Large Neighborhood Search with Decision Diagrams*

Xavier Gillard† , **Pierre Schaus**
Université Catholique de Louvain, BELGIUM
{xavier.gillard, pierre.schaus}@uclouvain.be

Abstract

Local search is a popular technique to solve combinatorial optimization problems efficiently. To escape local minima one generally uses metaheuristics or try to design large neighborhoods around the current best solution. A somewhat more black box approach consists in using an optimization solver to explore a large neighborhood. This is the large-neighborhood search (LNS) idea that we reuse in this work. We introduce a generic neighborhood exploration algorithm based on restricted decision diagrams (DD) constructed from the current best solution. We experiment DD-LNS on two sequencing problems: the traveling salesman problem with time windows (TSPTW) and a production planning problem (DLSP). Despite its simplicity, DD-LNS is competitive with the state-of-the-art MIP approach on DLSP. It is able to improve the best known solutions of some standard instances for TSPTW and even to prove the optimality of quite a few other instances.

1 Introduction

Local search is a popular approach to quickly obtain good solutions to combinatorial optimization problems [Hentenryck and Michel, 2009; Hoos and Stützle, 2004]. Unfortunately, a simple gradient descent based on simple perturbations such as 2-exchange moves can quickly get trapped into a local minima. Metaheuritics such as Tabu Search or Simulated Annealing can help escape from local minima. Another alternative is to explore larger neighborhoods to improve the current best solution. By exploring larger neighborhoods, the need for metaheuristics becomes less important, as the search is less myopic. Building larger neighborhoods, however, often requires a great deal of expertise. A successful one for vehicle routing problems is the Lin-Kernighan neighborhood [Lin and Kernighan, 1973] that generalizes the 2-OPT move to K-Opt. For some problems, exponentially sized neighborhoods can be explored in polynomial time using algorithms such as max-flow or path algorithms in graphs. We then speak of very large-scale neighborhood search [Ahuja *et al.*, 2002]. These complex neighborhoods are often problem specific and difficult to adapt. A more generic approach based on this idea of enlarging the neighborhood uses an optimization solver to explore the neighborhood [Shaw, 1998]. The main advantage is that the user expertise can be limited to the modeling of the problem rather than the design of complex move algorithms. This approach, alternates between two phases: the random relaxation of a fraction of the decision variables, and the assignment of those variables using the solver as a black-box tool. This large neighborhood approach has been used with great success using constraint programming (CP) solvers to solve scheduling [Laborie *et al.*, 2018] or vehicle routing problems [Jain and Van Hentenryck, 2011]. Similar ideas have also been used with Mixed Integer Programming (MIP) solvers under the name of local branching [Fischetti and Lodi, 2010]. Recently, some combinatorial optimization problems have been solved efficiently using DD approach [Bergman *et al.*, 2016]. Generic solvers for these approaches have also been developed [Gillard *et al.*, 2020]. It is thus a natural idea to attempt exploring large neighborhoods using DD solvers.

Contributions We show how to design an efficient DD based neighborhood exploration reusing the idea of restricted DD introduced in [Bergman *et al.*, 2016]. We demonstrate the performances of the approach experimentally on two constrained optimization problems: a discrete lot sizing and scheduling problem (DSLP) [Gent and Walsh, 1999, Problem 58] and the traveling salesman with time windows (TSPTW) [Gonzalez *et al.*, 2020]. Despite the simplicity and genericity of the approach, it appears to be competitive with the state-of-the-art MIP models for the DSLP [Pochet and Wolsey, 2006]. It is able to improve some best known solutions for the TSPTW [Gonzalez *et al.*, 2020] and to prove the optimality of some benchmark instances.

2 Discrete Optimization

A discrete optimization problem is a constraint *satisfaction* problem with an associated objective function to be minimized. The discrete optimization problem $\mathcal{P}$ is defined as $\min\{f(x) \mid x \in D \wedge C(x)\}$ where C is a set of constraints, $x = \langle x_0, \ldots, x_{n-1}\rangle$ is an assignment of values to vari-

*This paper is a preprint version of the article that was submitted and accepted at IJCAI-22 (www.ijcai.org) held in Vienna in July 2022. All source code and data are freely available online at: https://github.com/xgillard/ijcai_22_DDLNS.

†Contact Author

ables, each of which has an associated finite domain D_i s.t. $D = D_0 \times \cdots \times D_{n-1}$ from where the values are drawn. The function $f : D \to \mathbb{R}$ is the objective to minimize.

Among all feasible solutions $Sol(\mathcal{P}) \subseteq D$ (i.e. satisfying all constraints in C), we denote the optimal solution by x^*. That is, $x^* \in Sol(\mathcal{P})$ and $\forall x \in Sol(\mathcal{P}) : f(x^*) \leq f(x)$.

3 Dynamic Programming

Dynamic programming (DP) was introduced in the mid 50's by Bellman [Bellman, 1954]. This strategy is significantly popular and is at the heart of many classical algorithms (e.g., Dijkstra's algorithm [Cormen *et al.*, 2009, p.658] or Bellman-Ford's [Cormen *et al.*, 2009, p.651]).

Even though a dynamic program is often thought of in terms of recursion, it is also natural to consider it as a labeled transition system. In that case, the *DP model* of a given discrete optimization problem $\mathcal{P}$ consists of:

- a set $S = \{S_0, \ldots, S_n\}$ of state-spaces among which one distinguishes the *initial state* r, the *terminal state* t and the *infeasible state* $\bot$.
- a set τ of transition functions s.t. $\tau_i : S_i \times D_i \to S_{i+1}$ for $i = 0, \ldots, n-1$ taking the system from one state s^i to the next state s^{i+1} based on the value d assigned to variable x_i (or to $\bot$ if assigning $x_i = d$ is infeasible). These functions should never allow one to recover from infeasibility ($\tau_i(\bot, d) = \bot$ for any $d \in D_i$).
- a set of transition cost functions $h_i : S_i \times D_i \to \mathbb{R}$ representing the immediate reward of assigning some value $d \in D_i$ to the variable x_i for $i = 0, \ldots, n-1$.
- an initial value v_r.

On that basis, the objective function $f(x)$ of $\mathcal{P}$ can be formulated as follows:

$$\text{minimize } f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i)$$

subject to

$$s^{i+1} = \tau_i(s^i, x_i) \text{ for } i = 0, \ldots, n-1; x_i \in D_i \wedge C(x_i)$$

$$s^i \in S_i \text{ for } i = 0, \ldots, n$$

where $C(x_i)$ is a predicate that evaluates to $true$ when the partial assignment $\langle x_0, \ldots, x_i \rangle$ does not violate any constraint in C.

The appeal of such a formulation stems from its simplicity and its expressiveness, which allows it to effectively capture the problem structure. Moreover, this formulation naturally lends itself to a DD representation; in which case it represents an exact DD encoding the complete set $Sol(\mathcal{P})$.

4 Decision Diagrams

In all generality, a decision diagram (DD) is a kind of layered automaton s.t. a path between the source and a terminal node traverses one node from each layer of the graph. In this structure, the labels on the arcs are interpreted as the affectation of a value to a variable. Hence, the DD as a whole is seen as a compact encoding for a set of solutions to a given problem.

Formally, a DD $\mathcal{B}$ is a layered directed acyclic graph $\mathcal{B} = \langle n, U, A, l, d, v, \sigma \rangle$ where n is the number of variables from the encoded problem, U is a set of nodes; each of which is associated to some state $\sigma(u)$. The mapping $l : U \to \{0 \ldots n\}$ partitions the nodes from U in disjoint layers $L_0 \ldots L_n$ s.t. $L_i = \{u \in U : l(u) = i\}$ and the states of all the nodes belonging to the same layer pertain to the same DP-state-space ($\forall u \in L_i : \sigma(u) \in S_i$ for $i = 0, \ldots, n$). Also, it should be the case that no two distinct nodes of one same layer have the same state ($\forall u_1, u_2 \in L_i : u_1 \neq u_2 \implies \sigma(u_1) \neq \sigma(u_2)$, for $i = 0, \ldots, n$).

The set $A \subseteq U \times U$ from our formal model is a set of directed arcs connecting the nodes from U. Each such arc $a = (u_1, u_2)$ connects nodes from subsequent layers ($l(u_1) = l(u_2) - 1$) and should be regarded as the materialization of a branching decision about variable $x_{l(u_1)}$. This is why all arcs are annotated via the mappings $d : A \to D$ and $v : A \to \mathbb{R}$ which respectively associate a decision and value (weight) with the given arc.

Example 4.1. *An arc a connecting nodes $u_1 \in L_3$ to $u_2 \in L_4$, annotated with $d(a) = 6$ and $v(a) = 42$ should be understood as the assignment $x_3 = 6$ performed from state $\sigma(u_1)$. It should also be understood that $\tau_3(\sigma(u_1), 6) = \sigma(u_2)$ and the benefit of that assignment is $v(a) = h_3(\sigma(u_1), 6) = 42$.*

Because each r-t path describes an assignment satisfying $\mathcal{P}$, we will use $Sol(\mathcal{B})$ to denote the set of all the solutions encoded in the r-t paths of DD $\mathcal{B}$. Also, because unsatisfiability is irrecoverable, r-$\bot$ paths are typically omitted from DDs. It follows that a nice property from using a DD representation $\mathcal{B}$ for the DP formulation of a problem $\mathcal{P}$, is that finding x^* is as simple as finding the shortest r-t path in $\mathcal{B}$ (according to the weight v on arcs).

Exact-DD. For a given problem $\mathcal{P}$, an exact DD $\mathcal{B}$ is one that exactly encodes the solution set $Sol(\mathcal{B}) = Sol(\mathcal{P})$ of the problem $\mathcal{P}$. In other words, not only do all r-t paths encode valid solutions of $\mathcal{P}$, but no feasible solution is present in $Sol(\mathcal{P})$ and not in $\mathcal{B}$. An exact DD for $\mathcal{P}$ can be compiled in a top-down fashion[1]. This naturally follows from the above definition. To that end, one simply proceeds by a repeated unrolling of the transition relations until all variables are assigned as shown per Algorithm 1.

Algorithm 1 Top Down Compilation of an Exact DD

```
1:  Input: a DP-model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  L_0 ← {r}
3:  for i ∈ {0 ... n − 1}, u ∈ L_i, d ∈ D_i do
4:      u' ← τ_i(σ(u), d)
5:      if u' ≠ ⊥ then
6:          L_{i+1} ← L_{i+1} ∪ {u'}
7:          a ← (u, u')
8:          v(a) ← h_i(σ(u), d)
9:          d(a) ← d
10:         A ← A ∪ a
```

[1] An incremental refinement *a.k.a. construction by separation* procedure is detailed in [Cire, 2014, pp. 51–52] but we will not cover it here for the sake of conciseness.

Restricted DD In spite of the compactness of their encoding, the construction of DD suffers from a potentially exponential memory requirement in the worst case. Thus, using DDs to exactly encode the solution space of a problem is often intractable. This is why *bounded-size* approximation of the exact DD have been devised. While there exists several such approximations, this paper uses Restricted DD only. As shown in Algorithm 2, these are compiled by inserting a call to a width-bounding procedure (namely, *restrict*) to ensure that the width (the number $|L_i|$ of distinct nodes belonging to the L_i) of the current layer L_i does not exceed a given bound W. This procedure heuristically selects the most promising nodes and discards the others.

Formally, it is thus the case that given an exact DD $\mathcal{B}$ encoding the solutions of some given problem $\mathcal{P}$, and the restricted counterpart $\overline{\mathcal{B}}$ of $\mathcal{B}$; we have $Sol(\overline{\mathcal{B}}) \subseteq Sol(\mathcal{B})$. Consequently, the longest $r-t$ path in $\overline{\mathcal{B}}$ denotes a feasible solution of $\mathcal{P}$. And its length yields an upper bound on the optimal objective value.

Rough Lower Bound Recently, [Gillard *et al.*, 2021] proposed to use a *rough lower bound (RLB)* as a mean to speedup and tighten the bounds derived from bounded-width DDs. The intuition is that assuming the knowledge of an upper bound $\overline{v}$ on the optimal solution, and assuming that one is able to swiftly compute a rough lower bound $\underline{v_s}$ on the shortest $r-t$ path going through node s; s and all its descendants may be discarded whenever $\underline{v_s} \geq \overline{v}$ since all paths going through s are guaranteed not to improve $\overline{v}$.

Algorithm 2 shows how a restricted decision diagram is compiled taking an RLB into account. In practice, it suffices to include an additional check $rlb(u') \leq f(s^*)$ to discard node u' if its RLB is worse than the best known solution s^* (line 7). The rest of this paper refers to an invocation of Algorithm 2 as `CompileRestrictedDD(`$\mathcal{P}$`, `s^*`)` where $\mathcal{P}$ and s^* respectively denote the (sub-)problem to consider and its current best known solution.

Algorithm 2 Top Down Compilation of a Restricted DD

```
1:  Input: a DP-model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  Input: s* ← the best known solution or none.
3:  L_0 ← {r}
4:  for i ∈ {0 ... n − 1} do
5:      for u ∈ L_i, d ∈ D_i do
6:          u' ← τ_i(σ(u), d)
7:          if u' ≠ ⊥ ∧ rlb(u') ≤ f(s*) then
8:              L_{i+1} ← L_{i+1} ∪ {u'}
9:              a ← (u, u')
10:             v(a) ← h_i(σ(u), d)
11:             d(a) ← d
12:             A ← A ∪ a
13:     if |L_{i+1}| > W then
14:         L_{i+1} ← restrict(L_{i+1})
```

5 Large Neighborhood Search

As explained in the introduction, LNS is an incomplete optimization method that aims at being able to escape local minima while avoiding the need for the practitioner to devise specialized metaheuristics. To this ends, LNS attempts to find a balance between intensification (apply advanced inference algorithms to explore promising neighborhoods) and diversification (explore different neighborhoods). This is why starting from an initial solution s^*, LNS alternates between a relaxation phase and an reoptimization phase. During the relaxation phase, decisions made in s^* are challenged for a small fraction of the variables. Then, during the reoptimization, a solver operates a "black box" resolution of the remaining (sub-)problem. Whenever a solution s'^* improving the best known objective is discovered ($f(s'^*) < f(s^*)$); the incumbent best solution is updated.

Our Approach This paper proposes to use restricted DDs as a means to explore sets of solutions in a large neighborhood of s^*. Algorithm 3 shows how this is done in practice. Similar to vanilla LNS, our method must strike a balance between intensification and diversification. In our case, the intensification target is achieved through the compilation of a restricted DD (Algorithm 3 lines 10-11).

Algorithm 3 LNS with Decision Diagrams

```
1:  Input: a DP-model P = ⟨S, r, t, ⊥, v_r, τ, h⟩
2:  Input: s* ← the best known solution or none.
3:  MaxDepth ← |S| − 2
4:  d ← MaxDepth
5:  while end criterion not met do
6:      r' ← r
7:      if s* ≠ none then
8:          r' ← s*_d
9:      P' ← ⟨S, r', t, ⊥, v(r'), τ, h⟩
10:     neigborhood ← CompileRestrictedDD(P', s*)
11:     neighbor ← ShortestPath(neigborhood)
12:     if f(neighbor) < f(s*) then
13:         s* ← neighbor
14:         d ← MaxDepth
15:     else if d = 0 then
16:         d ← MaxDepth
17:     else
18:         d ← d − 1
```

In our method, three mechanisms are relevant to intensification. The first one is the (optional) use of an RLB procedure to discard nodes that cannot lead to an objective improvement. The second and third mechanisms follow from the behavior of the restriction procedure. As shown per Algorithm 4, one initially partitions the nodes of a given layer L_i between those nodes which *must* be kept in the layer, and the others (Algorithm 4 lines 5-11). This decision is based on the $mustKeep$ predicate (Definition 5.1) which states that a node n from the i^{th} layer must be kept if the value associated to variable x_i on the best $r-n$ path (denoted $p^*_{r-n}(i)$) is the same as the value of x_i in s^* (denoted $s^*(i)$). This guarantees that $s^* \in Sol(neighborhood)$ and hence that $neighborhood$ is an actual neighborhood of s^* (Algorithm 3 line 10).

Definition 5.1.

$$mustKeep(n, s^*, i) \iff p^*_{r-n}(i) = s^*(i)$$

The last of our intensification mechanism consists of the node selection heuristic which is used to select the nodes remaining in a layer after restriction. Algorithm 4 shows at lines 12-13 that the candidate nodes are filtered to keep only the best nodes according to their RLB.

There are two mechanisms at play in our method to ensure a fair amount of diversification. The first one consists of selecting a different root for the compilation of the restricted DDs. This is done is a systematic manner, optimistically starting with a node at the bottom of the DD which yielded the best solution; progressing towards the actual root of the problem (Algorithm 3 lines 6-9, 14-18). The second mechanism in use consists in the introduction of some randomness during a layer restriction. As shown in Algorithm 4, any node not satisfying the $mustKeep$ predicate might still be forced into the restricted layer with a small probability (line 7).

Algorithm 4 Restrict Procedure

```
1: Input: L_i : the layer that needs to be restricted
2: Input: s* : the best known solution, or none
3: Input: W : maximum layer width
4: Input: p : a small probability (e.g. 10%)
5: frontier ← 0
6: for k ∈ {0..|L_i|} do
7:    if mustKeep(L_i[k], s*, i) ∨ random() ≤ p then
8:       swap(L_i, frontier, k)
9:       frontier ← 1 + frontier
10: keep ← nodes[0..frontier[
11: candidates ← nodes[frontier..|nodes|[
12: sort(candidates based on their RLB)
13: truncate(candidates, max(0, W − |keep|))
14: L_i ← concat(keep, candidates)
```

Benefits Our approach offers several benefits. DDs leverage their underlying DP model as a mean to explore the neighborhood of a given best solution. Moreover, as opposed to vanilla LNS, our approach is sometimes able to *prove the optimality* of the instances it solves. This is usually only possible when using an exact method such as MIP or Branch-and-Bound [Bergman *et al.*, 2016]. Indeed, our method proceeds by generating sets of complete solutions at each iteration. Still, because the value of the best incumbent solution is improving over time, so is the pruning power of the RLB used when compiling the DD. From there, it follows that sometimes the pruning power of the RLB is sufficient to let the DD compile without requiring any restriction (when $d = 0$ and no layer ever exceeds the maximum width). In that event, the resulting DD is an exact DD which proves the optimality of the best solution it contains. Naturally, this capability stems from a tradeoff between the pruning power of RLB and the maximum layer width of the DD that are used. Therefore, the algorithm won't be able to always achieve a formal proof of optimality. Predicting whether it will succeed in delivering such a proof is undecidable. Still, we believe that the possibility for a metaheuristic approach to sometimes give a proof of optimality is an appreciable feature.

Another benefit of using our method comes from the configurable aspect of the DD compilation. Which means one can choose how wide the DD is allowed to be. Thereby arbitraging a balance between diversification and intensification.

6 Experimental Study

In order to evaluate the effectiveness of DD-LNS, we considered two sequencing problems: a discrete lot sizing and scheduling problem (DSLP) and the traveling salesman problem with time windows (TSPTW).

DSLP is a multi-item capacited lot sizing problem detailed in CSPLib [Gent and Walsh, 1999, Problem 58] and studied in depth in [Pochet and Wolsey, 2006]. It is a production planning problem where one item needs to be produced on a machine at each time slot to meet demands at minimal stocking and changeover costs. It is characterized by a 5-tuple $\langle \mathcal{I}, \mathcal{H}, \mathcal{S}, \mathcal{C}, \mathcal{Q} \rangle$ where:

- $\mathcal{I} = \{0, \ldots, n-1\}$ is the set of item types to produce,
- $\mathcal{H}$ is the problem time horizon,
- $\mathcal{S}$ is a stocking cost vector where $\mathcal{S}_i$ is the cost of stocking one unit of type i during one period,
- $\mathcal{C}$ is a changeover cost matrix where $\mathcal{C}_{i,j}$ is the cost of changing the machine configuration from producing item i to producing item j, and
- $\mathcal{Q}$ is a vector of demands per item. Given a time period $0 \leq t < \mathcal{H}$ and an item $i \in \mathcal{I}$, $\mathcal{Q}_t^i$ is used to denote the number of items of type i to deliver at time t. Without loss of generality, the next sections assume normalized demands. That is, $\mathcal{Q}_t^i \in \{0, 1\}\, \forall t, i$.

A MIP Formulation is given by equations (1) – (6). In line with the nomenclature from [Pochet and Wolsey, 2006], this model will be referred to as PIG-A-1.

$$\textbf{minimize} \sum_{i,j,t} \mathcal{C}_{i,j} \mathbf{c}_{i,j}^t + \sum_{i,t} \mathcal{S}_i \mathbf{s}_i^t \tag{1}$$

subject to

$$\mathbf{s}_i^0 = 0 \;\; \forall i \in \mathcal{I} \tag{2}$$

$$\mathbf{x}_i^t + \mathbf{s}_i^{t-1} = \mathcal{Q}_t^i + \mathbf{s}_i^t \;\; \forall i \in \mathcal{I}; 0 \leq t < \mathcal{H} \tag{3}$$

$$\mathbf{x}_i^t \leq \mathbf{y}_i^t \;\; \forall i \in \mathcal{I}, 0 \leq t < \mathcal{H} \tag{4}$$

$$\sum_{i,t} \mathbf{y}_i^t = 1 \;\; \forall i \in \mathcal{I}, 0 \leq t < \mathcal{H} \tag{5}$$

$$\mathbf{c}_{i,j}^t \geq \mathbf{y}_i^{t-1} + \mathbf{y}_j^t - 1 \;\; \forall i, j \in \mathcal{I}; 0 \leq t < \mathcal{H} \tag{6}$$

where $\mathbf{x}_i^t$ is a binary production variable (1 when item i is produced at time t, otherwise 0). $\mathbf{y}_i^t$ is a binary setup variable (1 iff machine is ready to produce i at time t). $\mathbf{c}_{i,j}^t$ is a binary changeover variable (1 iff configuration changed from i to j at time t). $\mathbf{s}_i^t$ is an integer stocking variable counting the number of items of type i stored at time t.

In this model, (1) is the objective function to minimize. (2) is a constraint imposing that the stock of every item is empty at startup. Equation (3) is a *conservation* constraint stating that when an item is produced, it is either delivered or stocked for later delivery. Constraint (4) forces the consistency between the production and machine configuration variables.

(5) is a *capacity* constraint stating that only 1 unit of one item is produced at each time. Finally, (6) is a constraint that enforces the consistency between the machine configuration variables ($\mathbf{y}_i^{t-1}, \mathbf{y}_j^t$) and the changeover variables ($\mathbf{c}_{i,j}^t$).

DP Model [2] It helps to define first from the input data $\mathcal{T}_t^i$ that gives the *previous* demand time for a given item i and time $0 \leq t \leq \mathcal{H}$.

$$\mathcal{T}_0^i = -1 \;\Big|\; \mathcal{T}_t^i = \begin{cases} t-1 & \text{if } \mathcal{Q}_i^{t-1} > 0 \\ \mathcal{T}_{t-1}^i & \text{otherwise} \end{cases}$$

The elements of a DLSP DP model are the following:

- a state $s^t \in \mathcal{S}_{\mathcal{H}-t}$ is a tuple $\langle k, u \rangle$ where k denotes the type produced at time $t+1$, and u is a vector comprising the previous delivery date for each item. In particular, we have $r = \left\langle -1, \left(\mathcal{T}_{\mathcal{H}}^0, \mathcal{T}_{\mathcal{H}}^1, \ldots, \mathcal{T}_{\mathcal{H}}^{n-1}\right)\right\rangle$
- $\tau_t(\langle k, u \rangle, d) = \begin{cases} \left\langle d, \left(u_0, \ldots u_{d-1}, \mathcal{T}_{u_d}^d, u_{d+1}, \ldots u_{n-1}\right)\right\rangle & \\ & \text{when } u_d \geq t \\ \bot & \text{otherwise} \end{cases}$
- $h_t(\langle k, u \rangle, d) = \begin{cases} \mathcal{S}_d \cdot (u_d - t) & \text{when } k = -1 \\ \mathcal{C}_{k,d} + \mathcal{S}_d \cdot (u_d - t) & \text{otherwise} \end{cases}$
- $v_r = 0$.

Rough Lower Bound Because DLSP is a simple case of Wagner-Whitin (WW) [Pochet and Wolsey, 2006] in the absence of changeover costs, an RLB for some state $s^t = \langle k, u \rangle \in \mathcal{S}_{\mathcal{H}-t}$ is given by the WW cost of s^t plus the total weight of a minimum spanning tree over the changeover costs of items i s.t. $u_i \geq 0$.

TSPTW TSPTW is a popular variant of the TSP where the salesman's customers must be visited within given time windows. This problem is notoriously hard to solve considering that even finding a feasible solution was proved NP-complete [Savelsbergh, 1985]. Formally, TSPTW is characterized by N a number of customers to visit, $\mathcal{D}$ a square matrix s.t. $\mathcal{D}_{i,j}$ is the distance between customers i and j; $\mathcal{H}$ the considered time horizon and $\mathcal{TW}$ is a vector of time windows s.t. $\mathcal{TW}_i = (e_i, l_i)$ where e_i is the earliest time when the salesman can visit i and l_i the latest.

CP Model A declarative Minizinc [**?**] constraint programming model of this problem is given next. The decision variable x_i defines the visited customer in i^{th} position of the tour. The auxiliary variables a_i represent the time when the salesman visits the i^{th} customer of the tour. The constraints ensure i) that the salesman's tour starts and ends at the depot ii) each city is visited exactly once iii) the time window constraints and iv) the salesman cannot travel faster than specified in the distance matrix between two consecutive visits but is allowed to wait until the beginning of the time window. Finally, the travel time objective is minimized.

[2] To the best of our knowledge, this problem has not been solved with DP before. The introduced model is thus also a contribution.

```
/* Make it a tour start/ending at city 0 */
constraint (x[0] = 0 /\ x[N] = 0 /\ a[0] = 0);
constraint alldifferent_except_0(x);
/* Enforce time windows */
constraint forall(i in 0..N)(Earliest[x[i]] <= a[x[i]]);
constraint forall(i in 0..N)(a[x[i]] <= Latest[x[i]]);
constraint forall(i in 0..N-1)(
  a[x[i+1]] >= a[x[i]] + Distance[x[i],x[i+1]]
);
/* Travel Objective */
int: travel = sum(i in 0..N-1)(Distance[x[i],x[i+1]]);
solve minimize travel;
```

DP Model There is a long history of DP models for the TSPTW [Savelsbergh, 1985; Dumas *et al.*, 1995; Mingozzi *et al.*, 1997]. In the context of this experimental study, we model it as follows:

- a state $s^t \in \mathcal{S}_t$ is a tuple $\langle t, c, \rho \rangle$ where t denotes the current time (a.k.a. makespan), c identifies the current location and ρ is the set of customers remaining to be visited. In particular, we have $r = \langle 0, 0, \{1 \ldots N-1\} \rangle$.
- $\tau_t(\langle t, c, \rho \rangle, d) = \begin{cases} \langle \max(e_d, t + \mathcal{D}_{c,d}),\ d,\ \rho \setminus \{d\} \rangle & \\ & \text{when } l_d \geq t + \mathcal{D}_{c,d} \\ \bot & \text{otherwise} \end{cases}$
- $h_t(\langle t, c, \rho \rangle, d) = \mathcal{D}_{c,d}$
- $v_r = 0$

Rough Lower Bound A simple yet effective rough lower bound for some state $s^t = \langle t, c, \rho \rangle$ is given by the sum of $\min\{D_{c,o} \mid o \in \rho\}$, the cost of a minimum spanning tree over the distance between the nodes in ρ, and $\min\{D_{o,0} \mid o \in \rho \wedge D_{o,0} \leq l_0\}$. Intuitively, these three terms respectively represent the shortest distance between the current position and any remaining customer; an estimation of the shortest travel distance between the remaining customers and the shortest distance between a customer potentially preceding the depot and the depot itself.

6.1 Experimental Setup

All our experiments were performed on the same physical machine equipped with two Intel(R) Xeon(R) CPU E5-2687W v3 and 128Gb of RAM. On that machine, each considered solver was given a 10 minutes timespan using a single thread and a maximum memory quota of 2Gb in order to solve each instance. For the evaluation of DD-LNS, all experiments use a generic framework in which we plugged the DP models given above.

DLSP We compared the performance of DD-LNS against the state-of-the-art MIP model (PIG-A-3) from [Pochet and Wolsey, 2006]. Because of the simplicity of our DP model, and because PIG-A-3 was tuned by MIP experts over a decade; we also included the simpler MIP models PIG-A-1 and PIG-A-2 in our comparison. These should give an idea of what a practitioner might reasonably expect when creating a model for the DLSP. All MIP models originate from [Pochet and Wolsey, 2006] and are written using FICO Xpress Mosel v8.11. Our experiments include 500 generated instances

Method	W	Optimum	1% Gap
PIG-A-1	N.A.	35	232
PIG-A-2	N.A.	97	226
PIG-A-3	N.A.	475	499
DD-LNS	10	167	449
DD-LNS	100	276	489
DD-LNS	1000	368	490

Table 1: Number of DLSP instances for which the optimum solution has been found and for which the best solution found is within 1% of the global optimum.

which could not be solved with either pure dynamic programming or branch-and-bound mdd [Bergman *et al.*, 2016; Gillard *et al.*, 2021]. These 500 instances have $|\mathcal{I}| = 10$, $\mathcal{H} \in \{50, 100\}$ and their $\mathcal{S}_i$ and $\mathcal{C}_{i,j}$ values range between 10 and 50.

TSPTW We compared the performance of DD-LNS against an implementation of the CP model from section 6 in Choco 4.10.6 [3] using a LNS that re-optimizes a small sequence[4] of 5 decision variables $x_i \dots x_{i+4}$ randomly selected at each restart. Our experiments cover the 467 instances of the benchmark suites which are usually used to assess the efficiency of new TSPTW solvers.

Results Table 1 shows the number of DLSP instances for which the best solution found by each solver matches the best known solution. It also shows the number of instances where the best solution found was within 1% of the global optimum ($\frac{found - best\_known}{best\_known} \leq 1\%$). From this table, it appears that the combination of LNS with Decision Diagrams is very efficient at finding good solutions. Indeed, this method outperforms the PIG-A-1 and PIG-A-2 models in all situations; even with a maximum layer width as small as 10 nodes. Furthermore, in spite of the simplicity of its underlying DP model, our DD-LNS approach fares comparably to the much more advanced PIG-A-3 models.

Because the TSPTW satisfiability is NP-complete, and in order to establish a fair comparison between CP-LNS and DD-LNS, we bootstrapped the problem resolution of all solvers with an initial feasible solution computed off-line (the same for both CP and DD-LNS). These initial solutions have been computed using a variant of [Da Silva and Urrutia, 2010]. Table 2 shows the number of TSPTW instances for which the best solution found by each solver matches the published best known solution. It also shows the number of instances where the best solution found was within 1 % of the overall best known solution. This table shows that both methods are highly efficient at finding good solutions to the TSPTW; DD-LNS having a slight edge over CP-LNS. During this phase of the experiments, DD-LNS proved 105 instances optimal and identified 75 new solutions with an objective value matching that of the published best known solution. In the same conditions, Choco identified 35 new solutions having an objective value equal to the best known. It could not prove the optimality of any instance.

[3] https://choco-solver.org/

[4] Several alternative relaxation schemes and parameters were experimented and this one gave the best results.

Method	W	Optimum	1% Gap
CP-LNS	N.A.	144	183
DD-LNS	10	197	245
DD-LNS	100	217	260
DD-LNS	1000	216	248

Table 2: Number of TSPTW instances for which the optimum solution has been found and for which the best solution found is within 1% of the overall best known value

As a mean to assess the effectiveness of these methods at optimizing TSPTW independently of the initial solution, we repeated the experiment; this time initializing the resolution with the published best known solution of each instance. This again proved the high efficiency of our method. Indeed it identified new solutions improving the objective value of standard benchmark instances. In practice, DD-LNS was able to identify 8 improving solutions in two benchmarks suites (AFG and OhlmannThomas).

An interesting general observation to make about our experiments stems from the fact that the maximum width W of the compiled DDs provides an easy means to tune the diversification level. Indeed, both Table 1 and Table 2 show that increasing the maximum layer width W improves the solver performance before it starts hampering it.

7 Related Work

Our approach can be considered as a hybridization of DD for optimization (DDO), beam search, LNS, and the phase saving heuristic which is customarily used in SAT solvers [Pipatsrisawat and Darwiche, 2007]. Combination of some of these ingredients have recently been proposed. But, to the best of our knowledge, no approach ever combined all of them. For instance, [López-Ibáñez and Blum, 2010] hybridized beam search with ant-colony optimization (ACO) in order to solve the TSPTW. As opposed to our method, is was driven by an ACO component rather than DP.

Articles [Demirović *et al.*, 2018] and [Björdal *et al.*, 2020] pursue a similar goal. They try to blend constraint programming, phase saving and LNS to solve hard combinatorial problems but focusing on propagators while this work relies on a dynamic programming formulation of the problem only.

8 Conclusions

We introduced and evaluated a method combining large neighborhood search with decision diagrams to solve hard combinatorial optimization problems having a dynamic programming formulation. Its simplicity and good performances (experimented on two problems) might be appreciable for practitioners. In particular, when one has to repeatedly take good decisions quickly; as would for instance be the case when adapting a production schedule based on an ever-evolving order book.

References

[Ahuja *et al.*, 2002] Ravindra K Ahuja, Özlem Ergun, James B Orlin, and Abraham P Punnen. A survey of very

large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[Bellman, 1954] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 11 1954.

[Bergman *et al.*, 2016] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and J. N. Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

[Björdal *et al.*, 2020] Gustav Björdal, Pierre Flener, Justin Pearson, Peter J Stuckey, and Guido Tack. Solving satisfaction problems using large-neighbourhood search. In *International Conference on Principles and Practice of Constraint Programming*, pages 55–71. Springer, 2020.

[Cire, 2014] Andre A Cire. *Decision Diagrams for Optimization*. PhD thesis, Carnegie Mellon University Tepper School of Business, 2014.

[Cormen *et al.*, 2009] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[Da Silva and Urrutia, 2010] Rodrigo Ferreira Da Silva and Sebastián Urrutia. A general vns heuristic for the traveling salesman problem with time windows, 2010.

[Demirović *et al.*, 2018] Emir Demirović, Geoffrey Chu, and Peter J Stuckey. Solution-based phase saving for cp: A value-selection heuristic to simulate local search behavior in complete solvers. In *International Conference on Principles and Practice of Constraint Programming*, pages 99–108. Springer, 2018.

[Dumas *et al.*, 1995] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

[Fischetti and Lodi, 2010] Matteo Fischetti and Andrea Lodi. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[Gent and Walsh, 1999] Ian P Gent and Toby Walsh. CSPLib: a benchmark library for constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–481. Springer, 1999.

[Gillard *et al.*, 2020] X. Gillard, P. Schaus, and V. Coppé. Ddo, a generic and efficient framework for mdd-based optimization. International Joint Conference on Artificial Intelligence (IJCAI-20); DEMO track, 2020.

[Gillard *et al.*, 2021] X. Gillard, Coppé V., P. Schaus, and Ciré A. Improving the filtering of branch-and-bound mdd solvers, 2021.

[Gonzalez *et al.*, 2020] Jaime E Gonzalez, Andre A Cire, Andrea Lodi, and Louis-Martin Rousseau. Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem. *Constraints*, pages 1–24, 2020.

[Hentenryck and Michel, 2009] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. 2009.

[Hoos and Stützle, 2004] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[Jain and Van Hentenryck, 2011] Siddhartha Jain and Pascal Van Hentenryck. Large neighborhood search for dial-a-ride problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 400–413. Springer, 2011.

[Laborie *et al.*, 2018] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

[Lin and Kernighan, 1973] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[López-Ibáñez and Blum, 2010] Manuel López-Ibáñez and Christian Blum. Beam-aco for the travelling salesman problem with time windows. *Computers & operations research*, 37(9):1570–1583, 2010.

[Mingozzi *et al.*, 1997] Aristide Mingozzi, Lucio Bianco, and Salvatore Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations research*, 45(3):365–377, 1997.

[Pipatsrisawat and Darwiche, 2007] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *International conference on theory and applications of satisfiability testing*, pages 294–299. Springer, 2007.

[Pochet and Wolsey, 2006] Yves Pochet and Laurence A. Wolsey. *Production Planning by Mixed Integer Programming*. Springer, 2006.

[Savelsbergh, 1985] Martin WP Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

[Shaw, 1998] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.