Combining Incomplete Search and Clause Generation: An Application to the Orienteering Problems with Time Windows

Trong-Hieu Tran[1,2,3](✉), Cédric Pralet[2,3], and Hélène Fargier[1,3]

[1] IRIT-CNRS, Toulouse, France
[2] ONERA/DTIS, Toulouse, France
trantronghieu97@gmail.com
[3] Université de Toulouse, Toulouse, France

Abstract. In this paper, we present a hybrid optimization architecture which combines on one side incomplete search processes that are often used to quickly find good-quality solutions to large-size problems, and on the other side clause generation techniques that are known to be efficient to boost systematic search. In this architecture, clauses are generated once a locally optimal solution is found. We introduce a generic component to store these clauses generated step-by-step. This component is able to prune neighborhoods by answering queries formulated by the incomplete search process. We define three versions of this clause basis manager and then experiment with an Operations Research problem known as the Orienteering Problem with Time Windows (OPTW) to show the efficiency of the approach.

Keywords: Incomplete search · Clause generation · Orienteering Problem with Time Windows

1 Introduction

Incomplete search methods are often used on large-size problems to quickly produce good-quality solutions. Such methods include *heuristic search*, where a solution is progressively built based on efficient heuristics, *local search*, where various neighborhoods help improve the current solution, and *metaheuristics* like tabu search, genetic algorithms, or iterated local search, to name just a few. To increase the performance of these incomplete methods, several hybridizations with complete search techniques developed for SAT and Constraint Programming (CP) have been proposed in the past [24], and there is rich literature on the topic both in terms of methods and applications.

In this paper, we study a new architecture combining incomplete search and SAT techniques. This architecture, which is given in Fig. 1, is inspired by the efficient complete search methods based on clause generation, namely CDCL [1,18]

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
A. A. Cire (Ed.): CPAIOR 2023, LNCS 13884, pp. 493–509, 2023.
https://doi.org/10.1007/978-3-031-33271-5_32

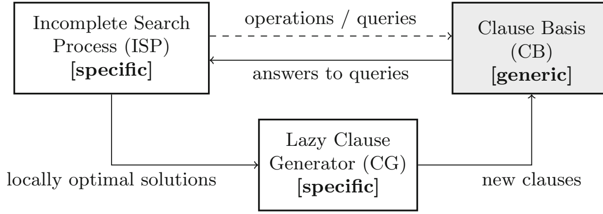

Fig. 1. Incomplete search combined with a clause basis

and Lazy Clause Generation (LCG [28]). The global search scheme works as follows. Each time the Incomplete Search Process (ISP) converges to a locally optimal solution, a *Clause Generator* (CG) analyzes this solution and produces clauses holding on Boolean decision variables of the problem. The clauses generated represent either the reasons why the current solution cannot be improved or conditions forbidding the local optimum or regions around to be reached again in the future. The clauses generated are then sent to a *Clause Basis* (CB). The latter is responsible for storing the clauses and answering various queries that are relevant for the main ISP to prune or to guide the neighborhood exploration. In this architecture, the clauses are generated in a lazy way, only for the parts of the search space that the ISP decides to explore. By doing so, the architecture involves a tight interaction between ISP and CB as well as a less frequent clause generation phase.

With regards to tabu search [9], the architecture obtained memorizes clauses instead of just storing recent local moves or recent solutions in a tabu list. One impact is that the clause manager must be able to quickly reason about the clauses collected, instead of just reading explicitly forbidden configurations. Concerning CDCL or LCG, one key difference is that the ISP is free to assign or unassign variables *in any order*, while the standard *implication graph* data structure used by CDCL or LCG relies on the assumption that the variable ordering in different layers of the graph is consistent with the order used for assigning and unassigning the decision variables. All these points raise several basic research questions:

- Which generic clause basis data structure should be used to be able to follow the decisions made in any order by an incomplete search process and to quickly reason about the set of clauses memorized?
- What is the effort required to integrate an existing specific ISP within such a generic architecture, and which key functions should the clause basis offer?
- What is the content of the clause generation module that analyzes the locally optimal solutions?
- From a practical point of view, is clause generation beneficial for an incomplete search that might have to explore thousands of successive neighborhoods per second?

To answer these questions, all along the paper, we take as an example a problem known as the Orienteering Problem with Time Windows (OPTW [32]). The paper is organized as follows: first, we recall the definition of OPTW and present

a hybrid algorithm that combines a state-of-the-art search algorithm for OPTW with a clause basis. Following this, we introduce clause generation mechanisms and three data structures for the clause basis. We present experimental results obtained on OPTW benchmarks to demonstrate that the architecture proposed allows boosting the baseline ISP, while the integration effort required is rather small. Finally, we compare our approach with relevant works in the literature.

2 Orienteering Problem with Time Windows

The OPTW belongs to the class of vehicle routing problems with profits, where a vehicle has a limited time budget to visit a subset of nodes. Formally, we consider a set of nodes $i \in \{0, 1, \ldots, N+1\}$, each with a reward R_i and a predefined time window $[e_i, l_i]$. Nodes 0 and $N+1$ correspond to the start and end depots, with $R_0 = R_{N+1} = 0$ and a time window $[0, T_{max}]$ where T_{max} is the limited time budget. A non-negative travel time t_{ij} is associated with each pair of nodes $i \neq j$. A visit duration d_i can also be considered for each node, but to keep it simple, we assume that the visit duration is already included in the travel time. A solution is a sequence $\sigma = [\sigma_0, \sigma_1, \ldots, \sigma_K, \sigma_{K+1}]$ that starts at node $\sigma_0 = 0$, visits a set of distinct nodes $\sigma_i \in \{1, \ldots, N\}$, and returns to node $\sigma_{K+1} = N+1$. Early arrival to a particular node leads to a waiting time, and a solution is feasible when it visits each selected node before its latest arrival time. More precisely, the visit start time of node 0 is $s_0 = 0$, and for two consecutive nodes i, j in σ the visit start time of node j is $s_j = \max(s_i + t_{ij}, e_j)$, and solution σ is feasible if and only if $s_j \leq l_j$ for every node j in σ. Next, an optimal solution is a feasible solution σ that maximizes the total reward ($\sum_{i \in \sigma} R_i$). Basically, an OPTW involves both selection and sequencing decisions, i.e. selection of a subset of nodes S and search for a feasible visit order σ for these nodes. Regarding the selection aspect, we introduce one Boolean decision variable $x_i \in \{0, 1\}$ per node i, where $x_i = 1$ means that node i is visited.

Challenging applications were modeled as OPTW in the past, such as delivery problems, satellite planning problems, or tourist trip design problems [11,32]. Since OPTW is proven as NP-hard [10], many researches on the topic rely on incomplete search. One first investigation in this direction was a tree heuristic for building a single tour without violating the time windows constraints [16]. Incremental neighborhood evaluation methods were also introduced to quickly determine the feasible node insertion moves given a current solution [29,31]. Later, [27] proposed an effective Large Neighborhood Search (LNS) strategy that was shown to outperform the previous approaches. The basic idea is to iteratively remove and reinsert nodes based on well-tuned removal and insertion heuristics, and to use restarts from a pool of elite solutions.

3 Incomplete Search Using a Clause Basis

In the rest of the article, we integrate the state-of-the-art LNS algorithm for OPTW defined in [27] within the hybrid architecture proposed. The enhanced

version, called *LNS-CB* for *LNS with a Clause Basis*, is depicted in Algorithm 1, where the few changes made on the baseline LNS version are highlighted in gray. Starting from an initial solution (Line 1), it iteratively destroys and repairs the current solution following the standard concept of LNS (Lines 5–6). It also uses an elite pool to record the best solutions obtained so far. This pool is reset whenever a better solution is found, and extended when a new equivalent solution is obtained (Line 9). When no improvement is found after R iterations, a restart is performed by picking a random solution in the elite pool (Line 11). The differences compared to the classical LNS algorithm are (a) the call to the clause generation function each time a full solution is produced (Lines 2, 7), and (b) the use of the CB as an argument to the repair function, the objective being to improve the repair phase (Line 6).

Algorithm 1. LNS-CB

```
1:  σ ← CONSTRUCT()
2:  CLAUSEGENERATION(σ, CB, maxConfSize)
3:  σ* ← σ;   elitePool ← {σ}
4:  while time limit is not reached do
5:      σ ← DESTROY(σ);
6:      σ ← REPAIR(σ, CB)
7:      CLAUSEGENERATION(σ, CB, maxConfSize)
8:      if σ better than σ* then
9:          σ* ← σ;   update elitePool
10:     else if no improvement after R iterations then
11:         σ ← a random solution in elitePool
12:     end if
13: end while
14: return σ*
```

The new repair phase is detailed in Algorithm 2. It takes as an input the current solution σ and the CB. We denote U as the set of unvisited nodes, and F as the set of feasible insertion moves (n, p) defined by a node $n \in U$ and a position p in σ. All insertion alternatives for each unvisited node are explored by EVALNEIGHBORHOOD(σ,U, CB) (Lines 2, 7). In this procedure, CB is used to prune neighbors that are invalid according to the clauses registered. Node insertions are iterated by selecting at each step a move that has the best evaluation according to the well-tuned heuristics of the original LNS method (Line 4), and they are performed until there is no more feasible move (Line 3).

The neighborhood evaluation function corresponds to Algorithm 3. It first determines the unvisited nodes that *must* be visited according to CB (Line 1), and if there is no such mandatory node, it determines the unvisited nodes that *can* be visited according to CB (Line 3). Then, for each node selected, the algorithm determines its best insertion position according to tuned insertion heuristics and the algorithm returns all pairs made by a node and its best insertion position.

Algorithm 2. REPAIR(σ,CB)

1: $U \leftarrow$ nodes that are not in σ
2: $F \leftarrow$ EVALNEIGHBORHOOD(σ, U, CB)
3: **while** $F \neq \emptyset$ **do**
4:    $(n^*, p^*) \leftarrow$ SELECT(F)
5:    Insert node n^* at position p^* in σ
6:    $U \leftarrow U \setminus \{n^*\}$
7:    $F \leftarrow$ EVALNEIGHBORHOOD(σ, U, CB)
8: **end while**
9: **return** σ

Algorithm 3. EVALNEIGHBORHOOD(σ,U,CB)

1: $U' \leftarrow \{n \in U \mid CB$ allows decision $[x_n = 1]$ and forbids decision $[x_n = 0]\}$
2: **if** $U' = \emptyset$ **then**
3:    $U' \leftarrow \{n \in U \mid CB$ allows decision $[x_n = 1]\}$
4: **end if**
5: $F \leftarrow \emptyset$
6: **for** each $n \in U'$ **do**
7:    $p \leftarrow$ best feasible insertion position for n in σ
8:    **if** $p \neq nil$ **then**
9:       $F \leftarrow F \cup \{(n, p)\}$
10:    **end if**
11: **end for**
12: **return** F

4 Lazy Clause Generation Module

Several kinds of clauses are generated during the search, and the generation of these clauses exploits problem-dependent techniques, as for cuts generated in Logic-Based Benders decomposition [14]. Note that for OPTW, we consider only clauses holding over the selection decisions, and not clauses related to the detailed sequencing decisions defining the order of the visits.

4.1 Clauses Generated from Time-Window Conflicts

A Time-Window conflict (TW-conflict) is a subset $S_c \subseteq [1..N]$ such that there is no feasible solution visiting all nodes in S_c. In terms of clause generation, a TW-conflict S_c corresponds to clause $\vee_{i \in S_c} \neg x_i$. Due to the exponential number of possible sets S_c, we generate TW-conflicts in a lazy way i.e. only when a local optimum is reached. Moreover, determining whether S_c defines a TW-conflict is NP-hard [26], but it is an easy problem if $|S_c|$ is bounded. This is why we consider a predefined maximum TW-conflict size referred to as *maxConfSize*.

Technically, whenever a locally optimal sequence σ^* is found over nodes in S^*, we seek TW-conflicts preventing the other nodes from being added to σ^*. In Algorithm 4, we try to find explanations for every unvisited node i (Line 1). With a predefined *maxConfSize*, the algorithm first heuristically selects a

Algorithm 4. CLAUSEGENERATION(σ^*,CB,*maxConfSize*)

1: **for** $i \notin \sigma^*$ **do**
2: $\quad S_c \leftarrow$ SELECT(σ^*, i, *maxConfSize*)
3: $\quad \mathcal{C} \leftarrow$ EXTRACTMINTWCONFLICTS($S_c \cup \{i\}$)
4: $\quad$ **for** each TW-conflict $C \in \mathcal{C}$ **do**
5: $\quad\quad$ Generate clause $\bigvee_{j \in C} \neg x_j$
6: $\quad$ **end for**
7: **end for**
8: [optional] Generate a temporary clause $\bigvee_{j \in Y} x_j$ with Y a subset of the nodes that are not selected in σ^*

set $S_c \subset S^* \setminus \{0, N+1\}$ containing *maxConfSize* $-$ 1 nodes in σ^* that *might* prevent node i from being visited. Then, in function EXTRACTMINTWCONFLICTS, a dynamic programming (DP) procedure determines whether $S_c \cup \{i\}$ is truly a TW-conflict. If so, it also extracts TW-conflicts of minimal cardinality (Line 3). Indeed, the smaller the clauses the better, since smaller clauses prune larger parts of the search space. Function EXTRACTMINTWCONFLICTS takes as an input a set of nodes $S \subseteq [1..N]$ and determines all minimal sets $S' \subseteq S$ (minimal in terms of cardinality) such that there is no feasible solution visiting all nodes in S'. For space limitation reasons, we do not detail the pseudo-code of EXTRACTMINTWCONFLICTS, but the key idea is to compute, for each set $C \subseteq S$, quantities of the form $a(C,i)$ representing the earliest arrival time at node $i \in C$ for a path starting at node 0, visiting all nodes in $C \setminus \{i\}$, and ending at node i. As in existing methods for Traveling Salesman Problems with Time Windows [3], these quantities are computed by increasing the size of C following a recursive formula. Then, $C \subseteq S$ is a TW-conflict when for every $i \in C$, either $a(C,i) > l_i$ or $\max\{a(C,i), e_i\} + t_{i,N+1} > T_{max}$. The first condition corresponds to late arrivals for every candidate last node i, while the second one corresponds to the violation of the time limit when returning to node $N+1$.

4.2 Clauses Related to Local Optima: Lopt-Conflicts

To avoid revisiting again and again the same solution, whenever reaching a locally optimal solution σ^*, it is possible to generate clause $\bigvee_{j \notin \sigma^*} x_j$ to force that at least one node unvisited in σ^* must be selected in the future. Such a clause is called a local optimum conflict or *Lopt-conflict*. To get small clauses that have a higher pruning power, we consider a maximum clause size *approxSize* and derive an *approximate* Lopt-conflict corresponding to a smaller clause $\bigvee_{j \in Y} x_j$ where Y contains at most *approxSize* nodes that are not involved in σ^* and that are chosen in function of their rewards (Algorithm 4, Line 8). To avoid pruning optimal solutions, such approximate clauses are used by CB only during a certain number of steps called *tabuSize*, similarly to a tabu search procedure, the main objective being to diversify search. We could also generate Pseudo-Boolean constraint $\sum_{i \in \{1,\ldots,N\}} R_i x_i \geq LB + 1$ whenever a new best total reward LB is found, but we focus here on clause generation.

5 Clause Basis Data Structures

The CB part is responsible for storing the clauses generated during search. Besides, the ISP needs to frequently query the clause basis, meaning that there is a need for *continuous* and *incremental* interactions between these two components. This raises many challenging questions about the choice of a specific data structure for CB. In principle, a clause basis manager must be able to:

- quickly integrate all the clauses generated step-by-step and compactly represent them (possibly with some trashing when the size of CB becomes too large);
- frequently update the partial assignment of the decision variables over which the clauses hold, to keep up-to-date knowledge of the content of the current solution considered by the main ISP. For LNS-CB, this occurs whenever a node is selected or removed, and these assign/unassign decisions can be sent to CB in any order;
- quickly answer to queries formulated by ISP, such as "evaluate whether decision $[x_i = 1]$ is feasible". For OPTW, if CB proves that this decision is infeasible given the current assignment and the clauses generated, then testing the insertion of node i in the current solution σ is unnecessary (neighborhood pruning). Another example is: "evaluate whether decision $[x_i = 1]$ is mandatory". If so, node i *must* be inserted into σ.

In the following we study three generic versions for CB:

- CB-UnitPropagation, where CB stores a list of clauses and performs incremental and decremental unit propagation to *evaluate* the consistency of the clause store for a given partial assignment of the x_i variables;
- CB-IncrementalSAT, where CB stores a list of clauses and employs powerful modern SAT solvers supporting incremental or assumption-based solving [2,7,21];
- CB-OBDD, where the clauses are stored in an Ordered Binary Decision Diagram (OBDD), a data structure defined in the field of knowledge compilation that has good compactness and efficiency properties [4,6].

5.1 CB-UnitPropagation

For this version of CB, unit propagation is used to prune infeasible values for the decision variables. In SAT, unit propagation can be achieved based on the *two-watched literals* technique, which consists in maintaining, in each clause, two distinct literals that can take value *true* [20]. In case there is no valid watched literal for a clause c, an inconsistency is detected. If only a single valid watched literal l is found, then clause c becomes unit and l must necessarily be *true* to satisfy the clause. In this case, literal $\neg l$ takes value *false* and unit propagation is applied to other clauses to further detect other propagated decisions.

In SAT, one advantage of the watched literals is that no literal reference needs to be updated when chronological backtracking occurs. But during incomplete

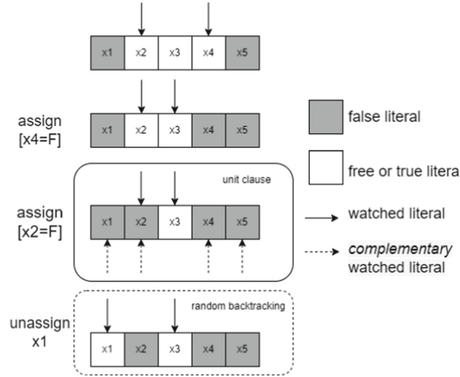

Fig. 2. Incremental and decremental unit propagation

search, variables can be assigned or unassigned in any order and some adaptations are required to maintain the watched literals. Precisely, to handle random variable unassignments and perform *decremental* unit propagation, we maintain a list of *complementary* watched literals *for each unit clause* c (see Fig. 2). Clause c is revised whenever one complementary watched literal l' becomes free due to unassignment decisions, and in this case l' can directly become a watched literal for c.

To answer the queries formulated by the ISP, we record a *justification* $justif(l)$ for each literal l. Basically, $justif(l) = \top$ means that literal l takes value true because of a decision received from the ISP, $justif(l) = c$ means that literal l is propagated by unit clause c, and $justif(l) = nil$ means that there is no clue about the truth value of l. Then, a decision like $[x = 1]$ is allowed if and only if literal $\neg x$ is not propagated or decided yet, i.e. $justif(\neg x) = nil$. The justification of each literal is updated during incremental and decremental unit propagation. Obviously, as unit propagation is incomplete, CB-UnitPropagation may not detect some infeasible or mandatory node selections. For example, let us consider four clauses $c_1 : \neg x_1 \vee \neg x_2$, $c_2 : \neg x_4 \vee \neg x_5$, $c_3 : x_2 \vee x_3 \vee x_4$, $c_4 : x_2 \vee x_3 \vee x_5$. If decision $[x_1 = 1]$ is made, clause c_1 becomes unit and we have $justif(x_1) = \top$ and $justif(\neg x_2) = c_1$. The other justifications take value *nil*. This implies that decision $[x_3 = 0]$ is still evaluated as possible, even if it would lead to a dead-end.

5.2 CB-IncrementalSAT

The idea of using incremental SAT solving was first proposed to improve the efficiency of the search for Minimal Unsatisfiable Sets [2]. In this case, the goal is to reuse as much information as possible between the successive resolutions of similar SAT problems. This is done by working with *assumptions*. Basically, an assumption $\mathcal{A}$ is a set of literals $\{l_1, \ldots, l_k\}$ where each literal is considered as an additional (unit) clause by the solver, but this unit clause is not permanently added to the original CNF formula $\mathcal{F}$ defining the problem to be solved. For

OPTW, the assumptions are exactly the node selection decisions. Then, a call $solve(\mathcal{F}, \mathcal{A})$ to an incremental SAT solver tries to find a model of $\mathcal{F}$ that satisfies all the assumptions in $\mathcal{A}$. Doing this, the incremental solver can reuse some previous knowledge and learn new clauses that will potentially be reused for future calls $solve(\mathcal{F}', \mathcal{A}')$ using an updated CNF formula $\mathcal{F}'$ or an updated set of assumptions $\mathcal{A}'$.

At the level of CB, to determine whether literal $l : [x_i = a]$ can still be assigned value true, it suffices to call $solve(\mathcal{F}, \mathcal{A} \cup \{l\})$ where $\mathcal{A}$ is the set of assumptions representing the selection decisions made so far by the search engine. Then, decision $[x_i = a]$ is forbidden by CB if and only if this call returns UNSAT. Contrarily to CB-UnitPropagation, the CB-IncrementalSat method is complete (it performs a full look ahead). One optimization allows us to reduce the number of calls to the *solve* function: when searching for the possible values of variable x_i given a set of assumptions $\mathcal{A}$, if $solve(\mathcal{F}, \mathcal{A} \cup \{x_i\})$ or $solve(\mathcal{F}, \mathcal{A} \cup \{\neg x_i\})$ returns a solution where another variable x_j takes value 1, then $x_j = 1$ is allowed and calling $solve(\mathcal{F}, \mathcal{A} \cup \{x_j\})$ is unnecessary.

5.3 CB-OBDD

Storing conflict clauses in an OBDD during a systematic search process has been explored in the past, e.g. for a search process based on DPLL [15]. We extend such an approach to deal with an incomplete search engine that again can assign/unassign the decision variables of the problem in any order.

As illustrated in Fig. 3, an OBDD defined over a set of Boolean variables X is a directed acyclic graph composed of one root node, two leaf nodes labeled by $\top$ and $\bot$, and non-leaf nodes labeled by a decision variable $x_i \in X$. Each node associated with variable x_i has two outgoing arcs corresponding to assignments $[x_i = 0]$ and $[x_i = 1]$ respectively (dotted and plain arcs in the figure). The paths from the root node to leaf node $\top$ correspond to the assignments that satisfy the logical formula represented by the OBDD, while the paths leading to leaf node $\bot$ correspond to the inconsistent assignments. Additionally, OBDDs are *ordered*, meaning that the variables always appear in the same order in any path from the root to the leaves. In practice, they are also *reduced*, meaning that redundant nodes (that have the same children) are recursively merged to save some space. Such a data structure offers several advantages, including the capacity to be exponentially more compact than an explicit representation of all models of a logical formula, and the capacity to perform several operations and answer several queries in polynomial time. For instance, given two OBDDs $\mathcal{O}_F$ and $\mathcal{O}_G$ representing logical formulas f and g and that use the same variable ordering, operation "$\mathcal{O}_F \wedge \mathcal{O}_G$" computes an OBDD representing $f \wedge g$ in polynomial time in the number of nodes in $\mathcal{O}_F$ and $\mathcal{O}_G$.

In CB-OBDD, one OBDD referred to as $\mathcal{O}_{CB}$ stores the clauses learned during search. Initially, $\mathcal{O}_{CB}$ only contains the leaf node $\top$ since all models are accepted. Each generated clause c_k can be transformed into an OBDD $\mathcal{O}_{c_k}$, and a set of new clauses $\{c_1, \ldots, c_n\}$ is added to $\mathcal{O}_{CB}$ by $\mathcal{O}_{CB} \leftarrow [\mathcal{O}_{c_1} \wedge \ldots \wedge \mathcal{O}_{c_n}] \wedge \mathcal{O}_{CB}$ (conjunction of the elementary OBDDs associated with the new clauses

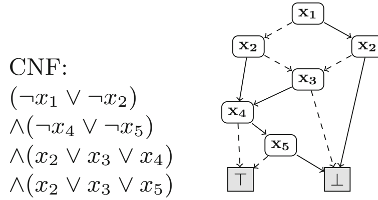

Fig. 3. A conjunction of clauses and an equivalent OBDD

followed by a batch addition into $\mathcal{O}_{CB}$). During search, CB-OBDD records the current list of assignments A_{CB} made by the incomplete search process (the assumptions). To determine whether a decision $[x = 1]$ is allowed, it suffices to *condition* $\mathcal{O}_{CB}$ by A_{CB}, and then to check that assignment $x = 0$ is not *essential* (not mandatory) for the resulting OBDD. The *conditioning primitive* and the search for *essential variables* are standard operations in OBDD packages. Their time complexity is linear in the number of OBDD nodes.

6 Computational Study

We carried out experiments on standard OPTW benchmarks[1] whose features are summarized in Table 1. The best known total reward for each instance is retrieved from [27]. All the experiments are performed on Intel(R) Core(TM) i5-8265U 1.60 GHz processors with 32 GB RAM. All implementations[2] are in C++ and compiled in a Linux environment with g++17.

Table 1. Features of the OPTW benchmarks

Instance Set	#instances	#nodes	remark
Solomon 1 (c1*, r1*, rc1*)	29	100	–
Solomon 2 (c2*, r2*, rc2*)	27	100	wider TW
Cordeau 1 (pr01-pr10)	10	48–288	–
Cordeau 2 (pr11-pr20)	10	48–288	wider TW

As the implementation of the state-of-the-art LNS algorithm [27] is not available online, we re-implemented it. We recover a similar performance even if there are some differences wrt. the results provided in the original paper, possibly due to random seeds or to a lack of information concerning a reset parameter R (we set $R = 50$ in our LNS implementation). Anyway, our primary objective was to determine whether conflict generation can help a baseline algorithm, therefore the slight differences in performance are not a real issue. The three CB proposed were implemented as follows:

[1] https://www.mech.kuleuven.be/en/cib/op.
[2] Github URL of the source code: https://github.com/thtran97/kb_ls_cpp.

- The CB-UnitPropagation data structure was implemented from scratch.
- For CB-IncrementalSat, we reused CryptoMiniSat[3] [30] that won the Incremental Track in the SAT competition 2020.
- For CB-OBDD, we reused the CUDD library that offers many functions to manage OBDDs.[4] CB-OBDD uses the dynamic reordering operations of CUDD [25]. Dynamic reordering can take some time but reducing the size of OBDDs can pay off in the long term.

6.1 Parameter Settings for clauseGeneration

In the hybrid optimization architecture proposed, the clauseGeneration procedure is problem-specific. For OPTW, we observed that the length of time windows has a large impact on the number of TW-conflicts generated for a given value of *maxConfSize*: many TW-conflicts are generated for the Solomon 1 & Cordeau 1 instances, contrarily to the Solomon 2 & Cordeau 2 instances that involve longer time windows. This is reasonable since longer time windows make the problem less constrained when considering only a few nodes. Besides, the complexity of the dynamic programming algorithm producing the TW-conflicts is exponential in *maxConfSize*. Thus, we decided to set $maxConfSize = 4$ after the analysis of the global search efficiency.

Another parameter is the heuristic according to which, given a locally optimal solution σ^* visiting a set of nodes S^*, we choose a subset $S_c \subseteq S^*$ for trying to explain why a customer $i \notin S^*$ cannot be inserted into σ^* (TW-conflicts). For this, we use the *NearestTimeWindow* heuristic: to define S_c, we choose $maxConfSize - 1$ nodes $j \in S^*$ such that the distance between the midpoint of the time window of j and the midpoint of the time window of i is as small as possible. However, generating TW-conflicts all the time can slow down the global search. Therefore, we define an *explanation quota xpQuota* for every node to reduce the workload of function extractMinTWconflicts. This quota is decreased by one unit each time a TW-conflict explaining the absence of i in a locally optimal solution is looked for. When the quota of i becomes 0 after *xpQuota* searches for TW-conflicts related to i, the absence of i in a locally optimal solution is not explained anymore. With such an approach, there is somehow a warm-up phase where TW-conflicts are learned, followed by an exploitation phase of these conflicts. After performing tests with different values of $xpQuota \in \{20, 60, 100\}$, we decided to set $xpQuota = 20$.

Last, concerning the generation of Lopt-conflicts to diversify search, we need to forbid during *tabuSize* iterations a region around a locally optimal solution, where the region size is controlled by the *approxSize* parameter which defines the maximum size of the approximate Lopt-conflicts. After several tests performed with $approxSize \in \{3, 5, 7\}$ and $tabuSize \in \{10, 50, 100, 200\}$, we set $approxSize = 7$ and $tabuSize = 50$ for the experiments.

[3] https://github.com/msoos/cryptominisat.
[4] https://github.com/ivmai/cudd.

6.2 Performance of the Versions of CB

Experiments are performed for the three CB data structures presented before. For LNS-CB-UnitPropagation (or shortly LNS-CB-UP), we actually consider two versions: one called LNS-CB-UP where no Lopt-conflict is generated, and another called LNS-CB-UP-Lopt where Lopt-conflicts are generated. For LNS-CB-IncrementalSAT (or shortly LNS-CB-Sat), we do not present the results obtained with the Lopt-conflicts due to space limitation reasons. For LNS-CB-OBDD, we do not use the temporary Lopt-conflicts as it would require (a) maintaining an OBDD containing only permanent TW-conflicts, and (b) making time-consuming conjunctions with the temporary Lopt-clauses that are still active at the current iteration.

Overall Performance. To quickly compare the baseline incomplete search algorithm (called LNS-noCB) and the versions using a CB, we first measured, for each solver and each instance, the average gap to the best known solution after five runs, each within 1 min. This gap g_s for solver s is defined by $g_s = 100 * (bk - bf_s)/bk$ where bf_s is the total reward of the best feasible solution found by s and bk is the best known objective value. Table 2 shows that for 1-minute time limit, using CB-UP globally improves the gaps (0.851% compared to 0.886% when using noCB), while using CB-UP-Lopt also generates competitive results. On the contrary, CB-Sat and CB-OBDD deteriorate the average gap (mean gaps equal to 1.739% and 1.418% respectively). Moreover, we also implemented a simple tabu list that prevents the algorithm from inserting (or removing) customers that were removed (or inserted) during the last k iterations. This tabu list made the LNS method highly effective for instances in the Solomon1 set, with an average gap of 0.083%. However, the average gaps obtained on other three sets are much larger, leading to a higher grand mean of the average gaps (2.332% for LNS-SimpleTabu, compared to 0.851% for LNS-UP).

To further analyze the results, each version of the solver is executed during 10 000 LNS iterations and the total time elapsed over each set is measured. Then, a speed-up rate compared to the noCB version is computed by $speedUp_s = 100 * (timeNoCB - timeWithCB_s)/timeNoCB$. Table 3 shows that the search process is accelerated with CB-UP and CB-UP-Lopt almost all the time, especially on the Cordeau instances where the speed-up reaches almost 50%. On the contrary, the search process is drastically slowed down with CB-Sat and CB-OBDD.

Table 2. Average gap (%) over 5 runs (maxCPUtime=60s, best average gaps in **bold**)

Instance set	Variants of CB in LNS					
	noCB	UP	UP-Lopt	SAT	OBDD	simpleTabu
Solomon1	1.093	1.093	1.304	1.492	1.315	**0.083**
Solomon2	0.416	0.387	**0.345**	0.607	0.497	4.097
Cordeau1	0.139	**0.078**	0.351	1.125	0.903	1.540
Cordeau2	1.898	**1.846**	1.900	3.729	2.958	2.119
Grand mean	0.886	**0.851**	0.977	1.739	1.418	2.332

Table 3. Speed-up (%) when solving during 10 000 LNS iterations

Instance set	Variant of CB in LNS			
	UP	UP-Lopt	SAT	OBDD
Solomon1	−8.83	−18.66	−2517.14	−646.14
Solomon2	**25.17**	**25.15**	−492.75	−163.62
Cordeau1	**48.66**	**47.04**	−2779.32	−2446.31
Cordeau2	**45.96**	**47.83**	−2092.35	−610.95

Slow Convergence with CB-SAT *and* CB-OBDD*.* Despite the rapidity of incremental solving with CryptoMiniSat, the results obtained show that the search process is slower for the LNS-CB-SAT version. The main reason for this is that there are numerous calls to $solve(\mathcal{F}, \mathcal{A} \cup \{l\})$, and each call must either find a full solution or prove that none exists.

As for CB-OBDD, while querying in OBDD is fast, the results are not as good as expected. Table 4 shows that the OBDDs obtained are globally compact given the number of conflicts. But the reordering operations performed to get such a compactness can take a lot of time: on some instances, CB-OBDD spends more than 60% of the CPU time for reordering the variables. Alternatively, it is challenging to heuristically compute in advance a good static variable ordering for the OBDDs, since we do not have the entire information about the conflicts when a static ordering must be defined. Meanwhile, we tested eight problem-dependent heuristics (e.g. ordering the selection variables depending on the rewards, the time windows, etc.), and as shown in Table 5, the best heuristics give poor results on some instances.

Better and Faster Search with CB-UP*.* Figure 4 details the evolution of the mean gap over each set of instances. Globally, we observe that LNS is boosted by CB-UP. In particular, for set Cordeau 1 involving many TW-conflicts, the search process converges much more quickly with the support of CB-UP. This is because more LNS iterations are performed thanks to the effectiveness of neighborhood pruning through CB-UP. The strength of CB-UP-LOPT is particularly visible over instance sets Cordeau 2 and Solomon 2. In these cases, even with very few TW-conflicts, the approximate Lopt-conflicts help guide the search towards other interesting search regions.

Table 4. Size of CB for each instance group (CPU time: 10 s)

Instance set	#OPTW nodes	#conflicts (average)	#OBDDnodes (average)	reorderingtime (%)
Solomon1	100	509.66	257.59	34.15
Solomon2	100	19.19	11.19	6.91
Cordeau1	48–288	109.10	303.50	67.73
Cordeau2	48–288	0.40	1.70	2.15

Table 5. Performance of the static and dynamic ordering strategies for OBDDs on two instances (pr01: 48 variables, best static order found = "increasing opening time"; pr06: 288 variables, best static order found = "decreasing rewards")

instance	LNS iteration	#conflicts	best-static-ordering		dynamic-ordering	
			#nodes	time(s)	#nodes	time(s)
pr01	1	0	1	0.0006	1	0.0012
	2	2	7	0.0013	5	0.0026
	3	8	36	0.0021	13	0.0041
	4	8	36	0.0030	12	0.0054
pr06	1	0	1	0.0885	1	0.4886
	2	55	69219	0.1803	477	2.5110
	3	80	6342191	19.1407	533	2.6108
	4	94	38250383	367.198	833	2.6422

7 Related Works

Incomplete search and SAT/CP were combined in Large Neighborhood Search [23], where a sequence of destroy-repair operations is performed on an incumbent solution. The destroy phase unassigns a subset S of the decision variables, while the repair phase can be delegated to a SAT/CP engine capable of quickly exploring all possible reassignments of S given the current partial assignment. Some authors also proposed to represent specific neighborhood structures using a tailored CP model and to translate the solutions found for this model into changes at the level of the global solution [22]. Others propose an efficient neighborhood exploration algorithm with the help of restricted decision diagrams [8]. In the same spirit, our CB is built to quickly detect inconsistent assignments at the selection level, therefore it can significantly reduce the neighborhood size to explore in the repair phase, but one difference is that we generate new conflicts during search and for the incomplete search process, CB only acts as a constraint propagation engine.

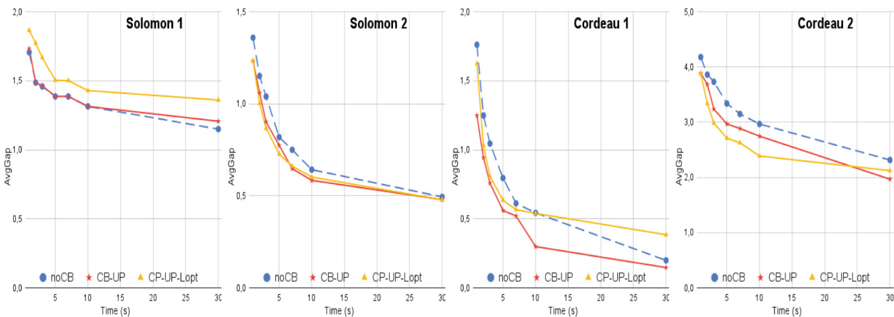

Fig. 4. Evolution of the average gaps for CB-UP and CB-UP-LOPT

Other hybrid approaches exploit the strengths of incomplete search and complete SAT/CP techniques at different search phases. As an illustration, in SAT, *Stochastic Local Search* (SLS) has been combined with DPLL or *Conflict Directed Clause Learning* (CDCL) [1,5,19]. For the SLS-CDCL version, the idea is that on one side, SLS can be run first to help CDCL have a heuristic for choosing variable values or to help CDCL update the activities of the variables, and on the other side CDCL can help SLS espace local optima. Another example is the composition of traditional CP search and Constraint-Based Local Search (CBLS [12]), where the two search approaches can exchange bounds, solutions, etc. In line with previous studies, inconsistency explanations generated at each iteration are stored in CB and then reused to help the search engine escape explored or invalid regions. In our case, by taking into account the current search state along with the clauses learned in the past iterations, CB may suggest mandatory assignments to quickly lead the search to promising regions.

Another technique uses inference methods such as unit propagation or constraint propagation, initially developed for complete search strategies, to speed up the neighborhood exploration during local search. One example following this line for SAT is the *unitWalk* algorithm [13,17]. At each iteration, it considers a complete variable assignment and performs a pass over this assignment to iteratively update the values of the variables with unit propagation. Compared to this work, one of the novelties in CB-UP is the decremental propagation aspect.

Last, the use of an external CB coupled with incomplete search can be compared with the use of a memory data structure in tabu search. On this point, instead of a simple list of forbidden local moves or forbidden variable assignments as in tabu search [9], CB memorizes logical formulas about the selection of nodes in a long-term way (possibly with some trashing when the size of CB becomes too large). CB is also equipped with efficient mechanisms to quickly reason about the formulas collected, instead of just reading explicit forbidden configurations. Another remark is that traditional tabu search is usually not recyclable i.e. the memory is reset at each resolution, while the time window conflicts stored in CB are easily recyclable for dynamic OPTWs where the reward associated with each node can change.

8 Conclusion and Perspectives

This paper presented a new hybrid optimization architecture combining an incomplete search process with clause generation techniques. Three generic clause basis managers were studied instead of just arbitrarily choosing a unique option, and the efficiency of the approach using unit propagation was demonstrated. One next step is to apply the approach to other problems like Team OPTW or flexible scheduling problems. Now that the generic clause bases are defined, the main effort to tackle a new problem is the definition of the problem-dependent clause generation procedure. Another perspective is to explore other clause basis managers (e.g. based on 0/1 linear programming and reduced-cost filtering), or knowledge bases covering pseudo-boolean constraints or cardinality constraints.

References

1. Audemard, G., Lagniez, J.M., Mazure, B., Saïs, L.: Integrating conflict driven clause learning to local search. In: 6th International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS 2009) (2009)
2. Audemard, G., Lagniez, J.-M., Simon, L.: Improving glucose for incremental SAT solving with assumptions: application to MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 309–317. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_23
3. Bellman, R.: Dynamic programming treatment of the travelling salesman problem. J. ACM (JACM) **9**(1), 61–63 (1962)
4. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. Comput. IEEE Trans. **100**(8), 677–691 (1986)
5. Crawford, J.: Solving satisfiability problems using a combination of systematic and local search. In: Second Challenge on Satisfiability Testing organized by Center for Discrete Mathematics and Computer Science of Rutgers University (1996)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002)
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
8. Gillard, X., Schaus, P.: Large neighborhood search with decision diagrams. In: International Joint Conference on Artificial Intelligence (2022)
9. Glover, F., Laguna, M.: Tabu search. In: Du, D.Z., Pardalos, P.M. (eds.) Handbook of Combinatorial Optimization, pp. 2093–2229. Springer, Boston (1998). https://doi.org/10.1007/978-1-4613-0303-9_33
10. Golden, B.L., Levy, L., Vohra, R.: The orienteering problem. Naval Res. Logistics (NRL) **34**(3), 307–318 (1987)
11. Gunawan, A., Lau, H.C., Vansteenwegen, P.: Orienteering problem: a survey of recent variants, solution approaches and applications. Eur. J. Oper. Res. **255**(2), 315–332 (2016)
12. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press, Cambridge (2005)
13. Hirsch, E., Kojevnikov, A.: UnitWalk: a new SAT solver that uses local search guided by unit clause elimination. Ann. Math. Artif. Intell. **43**, 91–111 (2002)
14. Hooker, J., Ottosson, G.: Logic-based Benders' decomposition. Math. Program. Ser. B **96**, 33–60 (2003)
15. Ignatiev, A., Semenov, A.: DPLL+ROBDD derivation applied to inversion of some cryptographic functions. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 76–89. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_8
16. Kantor, M.G., Rosenwein, M.B.: The orienteering problem with time windows. J. Oper. Res. Soc. **43**(6), 629–635 (1992)
17. Li, X.Y., Stallmann, M.F., Brglez, F.: QingTing: a fast SAT solver using local search and efficient unit propagation. In: Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003) (2003)
18. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 133–182. IOS Press (2021)
19. Mazure, B., Sais, L., Grégoire, É.: Boosting complete techniques thanks to local search methods. Ann. Math. Artif. Intell. **22**(3), 319–331 (1998)

20. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference, pp. 530–535 (2001)
21. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 242–255. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_19
22. Pesant, G., Gendreau, M.: A constraint programming framework for local search methods. J. Heuristics **5**(3), 255–279 (1999)
23. Pisinger, D., Ropke, S.: Large neighborhood search. In: Gendreau, M., Potvin, J.-Y. (eds.) Handbook of Metaheuristics. ISORMS, vol. 272, pp. 99–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91086-4_4
24. Prestwich, S.: The relation between complete and incomplete search. In: Blum, C., Aguilera, M.J.B., Roli, A., Sampels, M. (eds.) Hybrid Metaheuristics. Studies in Computational Intelligence, vol. 114, pp. 63–83. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78295-7_3
25. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD), pp. 42–47. IEEE (1993)
26. Savelsbergh, M.W.: Local search in routing problems with time windows. Ann. Oper. Res. **4**(1), 285–305 (1985)
27. Schmid, V., Ehmke, J.F.: An effective large neighborhood search for the team orienteering problem with time windows. In: ICCL 2017. LNCS, vol. 10572, pp. 3–18. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68496-3_1
28. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Solving RCPSP/max by lazy clause generation. J. Sched. **16**(3), 273–289 (2013)
29. Solomon, M.M.: Algorithms for the vehicle routing and scheduling problems with time window constraints. Oper. Res. **35**(2), 254–265 (1987)
30. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), pp. 244–257 (2009)
31. Vansteenwegen, P., Souffriau, W., Berghe, G.V., Van Oudheusden, D.: Iterated local search for the team orienteering problem with time windows. Comput. Oper. Res. **36**(12), 3281–3290 (2009)
32. Vansteenwegen, P., Souffriau, W., Van Oudheusden, D.: The orienteering problem: a survey. Eur. J. Oper. Res. **209**(1), 1–10 (2011)