Boosting incomplete search with conflict learning

Trong-Hieu Tran, Cédric Pralet, Hélène Fargier

▶ To cite this version:

Trong-Hieu Tran, Cédric Pralet, Hélène Fargier. Boosting incomplete search with conflict learning. Doctoral Program of the 27th International Conference on Principles and Practice of Constraint Programming, Oct 2021, Montpellier, France. 10.4230/LIPIcs.CP-DP.2021.7 . hal-04778224

HAL Id: hal-04778224

https://hal.science/hal-04778224v1

Submitted on 12 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Boosting incomplete search with conflict learning

Trong-Hieu Tran ✉
IRIT-CNRS, Université de Toulouse,Toulouse, France

Cédric Pralet ✉
ONERA/DTIS, Université de Toulouse, F-31055 Toulouse, France

Hélène Fargier ✉
IRIT-CNRS, Université de Toulouse,Toulouse, France

Abstract

In this paper, we introduce an ongoing work regarding a hybrid approach usable for obtaining high quality solutions to large-scale combinatorial optimization problems. This approach divides the process of solving a global problem into a master process that performs constraint-based search and a slave process that uses specific incomplete search techniques. In this hybrid architecture, the master level takes advantage of the conflicts discovered during incomplete search at the slave level, and reciprocally enhances the efficiency of the incomplete search since conflicts collected by the master level are used to avoid visiting the same parts of the search space over and over again. One of the novelties of this work is that the conflicts are memorized over the long-term in compact data structures, namely OBDDs or MDDs. The experimental results obtained on OPTW and FJSP instances show that even in our preliminary implementation, this kind of approach can reach some best-known results.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning; Mathematics of computing → Combinatorial optimization; Theory of computation → Randomized local search

Keywords and phrases incomplete search, knowledge compilation, conflict learning, GRASP

Digital Object Identifier 10.4230/LIPIcs.CP-DP.2021.7

1 Introduction

During the last decades, various incomplete search techniques were developed in the Operations Research (OR) community to quickly find good quality solutions to combinatorial optimization problems, especially for large instances. For example, local search (LS) approaches based on efficient neighborhoods such as *k-opt* were introduced for routing problems [18], many neighborhoods and metaheuristics were defined for Job Shop Problems (JSPs [19]), and techniques like ejection chains were applied to assignment problems [10], to name just a few. These developments were however made for specific OR problems, and the addition of side constraints often requires some refactoring.

On the opposite, a framework like Constraint Programming (CP) is less sensitive to the addition of side constraints. The main reason for this is that CP is based on modular declarative modeling languages and on generic search procedures like tree search with backtracking and constraint propagation.

In this paper, we develop a hybrid search strategy that uses specific OR incomplete search techniques within a generic constraint-based search framework to quickly obtain good quality solutions for a given problem. This strategy is based on the following principles:

- *Problem decomposition*: we use, on the one hand, a generic constraint-based master solver dealing with decisions and constraints that are out of the scope of standard OR problems, and on the other hand slave solvers that can fully exploit the specificity of some standard OR subproblems.

© Trong-Hieu Tran;
licensed under Creative Commons License CC-BY 4.0
The doctoral program of CP 2021.
Editor: Jeremias Berg; Article No. 7; pp. 7:1–7:10

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- *Incomplete search techniques and explanations for the subproblems*: to quickly get good-quality solutions, there is a need to avoid spending too much time in the resolution of one subproblem obtained for one combination of master decisions. This is why we use incomplete OR techniques (local search and its variants) at the level of each subproblem, with the assumption that inconsistency explanations can be delivered by the slave solver when no solution can be found. These explanations are potentially invalid, though, they can be added to the master problem for the sake of search diversification.
- GRASP (*Greedy Randomized Adaptive Search Procedure* [9]): several strategies can be used to interleave the master and slave solvers. We explore a GRASP-like metaheuristic that makes greedy choices at the master level: at each step, the master solver tries to assign a variable x with a value v and asks the subproblem whether it can still find a consistent solution. If yes, decision $[x = v]$ is committed. Otherwise, the master solver collects the explanation returned by the slave solver and tries to propose another decision. When the master solver encounters an inconsistency (no solution that is compatible with the set of accumulated conflicts), a restart from an empty assignment is performed.
- *Use of off-the-shell succinct data structures*: during the search, the master solver gathers conflict explanations returned by the slave solver. To get a compact representation of these conflicts and to quickly extract non-forbidden variable assignments, we use efficient data structures, namely Ordered Binary Decision Diagrams (OBDDs [6]) and Multivalued Decision Diagrams (MDDs [15, 1]).

Related work

Using problem decomposition and inconsistency explanations is not new in combinatorial optimization. See the use of Logic-Based Benders decomposition [13] in mathematical programming or SAT Modulo Theory (SMT [4]) for satisfiability problems. Basically, SMT exploits both a master problem involving Boolean choices tackled using highly efficient SAT techniques, and so-called *theory* solvers that are able to reason about the *predicates* activated by the Boolean choices. One difference with standard SMT is that our approach tackles the subproblems by incomplete search techniques that do not necessarily return an optimal solution. Also, with regard to existing works, the idea of using conflict-driven search is similar to Conflict-Driven Clause Learning (CDCL [3, 20]) used for SAT and Lazy Clause Generation (LCG [25]) used for constraint satisfaction. One difference though is that thanks to knowledge compilation languages like OBDD or MDD, the search process that we consider has no specific constraint on the order in which the variables of the model are assigned. Last, using specific reasoning procedures for some parts of the model is already done in CP through the specific constraint propagation mechanisms. Two differences here are that first we use incomplete reasoning techniques instead of "valid" constraint propagation, and second we specifically identify a master problem that can, for instance, deal with decisions corresponding to disjunctive choices.

Paper organization

The rest of the paper is organized as follows. Section 2 gives two examples that will be used throughout the paper. Section 3 details the two-level model considered. Section 4 describes the conflict-based search strategy using inconsistency explanations and knowledge compilation. Section 5 provides preliminary experimental results. Last, Section 6 gives the summary as well as some perspectives for this work.

2 Two examples

In the following, we consider two examples of problems for illustrating the approach.

Orienteering Problem with Time Windows (OPTW) [26]

OPTW is an extension of the Traveling Salesman Problem with Time Windows (TSPTW). In OPTW, we consider a set of N clients $c_1, \ldots, c_N$, with for each client c_i a time window usable for serving c_i, an associated service time, and a reward R_i obtained if c_i is served. Given a matrix of transition times between all pairs of clients, the objective is to visit (a subset of) clients within their allowed time windows while maximizing the total reward collected. A lower bound LB on the total reward can also be specified. This problem can be decomposed into two subproblems: (1) a master problem selecting the subset of clients to visit (one decision variable $x_i \in \{0,1\}$ per client c_i, subject to constraint $\sum_{i\in[1..N]} R_i x_i \geq LB$), and (2) a slave problem used for determining a sequence $\sigma = [c_{i_1}, \ldots, c_{i_k}]$ allowing to visit all clients selected by the master level within their time windows (a standard TSPTW); several efficient incomplete OR techniques are available for this second problem [23].

Flexible Job Shop Problem (FJSP) [16]

A standard JSP involves m machines and n jobs composed of a sequence of m operations, where each operation must be performed using a fixed machine. FJSP is an extension of JSP where each operation has a list of alternative machines, i.e. can be processed on any machine in this list. The objective is to find a schedule that processes all operations and minimizes the maximum completion time of an operation (makespan). Thus, FJSP can be decomposed into (1) a master problem that consists of assigning operations to machines (one machine-choice variable $x_i \in [1..M_i]$ per operation i), and (2) a slave problem corresponding to the sequencing of operations on machines, given an upper bound UB on the makespan; for this subproblem, efficient incomplete OR techniques available for JSP can be used [21].

3 Problem formulation

As usual, a Constraint Satisfaction Problem (CSP) is defined as a triple (X, D, C) where X is a finite set of variables, $D(x)$ is the finite domain of possible values of a variable $x \in X$, and C is a set of constraints over X. An assignment A of a set of variables $X' \subseteq X$ is a set of pairs (x, v) where $x \in X'$ and $v \in D(x)$, and where there is exactly one pair per variable $x \in X'$; set X' is also denoted by $\mathit{Vars}(A)$. The value associated with a variable x in A is referred to as $A[x]$. An admissible solution is an assignment of X that satisfies all the constraints in C. In the following, we describe the models used for the master and slave decision layers as CSPs. We focus on constraint *satisfaction*, given that optimization tasks can be fulfilled by solving a sequence of problems (CSPs can be augmented with a constraint on the objective value that is gradually tightened each time a new best solution is found).

3.1 Master problem

At the master level, we initially consider a CSP (X, D, C_m) — in our examples, C_m contains the selection (resp. allocation) constraints. Each assignment satisfying C_m that is found by the master search algorithm is given to the slave solver, that tries to tackle the corresponding subproblem — as in SMT [4], with the difference that in our work (i) the slave solver is a local search algorithm and (ii) the slave solver provides the master solver with *estimated* conflicts

whenever it cannot reach an admissible solution. Formally, a conflict (or explanation) A_c is an assignment of a subset of variables $\mathit{Vars}(A_c) \subseteq X$ that explains the inconsistency. Intuitively, each conflict corresponds to a set of decisions that *potentially* leads to an inconsistency.

Let $\mathcal{C}$ be the set of conflicts returned by the slave module after a new iteration, and C_{KB} be the set of conflicts learned during previous iterations ("KB" as "Knowledge Basis"). Then,

$$C_{KB}^{new} \leftarrow C_{KB}^{old} \cup \left\{ \neg \left(\bigwedge_{(x,v)\in A_c} (x = v) \right) \mid A_c \in \mathcal{C} \right\} \tag{1}$$

and the set of constraints of the master problem becomes:

$$C_m^{new} \leftarrow C_m^{init} \cup C_{KB}^{new} \tag{2}$$

A succinct data structure for memorizing conflicts

To circumvent the risk of memory explosion, we propose to use compact data structures such as OBDD [6] or MDD [2] for memorizing all the conflicts learned in C_{KB}. Basically, OBDDs and MDDs are decision diagrams whose nodes are labeled by variables and whose arcs represent value choices for the variables. The set of paths from a root node to a specific "*True*" leaf node in these diagrams then compactly represents a set of variable assignments (in our case, the set of assignments that are still acceptable). Such representations can be exponentially more compact than explicit lists of variable assignments. Additionally, OBDDs and MDDs allow to execute some basic queries in polytime, such as checking whether a complete assignment satisfies C_{KB} (consistency checking), extracting an assignment satisfying all constraints in C_{KB} (model extraction), or adding a new forbidden assignment.

3.2 Subproblems at the slave level

At the slave level, we must solve subproblems derived from the assignment chosen at the master level. Each subproblem contains a set of constraints C_s that must be satisfied. In the full generality of our scheme, these subproblems can be generic CSPs or more specific problems from the OR literature — e.g., TSPs, TSPTWs, JSPs, etc. Finding an admissible solution for each subproblem can itself be an NP-hard task. Besides, the subproblem varies from one call to another. For instance, in an OPTW, each new client selection at the master level (assignment $x_i = 1$ for a client i) adds a new sequencing variables to the subproblem handled at the TSPTW level; as for FJSPs, each machine assignment at the master level (assignment $x_i = m$ for an operation i) updates the (non-flexible) JSP handled at the slave level. Formally, we assume the existence of four generic functions ϕ_{set}, ϕ_{assign}, $\phi_{unassign}$, ϕ_{srch} for making the master and slave levels interact.

- $\phi_{set}(A) \implies (P, \alpha)$: this function allows instantiating the slave problem given an assignment at the master level. More precisely, A is an assignment of $X' \subseteq X$, P is the resulting subproblem at the slave level, defined over a set of variables Y, and α is an initial solution for P (i.e., an assignment of Y) that can for instance be constructed randomly or by a greedy algorithm.
- $\phi_{assign}(P, \alpha, x, v) \implies (P^{new}, \alpha^{new})$: this function incrementally updates the slave problem when assigning $[x = v]$ for a variable $x \in X$ and a value $v \in D(x)$, instead of generating the subproblem and the corresponding solution from scratch. For instance, when $x = v$ corresponds to the selection of a new client to be visited in an OPTW, the slave solver can update its current solution by inserting this new client at some position in the current sequence of visits.

- $\phi_{unassign}(P, \alpha, x) \implies (P^{new}, \alpha^{new})$: this function incrementally updates the slave problem when unassigning a variable $x \in X$. For instance, when x corresponds to the selection of a client in an OPTW, the subproblem can simply remove this client from the current sequence of visits.
- $\phi_{srch}(P, \alpha, mt) \implies (\alpha^{best}, sat, A_c)$: this function performs a local search for the slave problem starting from initial solution α, with a limited computation time mt. This function returns the best solution α^{best} found and a boolean value *sat* equal to *True* if α^{best} satisfies all the constraints of the slave problem, *False* otherwise. In this latter case, an explanation A_c of the inconsistency is also built (see below).

4 Incomplete conflict-based search with a knowledge basis

4.1 Generic scheme

The global search process proposed is described in Figure 1. The main idea is to use incomplete search techniques to progressively find an assignment A of X such that (1) A satisfies all the constraints in C_m and (2) there exists an admissible solution α of the subproblem P generated from A, i.e., $A \wedge \alpha$ satisfies all constraints in C_s. Similarly to what is done for (weighted) partial Max-SAT [7], it is possible to consider C_m as a set of *hard constraints* that have to be respected at all times, and C_s as a set of *soft constraints* that can be violated during search. Our goal is then to minimize the violation degree of the constraints in C_s by first incrementally updating the decisions made at the master level (decisions over A), and then performing local search at the slave level (decisions over α), while maintaining the satisfaction of all constraints in C_m at any time.

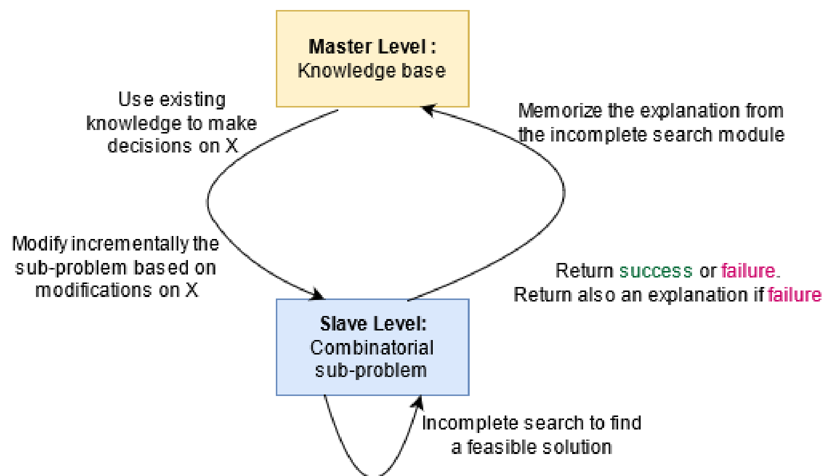

Figure 1 Proposed architecture

4.2 Pseudo-code of the method

The pseudo-code of the method proposed is given in Algorithm 1. At each step, the algorithm maintains a current assignment A for the master level and a corresponding set of unassigned variables $X_u \subseteq X$. Iterations are performed as long as there exists at least one unassigned variable, i.e. $X_u \neq \emptyset$ (lines 5-21). At each step, the solver searches for a completion A' of A that satisfies all constraints in C_m (line 5). This query can be addressed in polynomial

time with the use of an adequate language for representing the conflicts (polynomial time *in the size of the data structure*). If there is no model, the algorithm restarts from an empty assignment (line 6-9). Otherwise, one new assignment (x, v) belonging to A' is chosen using a specified heuristic. The latter can be problem-dependent (e.g., based on insertion heuristics used for OPTW or FJSP) or not (e.g., based on the ordering of variables in the decision diagrams). Subproblem P is updated according to the new assignment $[x = v]$ (line 11-14). Incomplete search is then used at the slave level to try and get an admissible solution for P (line 15). If a consistent solution is found, then the search continues by considering the remaining variables in X_u. Otherwise, the failure is analyzed and the conflict explanation is added to C_m (line 17). If no solution has been found after a given maximum CPU time, the algorithm returns *nil*, i.e. UNKNOWN (not UNSAT since we use incomplete search).

Algorithm 1 Incomplete Search With KB

Input: *MaxTime*: maximum global CPU time, *mt*: maximum CPU time for each search phase at the level of the subproblem
Output: An admissible solution for the global problem or *nil* if no solution has been found

1 $(X_u, A) \leftarrow (X, \emptyset)$
2 $(P, \alpha) \leftarrow \phi_{set}(\emptyset)$
3 **while** $(cpuTime() < MaxTime)$ **do**
4   **if** $X_u = \emptyset$ **then return** (A, α)
5   select $A' \in D(X_u)$ s.t. $A.A'$ solution of (X, C_m)
6   **if** $A' = nil$ **then**
7   **if** $A = nil$ **then return** *nil*
8         $(X_u, A) \leftarrow (X, \emptyset)$
9         $(P, \alpha) \leftarrow \phi_{set}(\emptyset)$
10   **else**
11         $(P^{old}, \alpha^{old}) \leftarrow (P, \alpha)$
12   select $(x, v) \in A'$
13         $(X_u, A) \leftarrow (X_u \setminus \{x\}, A \cup \{(x, v)\})$
14         $(P, \alpha) \leftarrow \phi_{assign}(P, \alpha, x, v)$
15         $(\alpha, sat, A_c) \leftarrow \phi_{srch}(P, \alpha, mt)$
16   **if** $\neg sat$ **then**
17             $C_m \leftarrow C_m \cup \{\neg(\wedge_{x \in Vars(A_c)}(x = A_c[x]))\}$
18             $(X_u, A) \leftarrow (X_u \cup \{x\}, A \setminus \{(x, v)\})$
19             $(P, \alpha) \leftarrow (P^{old}, \alpha^{old})$
20   **end**
21   **end**
22 **end**
23 **return** *nil*

4.3 Conflict extraction based on a QuickXplain algorithm

The algorithm used to produce conflicts is inspired by [14] (see Algorithm 2). When the local search module does not find an admissible solution within the allocated time, the objective is to extract a minimal set of decisions that explains the failure. Intuitively, the idea is to focus on decisions that can lead to a change in the subproblem consistency. If the unsatisfiable subproblem P becomes satisfiable again when unassigning a variable $z_k \in X$, then z_k is *potentially* involved in the explanation of the failure. This is why in our quick explanation approach, z_k is added to the *estimated* inconsistent explanation in this case. This procedure is repeated until all the variables in $Vars(A)$ are considered.

Algorithm 2 QUICKXPLAIN

Input: A: an assignment of a subset of variables $X' \subseteq X$ whose inconsistency must be evaluated, mt: maximum computation time allowed to evaluate the feasibility of the assignment at the level of the subproblem
Output: Z: a subset of variables $Vars(Z) \subseteq Vars(A) \subseteq X$ such that assignment $A[Z]$ explains the failure

```
1  [z_1, ..., z_p] ← permutation(X')
2  Z ← ∅
3  for k = 1 to p do
4  |  (P^old, α^old) ← (P, α)
5  |  (P, α) ← φ_unassign(P, α, z_k)
6  |  (α, sat, A') ← φ_srch(P, α, mt)
7  |  if sat then
8  |  |  Z ← Z ∪ {z_k}
9  |  |  (P, α) ← (P^old, α^old)
10 |  end
11 end
12 return Z
```

5 Experiments and discussion

5.1 Case study 1 : solving OPTW by using LS and OBDD

For OPTW, a boolean variable $x_i \in \{0, 1\}$ is used to represent the selection of client c_i. A conflict in this type of problem is a clause $(\neg x_{c_1} \vee \ldots \vee \neg x_{c_k})$. Because the conflicts involve boolean variables only, they can be efficiently stored in an OBDD. When the conflicts are stored in an OBDD, it is easy to extract or complete a model by following a path from the root node of the decision diagram to the *True* leaf node. Optimization of the global reward can also be handled, through the association of weights to the edges — these weights are the rewards corresponding to the visit of the clients (or 0 if the client is not selected).

Experiments were performed on classical OPTW instances.[1] In our implementation, we used the PyCUDD library to manage operations on OBDDs. Many of the best known bounds have been recovered by our approach. Computation times are also competitive with state-of-the-art methods. As we can see in Table 1, the number of OBDD nodes used for memorizing conflicts is far less than the number of conflicts encountered during the search.

Instance	ILS+GRASP [26]	ILS [27]	OBDD + LS	#conflicts	#OBDD (nodes)	CPU time (s)
c107	**370**	360	**370**	7350	1789	5.59
r101	**198**	182	**198**	5720	1708	3.38
r104	**303**	297	299	15936	2129	9.72
r107	**299**	288	294	11661	2347	7.94
r109	**277**	276	**277**	15118	4645	9.55

Table 1 Results obtained on classical instances of OPTW; CPU Time (s) was measured on an Intel Core i5-10210U processor, 1.6 GHz, 16 GB of RAM. Each test was performed with $maxIterations = 100$.

5.2 Case study 2 : solving FJSP by using LS and MDD

In FJSP, an integer variable $x_i \in [1..M_i]$ is used to represent the machine selection for operation i, where M_i is the number of alternative machines available for i. An inconsistency

[1] `https://www.mech.kuleuven.be/en/cib/op`

explanation is, in this case, a minimal set of decisions $A_c = \{(x_{i_1}, v_{i_1}), \ldots, (x_{i_k}, v_{i_k})\}$ such that no consistent plan is found with regards to a *makespan* upper bound UB given as an input. Such conflicts over integer variables can be efficiently stored in an MDD [2]. For the management of MDDs, we used the SALADD library [24].

We carried out experiments on the Brandimate instances [5]. An analysis of the best makespan found for some instances is shown in Table 2. Our solver (MDD + LS), however, falls short of reaching several best-known results. One of the reasons could be the lack of efficiency of the local search procedure at the slave level. Intuitively, even if the master level makes an optimal machine selection (i.e. a selection that can lead to an optimal global solution), an optimal sequencing might not be found at the slave level since incomplete search is performed. Another weakness is that the number of conflicts learned remains limited. More precisely, our implementation of the QuickXplain procedure seems not to be effective yet in several cases since the conflict extraction time is too long compared to the global search time, especially for large-scale instances.

Instance	$n \times m$	LB	LEGA [12]	KBACO [28]	KBVNS [16]	HA [17]	MDD+LS
mk01	10×6	36	40	**39**	40	40	40
mk02	10×6	24	29	29	**26**	**26**	27
mk03	15×8	204	-	**204**	**204**	**204**	**204**
mk04	15×8	48	67	65	**60**	**60**	61
mk05	15×4	168	176	173	173	**172**	177
mk06	10×15	33	67	67	58	**57**	65
mk07	20×5	133	147	144	**139**	**139**	146
mk08	20×10	523	**523**	**523**	**523**	**523**	**523**
mk09	20×10	299	320	311	311	**307**	**307**
mk10	20×15	165	229	229	209	**197**	219

Table 2 Makespan values obtained on Brandimate instances [5] for state-of-the-art methods and for our approach (MDD+LS). Each test was performed with $maxTime = 60s$.

6 Conclusion and future works

In this paper, we introduced an ongoing work related to a hybrid method for solving optimization problems, where a generic master solver (reasoning typically about selection/allocation constraints) and a specific slave solver (reasoning typically about routing/scheduling decisions through fast incomplete search procedures) collaborate. To improve search, the conflicts encountered during incomplete search at the slave level are recorded in compact data structures, namely OBDDs or MDDs, and reused to guide the solving process.

Preliminary experiments showed that for several benchmarks, there is probably a need to intensify search at the slave level (for instance by using Large Neighborhood Search instead of Local Search), and a need to speed up the extraction of an explanation when a failure occurs. The experiments also showed that the conflicts learned during search can be efficiently stored thanks to OBDDs or MDDs, but the size of the decision diagrams and the compilation time increase while conflicts are added. To overcome this limitation, other knowledge compilation languages could be considered [8], as well as approximate compilation techniques [11]. Another important factor affecting the performance of decision diagrams is the variable ordering [22], and this point should also be further investigated.

A last perspective is to handle uncertainty in combinatorial problems where some information can be changed at the last minute. In such cases, the knowledge basis could be reused to avoid performing search from scratch.

References

1. J. Amilhastre, H. Fargier, A. Niveau, and C. Pralet. Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. *International Journal on Artificial Intelligence Tools*, 23(04):1460015, 2014.
2. H.R. Andersen, T. Hadzic, J.N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.
3. G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais. Integrating conflict driven clause learning to local search. *arXiv preprint arXiv:0910.1247*, 2009.
4. C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.
5. P. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations research*, 41(3):157–183, 1993.
6. R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
7. B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MAXSAT. *AAAI/IAAI*, 263268:9, 1997.
8. A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.
9. T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
10. F. Glover and C. Rego. Ejection chain and filter-and-fan methods in combinatorial optimization. *4OR*, 4(4):263–296, 2006.
11. T. Hadzic, J.N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 448–462. Springer, 2008.
12. N.B. Ho, J.C. Tay, and E. M-K Lai. An effective architecture for learning and evolving flexible job-shop schedules. *European Journal of Operational Research*, 179(2):316–333, 2007.
13. J. Hooker and G. Ottosson. Logic-based Benders decomposition. *Mathematical Programming*, 96:33–60, 2013.
14. U. Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, volume 4. Citeseer, 2001.
15. T. Kam, T. Villa, and R. Brayton. Multi-valued decision diagrams: theory and applications. In *Multiple-Valued Logic*, volume 4, pages 9–62, 1998.
16. H. Karimi, S.H.A. Rahmati, and M. Zandieh. An efficient knowledge-based algorithm for the flexible job shop scheduling problem. *Knowledge-Based Systems*, 36:236–244, 2012.
17. X. Li and L. Gao. An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem. *International Journal of Production Economics*, 174:93–110, 2016.
18. S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.
19. A.S. Manne. On the job-shop scheduling problem. *Operations research*, 8(2):219–223, 1960.
20. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021.
21. M. Mastrolilli and L. M. Gambardella. Effective neighbourhood functions for the flexible job shop problem. *Journal of scheduling*, 3(1):3–20, 2000.
22. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.
23. M.W.P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.
24. N. Schmidt. *Compilation de préférences: application à la configuration de produit.* PhD thesis, Artois, 2015.

25 A. Schutt, T. Feydy, P.J. Stuckey, and M.G. Wallace. Solving RCPSP/max by lazy clause generation. *Journal of scheduling*, 16(3):273–289, 2013.

26 W. Souffriau, P. Vansteenwegen, G.V. Berghe, and D. Van Oudheusden. The multiconstraint team orienteering problem with multiple time windows. *Transportation Science*, 47(1):53–63, 2013.

27 P. Vansteenwegen, W. Souffriau, G.V. Berghe, and D. Van Oudheusden. Iterated local search for the team orienteering problem with time windows. *Computers & Operations Research*, 36(12):3281–3290, 2009.

28 L.-N. Xing, Y.-W. Chen, P. Wang, Q.-S. Zhao, and J. Xiong. A knowledge-based ant colony optimization for flexible job shop scheduling problems. *Applied Soft Computing*, 10(3):888–896, 2010.