may argue that the implementation schemes should be compared on the same search trees, for example, by using a heuristic that chooses the machine to schedule next at each step. However, even in this case, the implementations will not explore the same search trees. Indeed, the decomposition-based approach may choose a different machine because it uses additional information from the new bounds that the traditional recomputation implementation cannot exploit. As a consequence, there are interferences between the search strategies and the heuristics in each of the approaches that cannot be avoided due to the nature of dynamic search procedures.

It is useful however to give further evidence of the practicability of the decomposition scheme by carefully examining the behavior of RLDSMIN and DLDSMIN on two benchmarks: ORB4, where their execution times are essentially the same, and ABZ5, where DLDSMIN is significantly faster. The number of branchings on ORB4 for RLDSMIN and DLDSMIN is 35,434 and 42,855 respectively. This indicates that, on this benchmark, DLDSMIN performs slightly more branchings per second than RLDSMIN. The number of branchings on ABZ5 for RLDSMIN and DLDSMIN is 28,751 and 17,096 respectively. DLDSMIN explores a significantly smaller tree in this case, but it also performs more branchings per second. Overall, these results seem to indicate that DLDSMIN does not impose any significant overhead over RLDSMIN.

11. CONCLUSION

In recent years, much research has been devoted to search strategies for constraint satisfaction. In particular, LDS and its variations were shown to produce

significant gain in efficiency over depth-first or best-first search. In addition, recomputation-based schemes were proposed as a general and efficient way to implement search strategies generically.

This article was motivated by the increasing use of sophisticated search procedures in practical applications. These search procedures may, for instance, involve randomization, dynamic branching functions, global cuts and nogoods, probes, and semantic backtracking to name only a few. These highly dynamic search procedures raise fundamental challenges for recomputation-based schemes that must restore exactly the same branchings during recomputation. As a consequence, recomputation-based implementations need to enhance computation paths with additional information to guarantee correctness, which may increase memory consumption significantly and decrease the genericity of the implementation.

To remedy these limitations, this article proposed a decomposition-based implementation of search strategies. By working directly in terms of subproblems instead of computation paths, decomposition-based implementations avoid the pitfalls of recomputation and combine the efficiency of recomputation-based schemes with the robustness needed in advanced applications. The implementation makes use of a trie structure to represent subproblems compactly.

Experimental results demonstrate that decomposition is a viable alternative to recomputation. On a set of traditional job-shop scheduling benchmarks, decomposition significantly outperforms recomputation while consuming roughly the same memory space. Decomposition is also shown to be rather insensitive to the wave size, freeing users from the burden of finding a good compromise between the wave size and efficiency. These results were surprising, since we expected decomposition and recomputation schemes to behave rather similarly. The benefits of decomposition come from its restarting strategy which solves subproblems using the original search procedure. As a consequence, it may be valuable to reconsider restarting strategies in other contexts, since they may boost performance significantly. Recent theoretical results (e.g., Chen et al. [2001]) indicate that this indeed is an important avenue for further research.

REFERENCES

Baptiste, P. and Pape, C. L. 1995. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of International Joint Conference on Aritificial Intelligence (IJCAI-95)*. 600–606.

Chen, H., Gomes, C., and Selman, B. 2001. Formal models of heavy-tailed behavior in combinatorial search. In *Proceedings of the 7th International Conference on the Principles and Practice of Constraint Programming (CP'01)*. Cyprus.

Clocksin, W. and Alshawi, H. 1988. A method for efficiently executing Horn clause programs using multiple processors. *New Gen. Comput. 5*, 361–376.

Colmerauer, A. 1990. An introduction to Prolog III. *Commun. ACM 28*, 4, 412–418.

El Sakkout, H. and Wallace, M. 2000. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints 5*, 4 (Oct.).

Fikes, R. 1968. A heuristic program for solving problems stated as non-deterministic procedures. Ph.D. dissertation, Comput. Sci. Dept., Carnegie-Mellon Univ. Pittsburgh, Pa.

Freuder, E. and Hubbe, P. 1995. A disjunctive decomposition control schema for constraint satisfaction. In *Principles and Practice of Constraint Programming*, V. Saraswat and P. V. Hentenryck, Eds. The MIT Press, Cambridge, Mass.

HARALICK, R. AND ELLIOT, G. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intel. 14*, 263–313.
HARVEY, W. AND GINSBERG, M. 1995. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (Montreal, Ont., Canada).
HOOKER, J. 2000. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. Wiley, New York.
HOOKER, J., OTTOSSON, G., THORNSTEINSSON, E., AND KIM, H.-J. 2001. A scheme for unifying optimization and constraint satisfaction methods. *Knowl. Eng. Rev. 15*, 11–30.
JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint logic programming. In *Proceedings of POPL-87* (Munich, Germany).
KNUTH, D. 1998. Sorting and Searching. In *The Art of Computer Programming*, vol. 3 (Second Edition). Addison-Wesley, Reading, Mass.
KORF, R. 1996. Improved limited discrepancy search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)* (Portland, Ore.). 209–215.
LABURTHE, F. AND CASEAU, Y. 1998. SALSA: A language for search algorithms. In *Proceedings of the 4h International Conference on the Principles and Practice of Constraint Programming (CP'98)* (Pisa, Italy).
LAURIERE, J.-L. 1978. A language and a program for stating and solving combinatorial problems. *Artif. Intel. 10*, 1, 29–127.
MACKWORTH, A. 1977. Consistency in networks of relations. *Artif. Intel. 8*, 1, 99–118.
MESEGUER, P. 1997. Interleaved depth-first search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (Nagoya, Japan).
MONTANARI, U. 1974. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci. 7*, 2, 95–132.
NUIJTEN, W. 1994. Time and resource constrained scheduling: A constraint satisfaction approach. Ph.D. dissertation, Eindhoven University of Technology.
PERRON, L. 1999. Search procedures and parallelism in constraint programming. In *Proceedings of the 5th International Conference on the Principles and Practice of Constraint Programming (CP'99)* (Alexandra, Va.).
PLOTKIN, G. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, CS Department, University of Aarhus.
REFALO, P. 1999. Tight cooperation and its application in piecewise linear optimization. In *Proceedings of the 5th International Conference on the Principles and Practice of Constraint Programming (CP'99)* (Alexandra, Va.).
REFALO, P. AND VAN HENTENRYCK, P. 1996. CLP($\mathcal{R}_{lin}$) Revised. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming* (Bonn, Germany).
SALKIN, H. 1970. On the merit of the generalized origin and restarts in implicit enumeration. *Oper. Res. 18*, 549–554.
SCHULTE, C. 1997. Programming constraint inference engines. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming* (Schloss Hagenberg, Linz, Austria). Lecture Notes in Computer Science vol. 1330. Springer-Verlag, New York, 519–533.
SCHULTE, C. 2000. Programming deep concurrent constraint combinators. In *Proceedings of the 2nd International Workshop on Practical Aspects of Declarative Languages (PADL'00)* (Boston, Mass.) Lecture Notes in Computer Science vol. 1753. Springer-Verlag, New York.
SHAPIRO, E. 1987. An OR-parallel execution algorithm for PROLOG and Its FCP implementation. In *Proceedings of the 4th International Conference on Logic Programming* (Melbourne, Australia). 311–337.
VAN HENTENRYCK, P. 1989. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, Mass.
VAN HENTENRYCK, P. 1990. Incremental Constraint Satisfaction in Logic Programming. In *Proceedings of the 7th International Conference on Logic Programming* (Jerusalem, Israel).
VAN HENTENRYCK, P. 1999. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass.
VAN HENTENRYCK, P. AND LE PROVOST, T. 1991. Incremental constraint satisfaction in logic programming. *New Generation Computing*. (Selected Paper from ICLP'90).

Van Hentenryck, P., Perron, L., and Puget, J.-F. 2000. Search and strategies in OPL. *ACM Transactions on Computational Logic 1*, 2 (October), 1–36.

Walsh, T. 1997. Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*. (Nagoya, Japan).

Waltz, D. 1972. Generating semantic descriptions from drawings of scenes with shadows. Tech. Rep. AI271. MIT, Cambridge, Mass, Nov.

Wolsey, L. 1998. *Integer Programming*. Wiley, New York.

Received August 2001; revised April 2002 and November 2002; accepted November 2002