Conflict Ordering Search for Scheduling Problems

Steven Gay[1], Renaud Hartert[1], Christophe Lecoutre[2], Pierre Schaus[1]

[1] UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium, {firstname.lastname}@uclouvain.be

[2] CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France
lecoutre@cril.fr

Abstract. We introduce a new generic scheme to guide backtrack search, called Conflict Ordering Search (COS), that reorders variables on the basis of conflicts that happen during search. Similarly to generalized Last Conflict (LC), our approach remembers the last variables on which search decisions failed. Importantly, the initial ordering behind COS is given by a specified variable ordering heuristic, but contrary to LC, once consumed, this first ordering is forgotten, which makes COS conflict-driven. Our preliminary experiments show that COS – although simple to implement and parameter-free – is competitive with specialized searches on scheduling problems. We also show that our approach fits well within a restart framework, and can be enhanced with a value ordering heuristic that selects in priority the last assigned values.

1 Introduction

Backtracking search is a central complete algorithm used to solve combinatorial constrained problems. Unfortunately, it suffers from thrashing – repeatedly exploring the same fruitless subtrees – during search. Restarts, adaptive heuristics, and strong consistency algorithms are typical Constraint Programming (CP) techniques used to cope with thrashing.

Last Conflicts (LC) [9] has been shown to be highly profitable to complete search algorithms, both in constraint satisfaction and in automated artificial intelligence planning. The principle behind LC is to select in priority the last conflicting variables as long as they cannot be instantiated without leading to a failure. Interestingly enough, last conflict search can be combined with any underlying variable ordering heuristic. In *normal mode*, the underlying heuristic selects the variables to branch on, whereas in *conflict mode*, variables are directly selected in a conflict set built by last conflict.

While last conflict uses conflicts to repair the search heuristic, we show in this paper that conflicts can also be used to drive the search process by progressively replacing the initial variable heuristic. Basically, the idea behind our approach – namely, Conflict Ordering Search – is to reorder variables according to the most recent conflict they were involved in. Our experiments highlight that this simple reordering scheme, while being generic, can outperform domain specific heuristics for scheduling problems.

2 Related Works

We start by providing a quick overview of general-purpose search heuristics and schemes since our approach is definitively one of them. The simple variable ordering heuristic `dom` [5] – which selects variables by their domain size – has long been considered as the most robust backtrack search heuristic. However, a decade ago, modern *adaptive* heuristics were introduced. Such heuristics take into account information related to the part of the search space already explored. The two first proposed generic adaptive heuristics are `impact` [16] and `wdeg` [1]. The former relies on a measure of the effect of any assignment, and the latter associates a counter with each constraint (and indirectly, with each variable) indicating how many times any constraint led to a domain wipe-out. Counting-based heuristics [14] and activity-based search [11] are two recently introduced additional adaptive techniques to guide the search process.

Interestingly, Last Conflict (LC) [9] is a search mechanism that can be applied on top of any variable ordering heuristic. Precisely, the generalized form LC(k) works by recording and assigning first the k variables involved in the k last decisions that provoked a failure after propagation. The underlying search heuristic is used when all the last k conflicting variables have been assigned. While the ability of relying on an underlying search heuristic is a strong point of LC, setting the parameter k can be problematic, as we shall see later. The related scheme introduced in this paper goes further and orders permanently all conflicting variables using the time of their last conflicts, eventually becoming independent of the helper variable ordering heuristic.

3 Guiding Search by Timestamping Conflicts

We first introduce basic concepts. Then, we introduce Conflict Ordering Search and highlight its benefits within a context of restarts. We conclude this section by discussing the differences between LC(k) and COS.

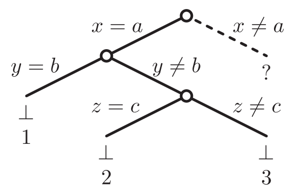
3.1 Background

CSP. A Constraint Satisfaction Problem (CSP) P is a pair $(\mathcal{X},\mathcal{C})$, where $\mathcal{X}$ is a finite set of variables and $\mathcal{C}$ is a finite set of constraints. Each variable $x \in \mathcal{X}$ has a domain $dom(x)$ that contains the allowed values for x. A valuation on a subset $X \subseteq \mathcal{X}$ of variables maps each variable $x \in X$ with a value in $dom(x)$. Each constraint $c \in \mathcal{C}$ has a scope $scp(c) \subseteq \mathcal{X}$, and is semantically defined by a set of allowed valuations on $scp(c)$; the valuations that satisfy c. A valuation on $\mathcal{X}$ is a solution of P iff it satisfies each constraint of P. A CSP is satisfiable iff it admits at least one solution.

Tree-Search. One can solve CSPs by using backtrack search, a complete depth-first exploration of the search space, with backtracking when a dead-end occurs. At each search node, a filtering process ϕ can be performed on domains by soliciting propagators associated with constraints. A CSP is in failure, denoted $\perp$, when unsatisfiability is detected by ϕ. A branching heuristic is a function that maps a non-failed CSP to an ordered sequence of constraints, called *decisions*. In this work, we only consider binary variable-based branching heuristics, i.e., heuristics that always generate sequences of decisions of the form $\langle x \in D, x \notin D \rangle$, where x is a variable of $\mathcal{X}$ and D a strict subset of $dom(x)$. A search tree is the structure explored by backtrack search through its filtering capability and its branching heuristic. A failed node in the search tree is a node where unsatisfiability has been detected by ϕ.

3.2 Conflict Ordering

Considering a variable-based branching heuristic, we can associate a failed search node with the variable involved in the decision leading to it. This allows us to timestamp variables with the number of the last conflict they caused (see Fig. 1). The basic idea behind Conflict Ordering Search is to leverage this timestamping mechanism to reorder the variables during search.



variable	x	y	z
stamp	0	1	3

Fig. 1. Conflict numbering and timestamps associated with each variable. Variables are stamped with the number of their latest conflict (or 0 by default).

Algorithm 1 describes Conflict Ordering Search. For simplicity, we only consider classical binary branching with decisions of the form $x \leq v$ and $x > v$. We use an integer `nConflicts` to count the number of conflicts and a reference `lastVar` to the last variable involved in a decision (initially *null* at the root of the search-tree). We also consider a one-dimensional array `stamps` that associates with each variable $x \in \mathcal{X}$ the last time variable x was involved in a conflict. They are all initialized to 0. We suppose that ϕ corresponds to a domain filtering consistency, which is at least as strong as the partial form of arc consistency ensured by the forward checking algorithm [5].

If the resulting CSP at line 1 is trivially inconsistent ($\perp$), false is returned (line 6). If the failure is due to a previous decision (line 3), the number of conflicts is incremented and the conflicting variable timestamped with this number (lines 4 and 5). Otherwise, *COS* returns true if a solution has been found, i.e., the domain of each variable in $\mathcal{X}$ is a singleton (lines 7 and 8). The selection of the next variable to branch on is performed between lines 9 and 13. Here, the timestamps are used to select the unbound variable involved in the latest conflict. If no unbound variable ever conflicted, the search falls back to the bootstrapping heuristic *varHeuristic*. When a new value has been selected by the heuristic *valHeuristic*$[x]$, we recursively call *COS*. One can observe that the complexity of selecting a variable is linear in time and space, hence scaling well.[3]

Algorithm 1: $COS(P = (\mathcal{X}, \mathcal{C})$: CSP)

```
   Output: true iff $P$ is satisfiable
 1 $P \leftarrow \phi(P)$
 2 if $P = \bot$ then
 3     if lastVar $\neq$ null then
 4         nConflicts $\leftarrow$ nConflicts $+ 1$
 5         stamps[lastVar] $\leftarrow$ nConflicts
 6     return false
 7 if $\forall x \in \mathcal{X}, |dom(x)| = 1$ then
 8     return true
 9 failed $\leftarrow \{x \in \mathcal{X}$ : stamps$[x] > 0 \wedge |dom(x)| > 1\}$
10 if failed $= \emptyset$ then
11     lastVar $\leftarrow$ $varHeuristic.select()$
12 else
13     lastVar $\leftarrow$ $\text{argmax}_{x \in \texttt{failed}}\{\texttt{stamps}[x]\}$
14 $v \leftarrow valHeuristic$[lastVar]$.select()$
15 return $COS(P_{|\texttt{lastVar} \leq v}) \vee COS(P_{|\texttt{lastVar} > v})$
```

Example 1. Let us consider a toy CSP with n "white" variables, and m "black" variables. White variables have a binary domain while black variables have $\{1, 2, \ldots, m-1\}$ as domain. We also add a binary difference constraint on each pair of black variables (thus making the CSP unsatisfiable), but no constraint at all on the white variables. Let us also assume a variable ordering heuristic that selects the white variables first, then the black variables. Hence, proving unsatisfiability using this heuristic requires to prove the "black conflict" for the 2^n valuations of the white variables (see left part of Fig. 2). Using COS on top of this heuristic allows one to detect unsatisfiability quickly. Indeed, the $m-2$ first conflicting black variables will be prioritized as a white variable cannot be involved in a conflict. The number of times the "black conflict" as to be proven thus becomes linear in the number of white variables n (see right part of Fig. 2).

COSPhase: a variant. Because it is known that remembering last assigned values for later priority uses can be worthwhile (see for example phase saving [15] in SAT), we propose such a variant for COS. So, when a positive decision $x \leq v$ succeeds, we record its value v. Then, when branching is performed on a

[3] The time complexity could be improved if an ordered linked-list is used instead of the array `stamps`.

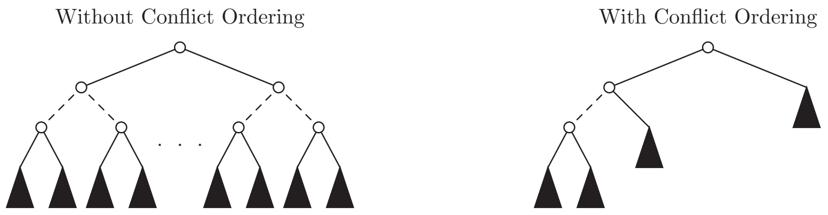


Fig. 2. Conflict Ordering Search reorders the variables to reduce the number of times the inconsistent black subtree has to be explored.

timestamped variable x, we exploit the associated recorded value v. If v is still in the domain of x, we use interval splitting on it, i.e., we branch with decisions $x \leq v$ and $x > v$, otherwise the value heuristic is solicited. Observe that this mechanism follows the first-fail/best-first principle.

3.3 Restarts and Timestamp Ordering

Depth-first search is far from always being the most efficient way to solve a CSP. In some cases, it may suffer from heavy-tailed distributions [4]. While restarting the search with a randomized heuristic is a typical way to avoid the worst parts of a long-tailed curve, *nogood learning* mitigates the effect of cutting a search process short by remembering parts of the search space already explored [8].[4]

In the context of COS, we observe that our approach not only remembers the sources of conflicts but also produce a variable ordering that yields a behavior similar to randomization. We thus propose to use no additional randomization when restarting, only using the natural randomizing effect of conflict ordering instead. The rationale is that while conflict ordering is good at finding a set of conflicting variables in a given context – i.e., a sequence of previous decisions – restarting with conflict ordering has the effect of trying the latest conflict set in other contexts.

Example 2. Let us consider the toy CSP described in Example 1. The time required to prove unfeasibility could be drastically reduced if a restart occurs after having explored the inconsistent black subtree at least once. Indeed, in this context, the second restart will directly explore the black search tree without even considering the white variables that have been "disculpated".

3.4 Differences with Last Conflict Search

Although similar, we show that COS and LC are rather different. Indeed, LC relies on a parameter k that corresponds to the maximum size of the conflict sets that can be captured by the search process. The value of k is of importance as setting k too low may not allow LC to capture the encountered conflict sets.

[4] Similar frameworks are typically used by SAT solvers [12].

For instance, LC(k) cannot capture the conflict set in Example 1 if k is lower than $m-2$. COS, however, does not require any parameter and is able to handle conflict sets of any size. While setting the parameter k to the number of decision variables may solve the problem, LC still suffers from resets of its conflict set that occur each time the conflicting variables have been successfully assigned. Conversely, COS does not forget conflicting variables and progressively reorders those variables to give priority to the recently conflicting ones. This is particularly important with restarts as LC is not designed to focus on conflict sets in such contexts (see Example 2).

4 Experiments

We have tested our approach on RCPSP (Resource-Constrained Project Scheduling Project) instances from PSPLIB [6]. We used a computer equipped with a i7-3615QM processor running at 2.30GHz. The problem has been modeled in the open-source solver OscaR [13], using precedence and cumulative constraints. Precedences are simple binary precedence constraints, and cumulative constraints use the Time-Tabling propagator presented in [2]. Both overload checking [19] or time-table edge-finding [17] were tested but energy-based reasoning does not help much on PSPLIB instances, whose optimal solutions typically waste capacity. Adding TTDR [3] helps even with learning searches, but it makes the experiments harder to reproduce, thus we chose to not use it.

4.1 Branch-and-bound

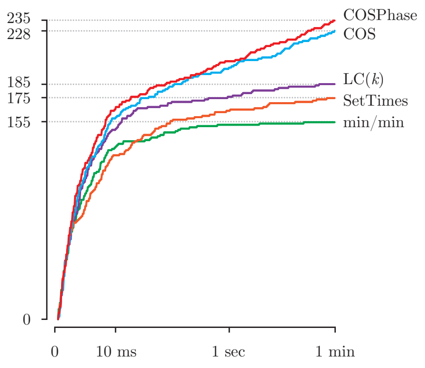
The goal of this first experiment is to find an optimal solution using a pure branch-and-bound search. We compare five search solving methods. The first is a simple min/min scheme, which selects the variable with the smallest minimal value and chooses the smallest value (for assignment). The second one is the scheduling-specialized SetTimes heuristic with dominances [7], which is a min/min scheme that simply postpones assignments (instead of making value refutations) when branching at right, fails when a postponed task can no longer be woken up, and assigns the tasks that are not postponed and cannot be disturbed by other tasks. Finally, the last three heuristics correspond to conflict-based reasoning searches, namely, LC(k) for the best value of k, our main contribution COS, and COSPhase based on the variant presented in Section 15. All these have a min/min helper heuristic.

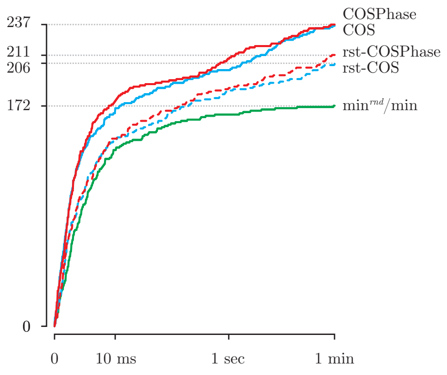
We have observed that the ranking of these five search methods is the same on the four RCPSP benchmarks (J30, J60, J90, J120) from PSPLIB. Results are represented on the left part of Fig. 3 where the y-axis is the cumulated number of solved instances and the x-axis is the CPU time. SetTimes is clearly an improvement on min/min, as it finishes ahead and seems to continue its course on a better slope than min/min. In turn, the well-parameterized LC (we empirically chose the best possible value for k) fares better than SetTimes. Finally, COS allows us to close a higher number of instances, and the variant COSPhase improves COS even further.

4.2 Branch-and-bound with Restarts

As illustrated in Example 2, keeping the conflict ordering between restarts could drastically reduce search efforts. The aim of this second experiment is to compare the performance of COS if the ordering is kept between restarts or not. Experimental settings are the same as before except that we use restarts and nogood recording has explained in [10]. The first iteration is limited to 100 failures and increases by a 1.15 factor. We compared the performance of COS and COSPhase with and without reset (we add "rst-" as prefix for the resetting version). All searches rely on $\min^{rnd}/\min$ – a randomized version of min/min that breaks ties randomly – as helper heuristic.

Results are presented in the right part of Fig. 3. First, we observe that restarts do have a positive effect as $\min^{rnd}/\min$ obtains better results than min/min in the previous experiment. Next, we see that resetting the conflict order has a bad effect on the search process. Indeed, these variant obtain worse results than in the pure branch-and-bound framework. This highlights that using conflict-based search as a full heuristic can yield much better results than using it as a repairing patch. Finally, phase recording does not seem to help anymore.





Search	**#closed**	**Σ Makespans**
COSPhase	235	77789
COS	228	77952
LC(k)	185	79010
SetTimes	175	80970
min/min	155	80038

Search	**#closed**	**Σ Makespans**
COS	237	77770
COSPhase	236	77721
rst-COSPhase	211	78301
rst-COS	206	78476
$\min^{rnd}/\min$	172	79510

Fig. 3. On the left, results obtained for pure branch-and-bound, and on the right, results obtained with branch-and-bound with restarts. Graphs at the top show the cumulated number of solved RCPSP instances from PSPLIB120. The tables at the bottom compare branching heuristics at the end of the 60s timeout, giving the number of closed instances and the sum of makespans.

4.3 Destructive lower bounds.

We performed similar experiments for destructive lower bounds. We added the Time-Table Edge-Finding propagator presented in [17] since it has a large impact in this case. The results are similar. We also compared COS to the recently introduced Failure Directed Search [18] by implementing it directly in CP Optimizer. Unfortunately our COS implementation in CP Optimizer was not able to obtain results competitive with FDS.

5 Conclusion

In this paper, we have proposed a general-purpose search scheme that can be combined with any variable ordering heuristic. Contrary to Last Conflict, Conflict Ordering Search is very aggressive, discarding progressively the role played by the heuristic. Besides, by means of simple timestamps, all variables recorded in the global conflict set stay permanently ordered, the priority being modified at each new conflict. We have shown that on some structured known problems our approach outperforms other generic and specific solving methods. So, COS should be considered as one new useful technique to be integrated in the outfit of constraint systems.

Acknowledgments

Steven Gay is financed by project Innoviris 13-R-50 of the Brussels-Capital region. Renaud Hartert is a Research Fellow of the Fonds de la Recherche Scientifiques - FNRS. Christophe Lecoutre benefits from the financial support of CNRS and OSEO (BPI France) within the ISI project Pajero. The authors would like to thank Petr Vilím for his help with the comparison to FDS in CP Optimizer.

References

1. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
2. Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *Proceedings of CP'15*. Springer International Publishing, 2015.
3. Steven Gay, Renaud Hartert, and Pierre Schaus. Time-table disjunctive reasoning for the cumulative constraint. *Integration of AI and OR Techniques in Constraint Programming*, pages 157–172, 2015.
4. C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
5. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

6. Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. Benchmark instances for project scheduling problems. In *Project Scheduling*, pages 197–212. Springer, 1999.
7. Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling, 1994.
8. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Nogood recording from restarts. In *Proceedings of IJCAI'07*, pages 131–136, 2007.
9. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
10. Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:147–167, 2007.
11. L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proceedings of CPAIOR'12*, pages 228–243, 2012.
12. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, pages 530–535, 2001.
13. OscaR Team. OscaR: Scala in OR, 2012. Available from `bitbucket.org/oscarlib/oscar`.
14. G. Pesant, C.-G. Quimper, and A. Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43:173–210, 2012.
15. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 294–299. Springer, 2007.
16. P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of CP'04*, pages 557–571, 2004.
17. Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 230–245. Springer, 2011.
18. Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *Integration of AI and OR Techniques in Constraint Programming*, pages 437–453. Springer, 2015.
19. Armin Wolf and Gunnar Schrader. O(n log n) overload checking for the cumulative constraint and its application. In *Declarative Programming for Knowledge Management*, pages 88–101. Springer, 2006.