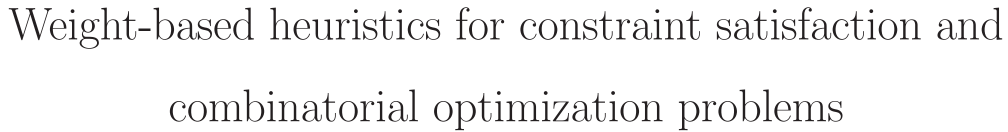
hal-00660715, version 1 - 17 Jan 2012



Weight-based heuristics for constraint satisfaction and combinatorial optimization problems

Marie-José Huguet Pierre Lopez Wafa Karoui

CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM ; LAAS ; F-31077 Toulouse, France
e-mails: {huguet,lopez}@laas.fr, karoui.wafa@gmail.com

Abstract

In this paper, we propose mechanisms to improve instantiation heuristics by incorporating weighted factors on variables. The proposed weight-based heuristics are evaluated on several tree search methods such as chronological backtracking and discrepancy-based search for both constraint satisfaction and optimization problems. Experiments are carried out on random constraint satisfaction problems, car sequencing problems, and jobshop scheduling with time-lags, considering various parameter settings and variants of the methods. The results show that weighting mechanisms reduce the tree size and then speed up the solving time, especially for the discrepancy-based search method.

Keywords: Weight-based heuristics, tree search, constraint satisfaction, scheduling.

1 Introduction

This paper deals with the solving of combinatorial problems expressed in the *Constraint Satisfaction Problem* (CSP) formalism. A CSP is defined by a triple (X, D, C) where $X = \{X_1, \ldots, X_n\}$ is a finite set of variables, $D = \{D_1, \ldots, D_n\}$ is the set of domains for each variable, each D_i being the set of discrete values for variable X_i, and $C = \{C_1, \ldots, C_m\}$ is a set of constraints [8, 32]. For optimization problems, an objective function f is added to the problem definition and we talk about a *Constraint Satisfaction Optimization Problem* (CSOP). An instantiation of a subset of variables corresponds to an assignment of these variables by a value taken from their domain. An instantiation is said to be complete when it concerns all the variables of X. Otherwise, it is called a partial instantiation. A solution is a complete instantiation satisfying the constraints. An inconsistency in the problem is raised as soon as a partial instantiation

hal-00660715, version 1 - 17 Jan 2012

cannot be extended to a complete one. For a CSOP, a solution s^* is optimal if its cost $f(s^*)$ is lower than or equal to the cost of all other solutions (for a minimization problem).

Following this definition, it is clear that CSPs can model a broad spectrum of decision problems, from satisfiability problems to optimization problems. Solving CSPs is a general research domain based on tree search methods embedding three main components: search tree expansion strategy, constraint propagation mechanisms, and variable/value ordering heuristics. CSPs are known to be NP-complete problems and each component has a strong impact on the quality of the search method. CSPs appear in various domains and many search algorithms have been developed to solve them. In this paper, we are concerned with complete methods that have the advantage of finding at least a solution to a problem, if such a solution exists. A widely studied class of complete algorithms relies to depth first search and backtracking mechanisms.

More particularly, the purpose of this paper is to propose improving techniques for tree search. The method that we privileged here is discrepancy search, an alternative to depth first search (the principles and references are given in the next section on the scientific background). We then propose to analyze the causes of failures in the search tree and derive variables weighting for ordering heuristics. In the first part of the paper, we use these techniques for constraint satisfaction problems, in particular randomly generated CSPs and car-sequencing instances. A variant, named as `YIELDS`, of the seminal limited discrepancy search (`LDS`) method serves as a support for developing the search tree. In the second part, the techniques are adapted for handling combinatorial optimization problems. A climbing discrepancy search (`CDS`) variant with weighted factors is proposed for jobshop scheduling with time-lags. We selected this particular scheduling problem for its intrinsic genericity, as well as its practical relevance in the process industry.

The paper is organized as follows: The next section provides a brief overview and background on search methods and ordering heuristics. Section 3 describes a heuristic based on variable weighting and its integration in a discrepancy-based search method to solve different constraint satisfaction problems, namely random CSPs and car-sequencing benchmarks. Section 4 adapts the previous mechanisms for an optimization context (shop scheduling with makespan minimization). It reports computational results on jobshop scheduling problems with time-lags. The last section concludes this study and gives some ideas for further works in this area.

2 Background

In a tree search method, at each step, a partial solution is extended by assigning a value to an extra variable. When none of the values of a variable are consistent with the partial solution (dead-end), backtracking takes place. This kind of methods is usually stopped either as soon as a solution is obtained or when the complete tree has been explored. In the worst case, it needs an exponential time in the number of variables. Improvements of backtracking algorithm have focused on the three phases of the algorithm [8]: ordering heuristics, moving forward (*look-ahead* schemes), and backtracking (*look-back* schemes). The most common principle for performing systematic search traverses the space of partial solutions in a depth-first manner.

Chronological Backtracking (`CB`) is a well-known method based on the depth-first search principle for solving combinatorial problems. The method `CB` extends a partial instantiation by assigning to a new variable a value, which is consistent with the previous instantiated variables. When a dead-end appears, it goes back to the latest instantiated variable trying another value.

Forward-Checking (`FC`) and *Arc-Consistency* (`AC`) are two inference method types, which can be associated with (`CB`) (or any tree search methods), respectively denoted hereafter by `CB-FC`, and `CB-AC` *a.k.a.* `MAC` (*Maintaining Arc-Consistency*). For `CB-FC`, propagations are limited to variables in the neighborhood of the latest instantiation. `MAC` suppresses inconsistent values in the domain of all uninstantiated variables. Although `CB-FC` was considered as the best instantiation algorithm for a long time [29], `MAC` is now recognized as one of the most performing existing method [36].

Variable and value ordering heuristics may have a great impact for solving decision problems. They were studied in many various fields like SAT, CSP, or combinatorial optimization [18, 24]. They aim to provide an order on variables and values to speed up the search for obtaining a solution (possibly having good quality). The proposed order for the selection of the next variable or the next value for a variable instantiation can be static (*i.e.,* the orders are definitively chosen at the beginning of the search) or dynamic (*i.e.,* the orders may change during the search). Variable and value ordering heuristics are generally based on opposite principles. Variable ordering heuristics exploit the *fail-first* principle [2, 13]. It aims to reduce the tree size by selecting firstly most constrained variables, which can prune quickly some inconsistent branches. On the contrary, value ordering heuristics commonly use the *succeed-first* principle for selecting values that can belong to a solution so as to restrict backtracks. Some generic (dynamic) variable ordering heuristics consider the concept of degree of a variable, defined as the number of constraints involving it. For example:

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

- `dom/ddeg` [3]: it selects first the variable having the minimal ratio between domain size and degree. The degree can be computed dynamically by the number of constraints linking a given variable to uninstantiate variables;
- `dom/wdeg` [4]: it selects first the variable having the minimal ratio between domain size and weighted degree. A weight factor is associated to each constraint. When a dead-end occurs, the weight of inconsistent constraints is increased. For a given variable, its weighted degree (wdeg) is the sum of weights of constraints involving this variable and not yet instantiated variables.

The heuristic `dom/wdeg` was embedded into a `MAC` algorithm and has proved its efficiency on a large range of both random and real problems [4]. This method was improved by Grimes and Wallace [12] including restarts to the original method. The authors of `MAC` associated with `dom/wdeg` explain its efficiency by the fact that weights on constraints allow the search to be guided towards difficult parts of the problem and limit the redundancy during solving.

Another way to improve tree search methods was proposed with Last-Conflict (`LC`) analysis [23]. `LC` aims to backtrack on the variable having produced the last failure. For that purpose, this variable tends to be instantiated first whatever is the selection made by the ordering heuristics on variables. This analysis can be generalized to k last conflicts. The `LC` analysis has been proven to be efficient with classical ordering heuristics (such as `dom/ddeg`) but it is less interesting with heuristics using the weighted degree principle.

Limited Discrepancy Search (`LDS`) [14] proposes an alternative way to backtrack when a dead-end occurs. Since a good value ordering heuristic cannot avoid bad guesses (*i.e.,* choosing, for a given variable, a value that does not participate in any solution), `LDS` tackles this problem by gradually increasing the number of allowed discrepancies, where a *discrepancy* is associated with any decision point in a search tree when the choice goes against a value ordering heuristic. This method stops when a solution is found or when the maximum number of discrepancies is reached (in case of inconsistency).

Several variants of discrepancy-based methods were proposed. These methods differ by the manner discrepancies are applied: either first at the top of the search tree, or first at the end; and by the fact that redundancies are allowed or not during the search tree expansion. For instance, `LDS` applies discrepancies at the top first and has redundancy since a partial instantiation is visited more than once in successive iterations. *Improved LDS* (`ILDS`) [21] is a non-redundant variant applying discrepancy at the end first. *Depth-bounded Discrepancy Search* (`DDS`) [33] first favors discrepancies at the top of the search tree authorizing the discrepancies only in the

first levels of a given depth; it is non-redundant. Another variant is the method YIELDS [20], which is based on LDS and includes a mechanism to limit the exploration when the problem is inconsistent, even if the total number of discrepancies has not been used.

3 Weight-based heuristic for Constraint Satisfaction Problems

3.1 Variable weighting ordering heuristic

Let (X, D, C) be a binary CSP of n variables. $W_{var}(i)$ denotes the weight associated with each variable X_i. The variable weight vector W_{var} is the vector composed of weights of all variables of the problem: $W_{var} = [W_{var}(i)]_n$.

The proposed heuristic Wvar consists in associating a weight with each variable. This weight is increased whenever the corresponding variable is involved in a dead-end during search. The variable having the most important weight is selected for instantiation.

To illustrate this mechanism, we consider a CSP with three variables (X_0, X_1, X_2), all with domain {0,1,2,3,4}. The set of contraints C is represented by the following set of incompatible tuples: $\{(X_0,0),(X_1,4)\} \cup \{(X_0,0),(X_2,4)\} \cup \{(X_0,1),(X_1,4)\} \cup \{(X_0,1),(X_2,4)\} \cup \{(X_0,2),(X_1,4)\} \cup \{(X_0,2),(X_2,4)\} \cup \{(X_0,3),(X_1,4)\} \cup \{(X_0,3),(X_2,4)\} \cup \{(X_0,4),(X_2,2)\} \cup \{(X_0,4),(X_2,3)\} \cup \{(X_1,0),(X_2,0)\} \cup \{(X_1,0),(X_2,1)\} \cup \{(X_1,0),(X_2,2)\} \cup \{(X_1,0),(X_2,3)\} \cup \{(X_1,1),(X_2,0)\} \cup \{(X_1,1),(X_2,1)\} \cup \{(X_1,1),(X_2,2)\} \cup \{(X_1,1),(X_2,3)\} \cup \{(X_1,2),(X_2,0)\} \cup \{(X_1,2),(X_2,1)\} \cup \{(X_1,2),(X_2,2)\} \cup \{(X_1,2),(X_2,3)\} \cup \{(X_1,3),(X_2,0)\} \cup \{(X_1,3),(X_2,1)\} \cup \{(X_1,3),(X_2,2)\} \cup \{(X_1,3),(X_2,3)\}$.

In Figure 1, we represent the iterations of a discrepancy-based method using a weighting variable heuristic, for instance Wvar⊕Lexico, that is a heuristic based on vector W_{var} and on the lexicographical order to break the ties. Table 1 gives the value of the weights of each variable after each iteration. Initially, $W_{var}(i) = 0,\ \forall i \in \{0,1,2\}$. After iteration LDS(0) (*i.e.,* the LDS method at hand with 0 discrepancy), $W_{var}(2) = 1$ due to the dead-end on X_2. The method then restarts with 1 discrepancy and a new ordering heuristic based on Wvar. Atfer iteration LDS(1), $W_{var}(1) = 3$ due to three failures on X_1. During the iteration LDS(2), based on the new ordering between variables, the method gets a solution.

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

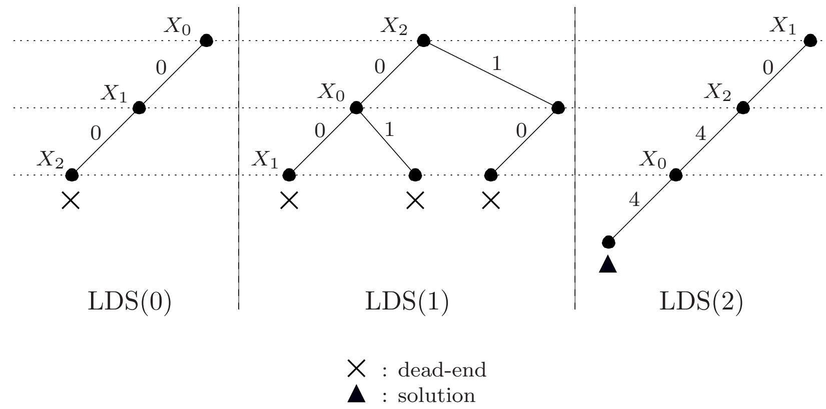


Figure 1: LDS search tree with the `Wvar` heuristic

Table 1: Variable weighting

W_{var}	after `LDS(0)`	after `LDS(1)`
$W_{var}(X_0)$	0	0
$W_{var}(X_1)$	0	+3
$W_{var}(X_2)$	+1	1

3.2 Integration in tree search methods

The heuristic presented previously can be grafted into tree search methods, such as discrepancy search or chronological backtracking. For discrepancy search and non-binary trees, two modes can be used to count discrepancies [10, 27]. First, the binary way: exploring the branch associated with the best value, according to a value ordering heuristic, involves no discrepancy, while exploring the remaining branches implies a single discrepancy. Second, the non-binary way: the values are ranked according to a value ordering heuristic such that the best value has rank 1; exploring the branch associated with a value of rank $k > 1$ leads to make $k-1$ discrepancies. In the following of this section, the heuristic is integrated into the `YIELDS` method proposed in [20] with a binary counting of discrepancies. In the `YIELDS` method, the `Wvar_YIELDS_Probe` algorithm (see Algorithm 1) is iterated either until a solution is found or until $CurrentMaxDiscr$ reached the maximum number of allowed discrepancies or until an inconsistency is detected.

Comparatively to the `YIELDS_Probe` algorithm, in `Wvar_YIELDS_Probe` the integration of the $Wvar$ heuristic needs the function `UpdateWeights` to update weights of variables when a dead-end occurs; and the selection of a successor of a node (function `Successors`) is based on the variable weights.

Algorithm 1: `Wvar_YIELDS_Probe`

```
Data: node, (X, D, C), W_var, CurrentMaxDiscr
Result: Sol
begin
    if Goal(node) then
        return Sol
    else
        suc ← Successors(node, (X, D, C), W_var)
        if Failure(suc) then
            UpdateWeights(node, W_var)
            return NIL
        else
            if CurrentMaxDiscr = 0 then
                return Wvar_YIELDS_Probe(First(suc), (X, D, C), W_var, 0)
            else
                Sol ← Wvar_YIELDS_Probe(Second(suc), (X, D, C), W_var,
                CurrentMaxDiscr − 1)
                if Sol ≠ NIL then
                    return Sol
                else
                    UpdateWeights(suc, W_var)
                return Wvar_YIELDS_Probe(First(suc), (X, D, C), W_var, CurrentMaxDiscr)
end
```

Based on the same principle, `LDS`, `YIELDS` with non-binary counting of discrepancies, and `CB` can be adapted for the `Wvar` heuristic.

3.3 Computational results

We propose to test the impact of the proposed variable weighting heuristic both in backtrack and in discrepancy search methods. The problems investigated in these experiments are car sequencing problems and random binary CSPs. The evaluation criteria are the number of expanded nodes and the CPU time. All algorithms were coded in C++. They were run on a Linux Fedora Core Duo 2.33 GHz PC having 4 Go of RAM.

3.3.1 Car sequencing problems

The car sequencing problem treats the placement of n cars in production on an assembly line that moves through various production units. Every production unit is responsible of installing

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

on cars potential options like airbags, sunroofs, radios, etc. Each unit has a limited capacity (constraint of the form r cars out of s, *i.e.,* the unit is able to produce at most r cars with a given option out of each sequence of s cars) and needs time to set up its associated option. Every car does not require all options but leads to a class of cars, which correspond to a specific list of options. A solution of the decision variant of the car sequencing problem (considered in this paper) is to find an assignment of cars to the slot that satisfies both the demand and the capacity constraints, which is NP-complete [11]. It was studied in many works [15, 17, 28, 30] and an optimization variant was considered in the 2005 ROADEF challenge [31]. To model this problem, we consider n class variables (one for each car), the domain of these variables is the set of car classes. We do not use any global constraint propagation among those proposed in [17] or in [28], but just `FC` propagation.

In these experiments, we consider the set of 70 satisfiable problems from the CSPLib [7] with a timeout of 200 seconds for each of them. We will compare chronological backtrack method *vs.* a discrepancy-based method following the various variable ordering heuristics proposed and using the same kind of constraint propagations (FC). Since the method YIELDS differs from the original method LDS by limiting the exploration for inconsistent problems (see Section 2), note that both methods are equivalent here because the instances under consideration are all satisfiable.

The first part of experiments consists in testing the impact of instantiation heuristics on `CB` and `LDS` with binary and non-binary counting modes. We then compare `CB` and each variant of `LDS` with two variable ordering heuristics: `Lexico`, which follows the lexicographical order, and `Wvar`$\oplus$`Lexico`,which chooses the variable X_i associated with the greatest weight $W_{var}(i)$ and, in case of ties, follows the lexicographical order. The value ordering is `MaxOpt`$\oplus$`Lexico` which selects value corresponding to a car which requires the greatest number of options (`MaxOpt`) and, to break the ties, uses the lexicographical order.

Table 2 presents for each method the number of solved problems out of 70 (#`Solved`), the average CPU time (`CPU`) in seconds and the average number of expanded nodes (`NEN`) to solve these problems. For each method, this table shows that the use of `Wvar`$\oplus$`Lexico` as variable ordering improves the number of solved problems. This improvement is very weak for `CB` and comes along with a high increase in the average CPU time (4 times in more) and in the average number of expanded nodes (2.8 times in more). However, for binary and non-binary `LDS`, this improvement is more important (increasing the number of solved problems from 43% for binary `LDS` up to 71% for non-binary `LDS`) and leads to a reduction of the average number of expanded nodes (decreasing about 75% with binary `LDS` and for 42% for non-binary `LDS`). The CPU

time increases very slightly with Wvar⊕Lexico in binary LDS (about 5%) and it decreases more importantly with Wvar⊕Lexico in non-binary LDS (about 19%). Globally, the LDS method with non-binary counting associated with the variable ordering Wvar⊕Lexico outperforms all the other methods.

Table 2: Performance of CB *vs.* LDS (binary and non-binary variants) for car-sequencing instances

	VarOrder	Lexico	Wvar⊕Lexico
CB	#Solved	36	37
	CPU (s)	3.08	12.54
	NEN	700164	1995906
Binary LDS	#Solved	41	59
	CPU (s)	9.87	10.42
	NEN	662301	165075
Non-binary LDS	#Solved	38	65
	CPU (s)	11.81	9.61
	NEN	541359	314350

In [17], a global constraint called regular is proposed and compared to the global sequencing constraint (GSC) available in Ilog Solver. For the same set of considered experiments (instances called *carseq10 ... carseq78* in [17]), 37 instances are solved using GSC and 39 using GSC combined with regular (with a timeout of 3600 seconds). In this paper, the ordering heuristics used are the best between dom and the slack heuristic proposed in [28]. With the proposed heuristic based on Wvar⊕Lexico, with just FC propagation, the best tree search method (Non-binary LDS) is clearly very well performing since it solved 65 instances.

3.3.2 Random binary CSPs

To obtain random binary CSPs, we used the model B generator developed by Frost *et al.* [9]. According to [35], we started from densities that permit to avoid flawed instances. We consider instances involving 30 and 40 variables having a uniform domain size of 25 and 20. The problem density (*i.e.,* the ratio of the number of constraints involved in the constraint graph over that of all possible constraints), denoted by p_1, varies. The constraint tightness (*i.e.,* the ratio of the number of disallowed tuples over that of all possible tuples), denoted by p_2, varies so that we obtain instances around the peak of complexity. The size of samples is 100 problem instances for each tuple (n, d, p_1, p_2) where n is the number of variables and d the maximal domain size.

The first part of experiments is devoted to testing the impact of two variable instantiation heuristics (dom/Wvar and dom/wdeg) on both CB and YIELDS methods using the same level

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

of constraint propagation (arc-consistency). The arc-consistency is obtained by the powerful algorithm proposed by Zhang [36]. In the following, `CB` with arc-consistency is named `MAC`, and `YIELDS` with arc-consistency is just denoted as `YIELDS`. For each method (`MAC`, `YIELDS` with binary and non-binary counting modes), we compute the standard deviation as follows: $100 * (\texttt{dom/wdeg} - \texttt{dom/Wvar})/\texttt{dom/wdeg}$. The value ordering heuristic is always `minconflicts` undergoing a succeed-first principle.

Figure 2 compares the results obtained by `MAC` using the two variable ordering heuristics `dom/Wvar` and `dom/wdeg`. The left part of the figure entitled by '`30-25-0.16`' stands for CSPs of 30 variables, 25 values, and a density p_1 of 0.16. The right part of the figure, entitled by '`40-20-0.8`' stands for CSPs with 40 variables, domain sizes of 20, and a density p_1 of 0.8. The X-axis corresponds to several values of p_2. The top of the figure gives the CPU time in milliseconds (`CPU`) for each method and the bottom of the figure gives the number of expanded nodes (`NEN`).

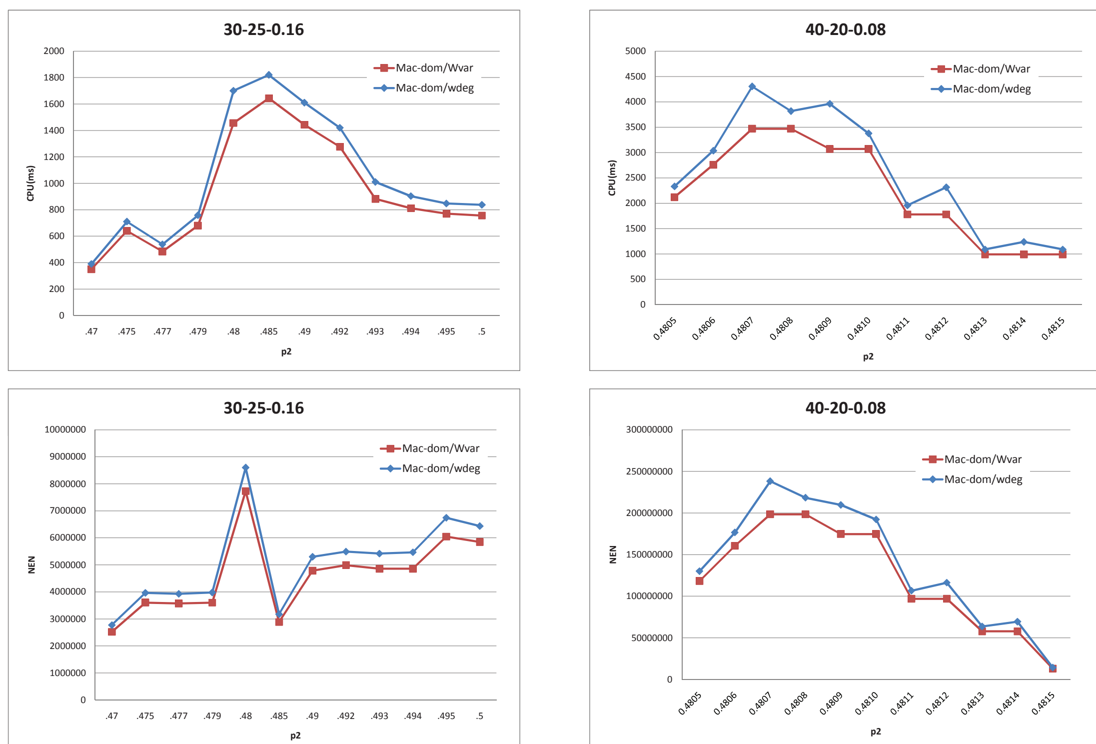


Figure 2: Random CSPs: Comparison of two weighting heuristics in `MAC`

This figure shows that `MAC` with the `dom/Wvar` variable ordering heuristic gives better results

comparatively with `dom/wdeg`. For instances belonging to set `40-20-0.8`, the CPU time is reduced by about 13.5% in average (from 9.1% to 23.1%) and the number of expanded nodes is reduced by about 11.3% (from 9.1% to 16.7%). For instances from the series `30-25-0.16`, the CPU time is reduced by about 10.6% (from 9.1% to 14.3%) and the number of expanded nodes is reduced by about 9.7% (from 9.1% to 11.2%). In the next figures, the method `MAC` with `dom/Wvar` is retained for comparison.

Figure 3 displays the results obtained by the method `YIELDS` with binary counting (denoted by `YIELDS-B`) for both variable ordering heuristics: `dom/Wvar` and `dom/wdeg`.

Figure 3: Random CSPs: Comparison of two weighting heuristics in binary `YIELDS`

The contribution of `dom/Wvar` towards `YIELDS-B` is less evident. Indeed, for the instances from `40-20-0.8`, the heuristic `dom/Wvar` causes reduction of 27.1% for the CPU time and of 57.9% for the number of expanded nodes. However, for the instances from `30-25-0.16`, NEN is still reduced (about 27.9%) but CPU is now increased (about 55.6%). Moreover, `YIELDS-B` with either `dom/Wvar` or `dom/wdeg` never improves `MAC` with `dom/Wvar` both on `CPU` and `NEN` for instances `40-20-0.8`. It is just profitable in `NEN` for instances from `30-25-0.16`.

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

Figure 4 reports the results obtained by the method `YIELDS` with non-binary counting (denoted by `YIELDS-NB`) for both variable ordering heuristics: `dom/Wvar`, and `dom/wdeg`, and compares these methods to `MAC` with `dom/Wvar`. `YIELDS-NB` with `dom/Wvar` clearly outperforms `YIELDS-NB` with `dom/wdeg`. In average, for instances from `40-20-0.8`, the CPU time is improved by about 46.4% and the NEN is improved by about 13.5%. For the instances from `30-25-0.16`, the CPU time is improved by about 34.1% and the NEN by about 21.1%.

Moreover, `YIELDS-NB` with `dom/Wvar` or `dom/wdeg` always improves `MAC` with `dom/Wvar` in both CPU time and NEN. For the proposed weighted heuristic `dom/Wvar`, we evaluate the average standard deviation between `MAC` and `YIELDS-NB`: $100 * (\texttt{MAC} - \texttt{YIELDS-NB})/\texttt{MAC}$. For CPU, the improvement of `YIELDS-NB` is around 80.2% and 78.9% for instances `40-20-0.8` and `30-25-0.16`, respectively; for NEN this improvement is of 50.9% and 77.1%.

Figure 4: Random CSPs: Comparison of two weighting heuristics in non-binary `YIELDS`

In the last part of experiments, three variants of `YIELDS` are compared, depending on the way for discrepancy counting: binary, non-binary, and also mixed [10]. For mixed counting we consider non-binary counting at the top and binary counting at the bottom, denoted by

YIELDS-NB-B, and the opposite, named YIELDS-B-NB. To mix the two kinds of counting, we use the depth for partitioning the search tree and change the counting modes. We test various partitioning depths of 1/5, 1/3, and 2/3 (measured from the top). In Figure 5, we compare the results obtained by YIELDS-B, YIELDS-NB, and a mixed counting mode with depth equals 1/5, denoted by YIELDS-NB-B-1/5.

Figure 5: Random CSPs: YIELDS with mixed counting

This figure shows that YIELDS-NB drastically improves YIELDS-B. In terms of average standard deviation, the improvement of YIELDS-NB is around 90.1% in CPU for instances 40-20-0.8 and around 78.1% for instances 30-25-0.16. Concerning NEN, this improvement is about 79.5% and 86.4% for instances 40-20-0.8 and 30-25-0.16, respectively. Furthermore, YIELDS-NB-B-1/5 again improves YIELDS-NB. The CPU time is improved by about 63.5% and 45.1% and NEN is improved by about 69.9% and 57.3% for instances 40-20-0.8 and 30-25-0.16, respectively.

Note that these results are not steady for other values of the depth, which remains open the right tuning for partitionning the search tree.

hal-00660715, version 1 - 17 Jan 2012

3.3.3 Discussion

Previous figures and tables show that for solving car-sequencing and random CSPs, non-binary variants of discrepancy-based methods are more efficient than binary variants or than MAC. Moreover, for random CSPs, `YIELDS-NB` is less sensitive, than the other approaches, to the variations of parameter p_2, *i.e.,* the constraint tightness. This tends to assume that `YIELDS-NB` is a robust method.

In complement to the above presented results, we have conducted experiments on the performance of the mixed counting mode to solve car-sequencing instances. We tested various partitioning depths but none improved the results obtained with the non-binary version of `YIELDS`. The interest of the mixed counting is thus still questionable for a discrepancy-based method.

Other weighting mechanisms were also tried. For instance, we have considered the integration of weighting values in both `MAC` and `YIELDS` methods. We tested three value ordering heuristics, each one being different from the other according to the mode to increment the weight of a value. However, the associated experiments did not produce valuable results.

4 Discrepancy and learning for jobshop problems with time-lags

This section addresses a combinatorial optimization problem: the jobshop scheduling problem with time-lags. A time-lag constraint is initially defined as a time-distance between the end of an operation and the start of another. This extension of classical shop scheduling problems (generally, they are already NP-hard in the strong sense) is of practical importance, as time-lag constraints appear frequently in industrial processes such as chemical plants, pharmaceutical, and food industry [26].

Few methods have been used to solve this type of problems. Wikum *et al.* [34] study single-machine problems with minimum and/or maximum distances between jobs. The authors state that some particular single-machine problems with time-lags are polynomially solvable, even if the general case is NP-hard. Brucker *et al.* [5] show that many scheduling problems (such as multi-processor tasks or multi-purpose machines) can be modeled as single-machine problems with time-lags and propose a branch-and-bound method. A local search approach can be found in [19]. A memetic algorithm is proposed in [6] and obtained good results on jobshop instances, especially for those with null minimum and maximum time-lags (named as "no-wait problems"). Branch-and-bound was proposed for single-machine problems with general time-lags [5]. The problem is also studied in [1] by integrating generalized resource constraint propagation and

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

branch-and-bound.

In this part, as in [1, 6], we only consider maximum time-lags between successive operations of the same job. In this case, a trivial schedule can be obtained by a greedy constructive algorithm. It consists in considering the jobs one after the other, and, for each job, all its operations are scheduled at their earliest start time. The first operation of the next job starts at the end of the partial schedule. However, this trivial schedule is of poor performance for makespan minimization. The proposition of methods for solving the jobshop scheduling problem with maximum time-lags is then relevant, even for just finding a non-trivial feasible solution.

4.1 Problem modeling and initial solution

In the jobshop problem, a set J of n jobs have to be scheduled on a set of m machines. Each job J_i corresponds to a linear sequence of n_i operations. Each operation must be processed on a unique non-preemptive machine and each machine can process only one operation at a time. Then, two operations that need the same machine cannot be processed at the same time. In the jobshop problem with time-lags (JSPTL) under study, time-lag constraints represent the distance between two operations belonging to the same job. The minimum distance (supposed to be positive or null) corresponds to the minimum time-lag and the maximum distance to the maximum time-lag. Maximum time-lags represent a specific difficulty since they can create many impossibilities for the operations insertion on the machines. A trivial schedule consisting in sequencing the entire jobs one after the other leads to poor makespan, and finding a non-trivial feasible solution remains a difficult issue. Problems containing only minimum time-lags are less difficult. In fact, minimum time-lags can be considered as a part of the previous operation. The difference is that the resource is available over the time-lag.

Solving the JSPTL consists in sequencing all of the operations on the machines such that the resource sharing constraints are satisfied and such that consecutive operations of each job respect the time-lag constraints. The objective is to find a schedule that minimizes the makespan ($\min C_{\max}$).

Assume that $i = 1, \ldots, n$ are job indexes and $j = 1, \ldots, n_i$ operation indexes for job i.

- (i, j) stands for the jth operation of job i,
- $m_{i,j}$ corresponds to the machine allocated to operation (i, j),
- $p_{i,j}$ is the duration of operation (i, j),
- $t_{i,j}$ represents the start time of operation (i, j),

. $TL^{\min}_{i,j,j+1}$ and $TL^{\max}_{i,j,j+1}$ correspond to the values of the minimum and maximum time lags, respectively, between operations (i, j) and $(i, j+1)$.

Constraints of the problem are mathematically formulated as follows:

$$C_{\max} \geq t_{i,j} + p_{i,j} \qquad \forall i = 1, \ldots, n\,, j = 1, \ldots, n_i, \tag{1}$$

$$t_{i,j+1} \geq t_{i,j} + p_{i,j} + TL^{\min}_{i,j,j+1} \qquad \forall i = 1, \ldots, n\,, j = 1, \ldots, n_{i-1}, \tag{2}$$

$$t_{i,j+1} \leq t_{i,j} + p_{i,j} + TL^{\max}_{i,j,j+1} \qquad \forall i = 1, \ldots, n\,, j = 1, \ldots, n_{i-1}, \tag{3}$$

$$t_{i,j} \geq t_{k,l} + p_{k,l} \text{ or } t_{k,l} \geq t_{i,j} + p_{i,j} \qquad \forall (i,j), (k,l)\ m_{i,j} = m_{k,l}, \tag{4}$$

$$t_{i,j} \geq 0 \qquad \forall i = 1, \ldots, n\,, j = 1, \ldots, n_i. \tag{5}$$

Constraints (1) establish the end times of the operations. Constraints (2) and (3) represent the temporal constraints between two consecutive operations of the same job including minimum and maximum time lags, respectively. Constraints (4) correspond to the resource sharing: two operations competing for the same machine cannot be processed at the same time and must then be sequenced.

In our modeling, the set of variables X correspond to job selection variables. The variable X_i is then the selection of the ith job to be scheduled on the machines. For each $i \in [1, \ldots, n]$, the domain D_i of each X_i variable is $\{J_1, \ldots, J_n\}$. Thus, the values of these variables must be all different. Once a job J_i is selected, the n_i operations of this job have to be scheduled on the machines by instantiating their start times from $t_{i,1}$ to t_{i,n_i} such that both resource sharing and time-lag constraints are satisfied.

To solve the JSPTL under study, we use the job insertion heuristic proposed by [1]. It consists in selecting a job J_i and in setting the start time of its first operation, $t_{i,1}$, to its earliest start time on the associated machine. Then, we go to the second operation and fix its start time $t_{i,2}$: if its earliest start time (for the required machine) does not match with the time-lag constraints associated with the first operation, we go back to the first operation and shift it to its next possibility. We proceed this way until we found a position where both operations are scheduled on the machines and the time-lag constraints are satisfied. We reiterate until all the operations of each job are scheduled. In the worst case, the jobs are scheduled in a single queue.

4.2 Weight-based heuristics for improving climbing discrepancy search

To solve problems with time-lags, we consider a variant of Climbing Discrepancy Search (CDS), a tree search method based on discrepancy devoted to optimization. CDS [25] starts from an initial solution proposed by a given heuristic and tries to improve it by increasing step by step

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

the number of discrepancies. It then builds a neighborhood around this initial solution. The leaves with a number of discrepancy equal to 1 are first explored, then those having a discrepancy number equal to 2, and so on. As soon as a leaf with an improved value of the objective function is found, the reference solution is updated, the number of discrepancy is reset to 0, and the process for exploring the neighborhood is restarted. To limit the search tree expansion, we fix a stop condition as a timeout on the CPU time.

We propose to integrate learning mechanisms based on weights in the CDS method. For the JSPTL, we then associate a weight W_i with each job J_i. During the search, if some operations of a given job J_i cannot be scheduled at their earliest start time on the associated machine, W_i is increased. In our problem, the weights $\{W_j\}_{j=1..n}$ (initially equal for all j) can be increased in three ways:

- Increment per operation (`Op`): The job weight is increased every time one of its operations cannot be inserted in some slack period and then must be postponed until the next slack period. The number of moves is cumulated to give the increase in the job weight.
- Increment per machine (`Mach`): The job weight is increased every time one of its operations is not inserted in the first slack period on one of its associated machine. We increment the weight at most once per machine (or operation). The maximum factor to get on a considered operation is equal to the number of machines (or operations).
- Unary increment per job (`Job`): The job weight is increased every time one or more operations of this job are not placed in their first slack periods on the associated machine. The weight is increased at most once for the same job.

In a given iteration of the CDS method, several branches are developed and a same job may have several weight increases. Thus, in addition of the manner to increase weights, we must choose a way to count them after a given iteration of the CDS method. We propose to consider either the sum of all weights obtained by the job during the incumbent iteration (denoted in the following by *Sum*), or the maximum of all its weights (denoted by *Max*). In the next iterations of CDS, the weights obtained are integrated in the instantiation heuristic to select the job to be scheduled.

Moreover, in our experiments we consider various discrepancy positions in the CDS method (see Section 2): we test to diverge alternatively, first at the top of the search tree, or first at its bottom. We try also to diverge only in a part of the tree, for example at the top limited by a given depth, and to visit the other part without discrepancies at all. We use the binary

hal-00660715, version 1 - 17 Jan 2012

discrepancy counting (it was experimentally proved better than the non-binary mode for the problem under study): the heuristic choice corresponds to 0 discrepancy, all the other choices correspond to one discrepancy.

Algorithm 2 summarizes the proposed improved CDS method principle. This algorithm is the same than CDS algorithm except in Line 6 where various parameter settings (in bold characters) offer several choices. *Div_pos* denotes the discrepancy position chosen to diverge first: either first at the top or first at the bottom. *W_Increment* denotes the case chosen to increment the discrepancies (`Op`, `Mach`, `Job`, or 0 when no weights are associated with the jobs). *W_integration* denotes the mode for weight integration after a given iteration (*Max* or *Sum*).

Algorithm 2: `Improved_CDS_Probe`

Data: X, D, C `% Variables, Domains of values, and Constraints`
Result: Sol
begin
    `%` k `is the discrepancy number` ;
1    $k \leftarrow 0$;
    `%` n `is the variable number` ;
2    $k_{max} \leftarrow n$;
    `%` S_{init} `is the initial solution` ;
3    $S_{init} \leftarrow initial_solution(X, D, C)$;
4   **while** $k \leq k_{max}$ **do**
5        $k \leftarrow k + 1$;
        `% Generate` k`-discrepancy branches from` S_{init} ;
6        $S'_{init} \leftarrow Generate(S_{init}, k, \textbf{Div_pos}, \textbf{W_increment}, \textbf{W_integration})$;
7   **if** $Best(S'_{init}, S_{init})$ **then**
            `% Update the initial solution` ;
8            $S_{init} \leftarrow S'_{init}$;
9            $k \leftarrow 0$;
10   **return** Sol ;
end

Example

We consider a jobshop with three jobs J_1, J_2, and J_3, and three machines m_1, m_2, and m_3. Every job has three operations. Table 3 gives the associated machine and duration for each operation of each job and the time-lag constraints between each pair of consecutive operations.

For the solving method CDS with weighted values, we consider, in this example, the lexicographical order for the job selection and, for the sake of simplicity, we use a non-binary counting

Table 3: Example of jobshop problem with time-lags

J_i	O_{i1}	time-lags (min, max)	O_{i2}	time-lags (min, max)	O_{i3}
J_1	$m_2, 3$	0, 1	$m_1, 5$	0, 0	$m_3, 4$
J_2	$m_1, 2$	0, 1	$m_2, 8$	0, 1	$m_3, 8$
J_3	$m_3, 9$	0, 0	$m_1, 5$	0, 0	$m_2, 5$

of discrepancies. The Gantt chart of Figure 6 depicts the solution obtained by the presented heuristic where the job selection is J_1, J_2, and then J_3. This solution has a makespan of 39.

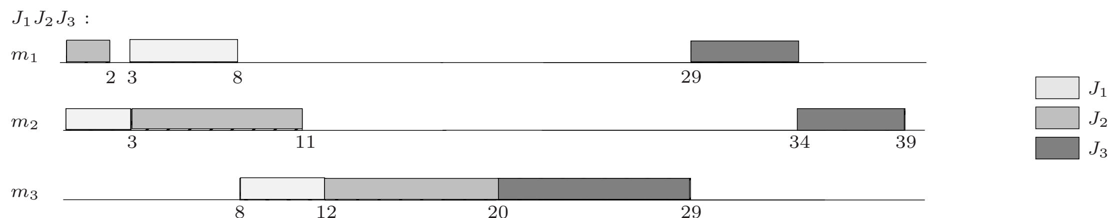


Figure 6: Gantt chart of the initial solution (0 discrepancy)

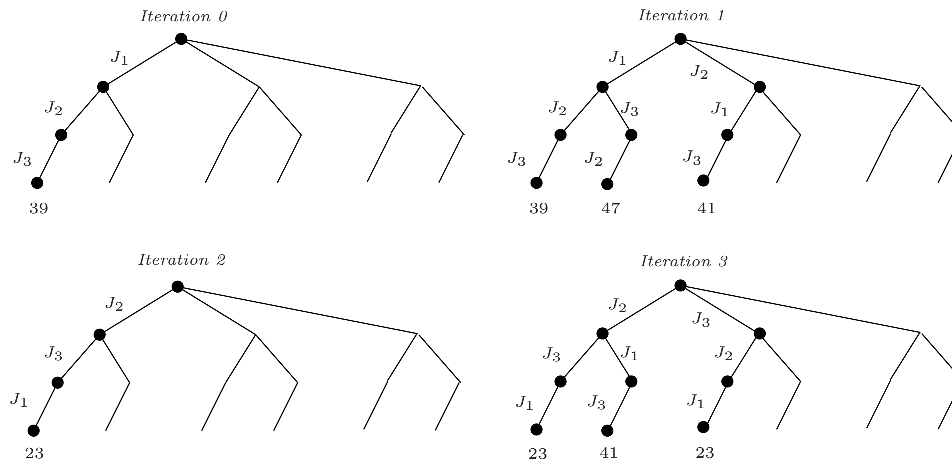


Figure 7: Iterations of the CDS method

Figure 7 presents several iterations of the proposed CDS method with weighted values. Based on the initial solution (iteration 0), the method develops 1-discrepancy solutions (iteration 1). The first one selects $J_1J_3J_2$ and leads to a makespan of 47, the second one selects $J_2J_1J_3$ with a makespan of 41. Figure 8 displays the Gantt charts of these two solutions.

Operations in dotted lines were shifted to respect both resource and time-lag constraints. Table 4 gives the new weights for these jobs and the end of iteration 1 for the three ways of

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

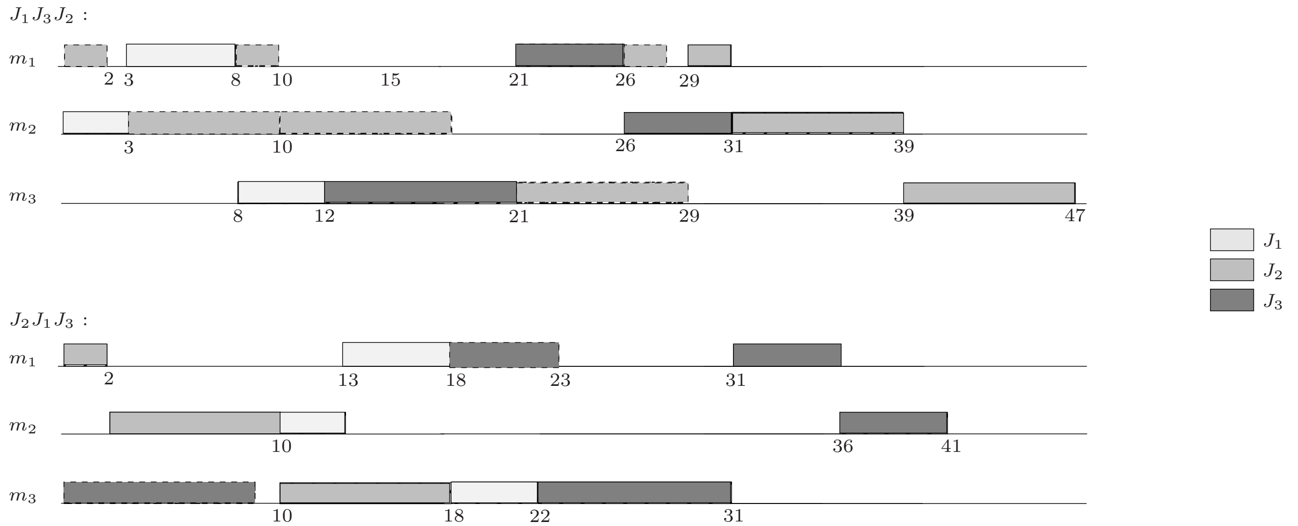


Figure 8: Gantt charts of 1-discrepancy solutions

Table 4: Weights of jobs after iteration 1 according to three counting modes

	W_1			W_2			W_3		
	Op	Mach	Job	Op	Mach	Job	Op	Mach	Job
$J_1J_2J_3$	-	-	-	-	-	-	-	-	-
$J_1J_3J_2$	-	-	-	6	3	1	-	-	-
$J_2J_1J_3$	-	-	-	-	-	-	2	2	1

counting. At iteration 0, there is no shift on operations and the job weights are not modified. At iteration 1, two sequences are then considered: $J_1J_3J_2$ and $J_2J_1J_3$ (see Figure 7). For the first sequence, *i.e.,* when attempting to place J_1 then J_3 then J_2, only operations of job J_2 are shifted to satisfy time-lag constraints (Figure 8). Therefore, job J_2 is the only job to increment its weight (column W_2 in Table 4). The same thing happens for job J_3 when searching for a solution with the order $J_2J_1J_3$ (column W_3 of the table). At the end of this iteration, J_2 has the highest priority, whatever the way for weight integration is (lexicographical order can be used to break the tie for mode `Job`), then J_3 and finally J_1, and there is no improvement of the initial solution with one discrepancy. Therefore, the proposed CDS method restarts with two discrepancies and selects jobs with the new defined order: $J_2J_3J_1$ (iteration 2). The first solution produced by this iteration has a makespan of 23 (see Figure 9). This leads to a lower makespan than the current best solution. The search is then interrupted and restarts with this solution as the new reference solution and search for 1-discrepancy solutions (iteration 3).

hal-00660715, version 1 - 17 Jan 2012

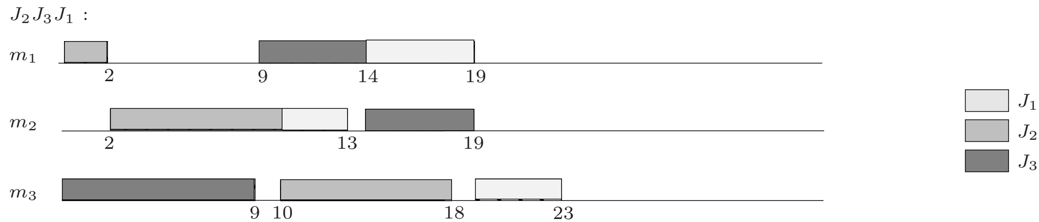


Figure 9: Gantt chart for the new reference solution of iteration 2

4.3 Computational results

Our experiments were conducted on instances presented in [6, 22]. These instances have been obtained from classical jobshop benchmarks where only maximum time-lags have been added (Lawrence's instances $\{laX\}_{X=1..20}$, and Fisher & Thompson's instances, $ft06$ and $ft10$). The maximum time-lags generated are associated with a coefficient α equal to 0, 0.25, 0.5, 1, 2, 3, 5, and 10 (note that, in the available instances, there are not all values of α for each instance). Maximum time-lags are calculated for each job J_i with these coefficients α and the durations $\{p_{ij}\}_{j=1..n_i}$ of the operations of the job.

To generate the initial solution, we tested different orders to sort the jobs. Considered orders are decreasing or increasing orders for job duration (`D`) or for job time-lags (`TL`); decreasing order of some combinations (`D⊕TL` and `D/TL`); lexicographical order (`Lexico`). The tests about the heuristic to generate the initial solution show that decreasing `D` gives the best results. Indeed in Table 5, one can observe the number of instances on which every heuristic is the best. It is also worth remarking that decreasing `D⊕TL` gives the same result (in bold). This is explained by the strong correlation between `D` and the value of the generated time-lags in these instances. For this first test, experiments concern a set of 126 instances ($ft06$, $la01..la20$) with only the values of α proposed by [6]. There is a timeout of 200 seconds for the various CDS methods.

Table 5: Comparison of heuristics to generate an initial solution (number of best solutions)

Order	`D`	`TL`	`D⊕TL`	`D/TL`	`Lexico`
Decreasing	**55**	51	**55**	29	20
Increasing	5	12	5	22	

For the second part of experiments, we tested many combinations of parameters: counting weights using cases `Op`, `Mach`, `Job`, or 0 if we do not use weights, and integrating weights after an iteration using *Max* or *Sum*. If weights are not used, the variant is named 0. A discrepancy may be applied at the top first or at the bottom first. In this experimental study, the case top-first

always leads to better solutions. Table 6 then compares several parameter settings for the CDS method, considering that top-first strategy is always selected to do discrepancies. It gives the number of times each combination of parameters obtained the best result for each given set of instances. We consider in these experiments the same set of 126 instances and the timeout of the CDS method is still limited to 200 seconds.

Table 6: Comparison of proposed CDS variants (number of best solutions)

Variant	ft06	la01-05	la06-10	la11-15	la16-20	Total
0	1	**21**	**19**	**19**	**16**	**76**
`Op`-*Sum*	*5*	6	3	4	6	24
`Op`-*Max*	2	4	1	5	6	18
`Mach`-*Sum*	3	6	*5*	5	*8*	*27*
`Mach`-*Max*	*5*	8	0	*7*	7	*27*
`Job`-*Sum*	3	8	0	6	7	24
`Job`-*Max*	3	*10*	1	6	7	*27*

Surprisingly, the CDS method with no weights (0) is very well performing. It gives the best results (in bold) in average over all sets of instances but also over all Lawrence's instances. Fisher & Thompson's instances are better solved with weighted variants of the method. For these weighted CDS methods, the counting modes based on `Mach` and `Job` outperform in average the `Op` mode. Actually, the method based on `Mach` generally beats that based on `Op` (values in italic). Thus, the variants 0, `Mach`-*Sum* and `Mach`-*Max* are retained in the sequel to go further in the experiments.

Then we tested a variant of the weighted CDS method with depth-bounded discrepancies (denoted by CDDS in [16]) to limit discrepancies at the top of the search tree. In Table 7 we give the results obtained for two depth limits (1/2 and 1/3) and the weighted mechanism is either `Mach`-*Sum* or `Mach`-*Max*. The comparisons are provided in terms of number of times each method obtained the best result and in terms of number of times each method improves the best-known value for the makespan (in parenthesis). For these experiments, we consider a set of 149 instances ($ft06$, $ft10$, $la01..la20$ with various values for α: 0, 0.25, 0.5, 1, 2, 3, 5, and 10) and the timeout of the CDS method is still limited to 200 seconds.

Table 7 shows that the benefit of weighted mechanisms is not steady for all variants. Indeed, including weights damages the results of CDS (columns 2 and 3 *vs.* column 1) while it improves the results for climbing depth-bounded discrepancy search (columns 2 and 3 of CDSS-1/2 and

hal-00660715, version 1 - 17 Jan 2012

Table 7: Comparison of variants with depth-bounded discrepancies (number of best solutions and number of benefits)

	CDS			CDDS-1/2			CDDS-1/3		
	0	Mach-*Sum*	Mach-*Max*	0	Mach-*Sum*	Mach-*Max*	0	Mach-*Sum*	Mach-*Max*
ft06-ft10	9 (0)	10 (0)	12 (0)	8 (0)	13 (0)	14 (0)	5 (0)	13 (0)	12 (0)
la01-05	23 (0)	10 (0)	15 (0)	7 (0)	20 (0)	23 (0)	7 (0	23 (0)	22 (0)
la06-10	22 (0)	9 (0)	7 (0)	16 (1)	11 (0)	15 (0)	16 (1)	11 (0)	15 (0)
la11-15	13 (11)	18 (10)	20 (10)	9 (9)	24 (12)	25 (12)	9 (9)	24 (12)	25 (12)
la16-20	23 (2)	10 (1)	9 (0)	12 (2)	19 (1)	16 (0)	12 (2)	19 (1)	16 (0)
Total	90 (13)	57 (11)	63 (10)	52 (12)	67 (13)	93 (12)	49 (12)	90 (13)	90 (12)

CDDS-1/3 *vs.* their respective column 1). Table 7 also illustrates that the CDS method with weighted mechanisms is more efficient when the depth in the tree for doing discrepancies is bounded (columns 2 and 3 of CDDS-1/2 and CDDS-1/3 *vs.* columns 2 and 3 of CDS). Conversely, CDS without weights does not take benefit from this bound on discrepancies (column 1 of all variants). In terms of deviation on the makespan (not presented in the table), the results are quite similar since it is about 1.16 for CDS without weights and no limit on discrepancies, about 1.19 for CDS without weights and when discrepancies are bounded, and about 1.17 for CDS with weights and whatever the limit on discrepancies is.

Considering that CDDS with weighted mechanisms obtains the overall greatest number of best solutions and almost the greatest number of improvements (93(12) for CDSS-1/2 and Mach-*Max*), we can conclude that using a weight-based heuristics is a promising approach to solve the JSPTL.

In the last part of experiments, we compare for all the 149 instances the proposed methods with the best-known results provided either by the memetic algorithm of [6] or by ILOG-Scheduler. For the 17 instances referred in Table 8 (instance name associated with its maximum time-lag), our methods found some improvements of the best-known results. In column 'BK', ILOG-Scheduler provides the best results for all the instances except for the two last no-wait instances ('la18 - 0' and 'la19 - 0') where the memetic algorithm is the best approach. On these instances, the CDDS methods with a depth limit of 1/2 or 1/3 obtain the same results; thus, we only report one on these results. In bold, we note the best obtained value on the makespan for a given instance over all the methods; the makespan is in italic if it corresponds to an improvement of the previous best-known results.

hal-00660715, version 1 - 17 Jan 2012

Table 8: Improved results on JSPTL benchmarks (makespan)

Instance	CDS 0	CDS `Mach`-*Sum*	CDS `Mach`-*Max*	CDDS 0	CDDS `Mach`-*Sum*	CDDS `Mach`-*Max*	BK
la06 - 0.25	1456	1496	1469	**1413**	1496	1469	1435
la11 - 0.25	**1861**	*1965*	*1965*	*1965*	*1965*	*1965*	2058
la11 - 0.5	**1874**	**1874**	**1874**	**1874**	**1874**	**1874**	1945
la12 - 0.25	*1682*	*1671*	**1656**	*1682*	*1682*	*1682*	1710
la12 - 0.5	1664	1605	1605	1775	**1398**	**1398**	1602
la12 - 1	1578	1558	1558	1586	**1397**	**1397**	1411
la13 - 0.25	*1897*	**1892**	**1892**	2052	**1892**	**1892**	1906
la13 - 0.5	**1787**	1808	1808	1888	1808	1808	1804
la14 - 0.25	**1823**	*2042*	*2042*	*2042*	*2042*	*2042*	2143
la14 - 0.5	*1964*	*1953*	*1953*	**1945**	*1953*	*1953*	2067
la14 - 1	*1772*	**1762**	**1762**	*1864*	**1762**	**1762**	1976
la14 - 3	*1576*	**1542**	**1542**	*1594*	**1542**	**1542**	1695
la15 - 0.25	*2048*	**2043**	**2043**	*2064*	**2043**	**2043**	2371
la15 - 0.5	*2118*	**1910**	**1910**	*2062*	*1922*	*1922*	2217
la17 - 0.25	**1410**	*1427*	1460	*1449*	*1427*	1460	1455
la18 - 0	**1733**	1973	1973	1823	1973	1973	1790
la19 - 0	1916	1965	1965	**1830**	1965	1965	1831

Table 8 illustrates that discrepancy search (CDS and CDDS methods) improves the results for some instances with tight time-lags (α generally lower than 1), that are hard instances. Focusing on the instances improved by the weighted mechanisms, it is interesting to remark that the way for weight integration (*Max* or *Sum*) has practically no impact (there is little difference only for 'la12 - 0.25' and 'la17 - 0.25' instances). Considering all the variants, the results are very tight; the global improvement from BK is of about 6.3%. Furthermore, as previously discussed, the results also show the interest of combining both weighted mechanisms and bounds on discrepancies, since CDDS based on the counting mode `Mach` participates in improving two of the instances ('la12 - 0.5' and 'la12 - 1'), which had not been improved by other methods.

Finally, it is worth noticeable that our discrepancy-based search methods (without weights, however) improves two no-wait instances for which the memetic algorithm proposed in [6] is generally the best performing.

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

5 Conclusion and further works

This paper presents various weighting mechanisms to improve instantiation heuristics, backtrack, and discrepancy search. These mechanisms are associated with different parameters to solve constraint satisfaction and combinatorial optimization problems.

In the satisfaction context, an experimental study was carried out on numerous binary CSPs and car sequencing benchmarks. The results showed that the proposed heuristic based on variable weighting is efficient for the studied tree search methods (based on chronological backtracking or on discrepancies). Furthermore, this study showed that variable ordering heuristics using weights on variables are the best suited heuristics when combined with discrepancy search; especially with non-binary counting mode. Finally, non-binary discrepancy search outperforms others methods (binary discrepancy search or chronological backtracking) on these experiments.

The second part of this work addresses the jobshop scheduling problem with time-lags. To solve it, a variant of Climbing Discrepancy Search (CDS) using weighting mechanisms is proposed. We studied various parameter settings for the proposed method, such as discrepancy positions, heuristics to generate the initial solution, and learning mechanisms based on weights associated with jobs. The proposed variants were tested on known benchmarks from the literature. Tests show that the proposed methods help to improve the makespan for some jobshop instances and that the combination of weights and bounds for discrepancies is promising.

A natural extension of this work is to consider other weighting mechanisms for both variable and value ordering to better understand the impact of weights for both satisfaction and optimization problems. Another extension should be to realize a more substantial computational experience. Hence, we would aim to increase the size and the number of considered instances for car-sequencing, random CSPs, or jobshop with time-lags. Other objectives are to study the impact of the variable weighting principle on other known heuristics and to study the impact counting modes for satisfaction or optimization problems. Some work is also to be done for increasing the efficiency of CDS with weights for jobshop with time-lags. In particular, the interaction between heuristic and weights will be studied. Finally, the introduction of extended resource constraint propagation rules taking account of time-lag constraints and the integration of upper bounds during the search could greatly improve the efficiency.

References

[1] C. Artigues, M.-J. Huguet, and P. Lopez. Generalized disjunctive constraint propagation for solving the job shop problem with time lags. *Engineering Applications of Artificial Intelligence*, 24(2):220–231, 2011.

[2] J. C. Beck, P. Prosser, and R. Wallace. Variable ordering heuristics show promise. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, pages 711–715, Toronto, Canada, October 2004.

[3] C. Bessière and J.-C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 61–75, Cambridge, Massachusetts, USA, August 1996.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, Valencia, Spain, August 2004.

[5] P. Brucker, T. Hilbig, and J. Hurink. A branch and bound algorithm for a single machine scheduling with positive and negative time-lags. *Discrete Applied Mathematics*, 94:77–99, 1999.

[6] A. Caumond, P. Lacomme, and N. Tchernev. A memetic algorithm for the job-shop with time-lags. *Computers and Operations Research*, 35:2331–2356, 2008.

[7] CSPLib. `http://csplib.org`.

[8] R. Dechter. *Constraint Processing*. Morgan Kaufmann, San Francisco, 2003.

[9] D. H. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random uniform CSP generators, 1996. `http://www.lirmm.fr/~bessiere/generator.html`.

[10] B. Gacias, C. Artigues, and P. Lopez. Parallel machine scheduling with precedence constraints and setup times. *Computers and Operations Research*, 37(12):2141–2151, 2010.

[11] I. P. Gent. Two results on car-sequencing problems. Research report 02-1998, APES, University of Strathclyde, UK, 1998.

[12] D. Grimes and R. J. Wallace. Learning from failure in constraint satisfaction search. AAAI Workshop on Learning for Search, Boston, Massachusetts, USA, July 2006.

hal-00660715, version 1 - 17 Jan 2012

hal-00660715, version 1 - 17 Jan 2012

[13] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[14] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 607–615, Montréal, Québec, Canada, August 1995.

[15] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.

[16] A. Ben Hmida, M. Haouari, M.-J. Huguet, and P. Lopez. Discrepancy search for the flexible job shop problem. *Computers and Operations Research*, 37(12):2192–2201, 2010.

[17] W. J. Van Hoeve, G. Pesant, L.-M.Rousseau, and A. Sabharwal. New filtering algorithms for combinations of among constraints. *Constraints*, 14(2):273–292, 2009.

[18] J. Hooker. *Logic-based Methods for Optimization: Combining Optimization and Constraint Satisfaction.* Wiley, New York, 2000.

[19] J. Hurink and J. Keuchel. Local search algorithms for a single-machine scheduling problem with positive and negative time-lags. *Discrete Applied Mathematics*, 112:179–197, 2001.

[20] W. Karoui, M.-J. Huguet, P. Lopez, and W. Naanaa. YIELDS: A yet improved limited discrepancy search for CSPs. In *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07), LNCS 4510, Springer*, pages 99–111, Brussels, Belgium, May 2007.

[21] R. E. Korf. Improved limited discrepancy search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96) and the 8th Innovative Applications of Artificial Intelligence Conference (IAAI'96)*, pages 286–291, Portland, Oregon, USA, August 1996.

[22] P. Lacomme. `http://www.isima.fr/~lacomme/Job_Shop_TL.html`.

[23] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Last conflict based reasoning. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 133–137, Trento, Italy, August 2006.

[24] C. Lecoutre, L. Sais, and J. Vion. Using SAT encodings to derive CSP value ordering heuristics. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:69–186, 2007.

hal-00660715, version 1 - 17 Jan 2012

[25] M. Milano and A. Roli. On the relation between complete and incomplete search: An informal discussion. In *Proceedings of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'02)*, pages 237–250, Le Croisic, France, March 2002.

[26] K. Neumann, C. Schwindt, and J. Zimmermann. *Project Scheduling with Time Windows and Scarce Resources.* Springer, 2002.

[27] N. Prcovic and B. Neveu. Ensuring a relevant visiting order of the leaf nodes during a tree search. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99), LNCS 1713, Springer*, pages 361–374, Alexandria, Virginia, USA, October 1999.

[28] J.-C. Régin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 32–46, 1997.

[29] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 2nd Workshop on Principles and Practices of Constraint Programming (PPCP'94), LNCS 874, Springer*, pages 10–20, Rosario, Orcas Island, Washington, USA, May 1994.

[30] B. Smith. Succeed-first or fail-first: A case study in variable and value ordering heuristics. In *Proceedings of the 3rd Conference on the Practical Applications of Constraint Technology (PACT'97)*, pages 321–330, London, UK, April 1997.

[31] C. Solnon, V. D. Cung, A. Nguyen, and C. Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'05 challenge problem. *European Journal of Operational Research*, 191(3):912–927, 2008.

[32] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press Ltd, London, 1993.

[33] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, volume 2, pages 1388–1395, Nagoya, Japan, August 1997.

[34] E. D. Wikum, D. C. Llewellyn, and G. L. Nemhauser. One-machine generalized precedence constrained scheduling problems. *Operations Research Letters*, 16(2):87–99, 1994.

[35] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171:514–534, 2007.

[36] Y. Zhang and R. H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 316–321, Seattle, Washington, USA, August 2001.

hal-00660715, version 1 - 17 Jan 2012