Ddo, un cadre générique et performant pour l'optimisation à base de diagrammes de décision

Xavier Gillard* **Pierre Schaus** **Vianney Coppé**
UCLouvain, 1348 Louvain-la-Neuve, Belgique
{xavier.gillard, pierre.schaus, vianney.coppe}@uclouvain.be

Résumé

Cet article est un combine et résume les articles suivants : [6, 5]. Il présente *ddo*, une bibliothèque générique et performante pour résoudre des problèmes d'optimisation à l'aide de diagrammes de décision. Pour ce faire, notre bibliothèque implémente l'approche par "branchements et bornage" proposée par [3] afin de trouver la solution optimale de programmes dynamiques. Grâce à notre bibliothèque nous avons non seulement été en mesure d'implémenter des solveurs pour les problèmes MISP, MCP, MAX2SAT et TSPTW. Aussi, *ddo* tire parti des capacités de calcul en parallèle du matériel sur lequel il s'exécute sans que le développeur n'ait d'effort particulier à fournir. Il en résulte que les solveurs implémentés avec *ddo* sont hautement performants. En particulier, nos expériences montrent que *ddo* surclasse Gurobi pour MCP. Ddo et nos solveurs d'exemples sont publiés sous une licence libre et leur code source est accessible en ligne [1].

1 Introduction

Les diagrammes de décision multivalués (MDD) sont une généralisation des diagrammes de décision binaires (BDD), lesquels sont utilisés depuis longtemps, e.a. parce qu'ils permettent de réaliser le model checking de systèmes complexes[4]. Plus récemment, ces modèles graphiques ont attiré l'attention de chercheurs dans les communautés PC et RO. La popularité des diagrammes de décision (DD) vient de leur capacité à encoder de larges espaces d'états de façon très compacte. C'est pourquoi ils sont entre autre utilisés dans le cas de la contrainte `Table`[7, 8]. Cet intérêt accru pour les DD a donné naissance à l'*optimisation à base de DD* (DDO) [2]. L'objectif de cette technique est de résoudre efficacement des problèmes d'optimisation combinatoires en exploitant la structure de ceux-ci au travers de DDs. Cet article s'inscrit dans cette ligne et poursuit un double objectif : d'une part, faire connaitre la technique au plus grand nombre. Et d'autre part, faciliter l'intégration de celle-ci avec d'autres solveurs et outils grâce à une bibliothèque générique et performante.

2 Les fondements

Un problème d'optimisation discrète est avant tout un problème de satisfaction de contraintes auquel une fonction d'objectif est associée. Parmi ces problèmes, certains ont une structure de *sous-problème* optimal qui leur permet d'être formulés comme des programmes dynamiques (DP). Bien que les modèles DP soient typiquement envisagés sous l'angle de la récursion, il est aussi naturel de les considérer comme des systèmes à transitions d'états. Auquel cas, un modèle DP consiste de : (a) un espace de solutions défini par les variables du problème et leurs domaines ; (b) un état initial, (c) une valeur initiale ; (d) une fonction de transition et (e) une fonction de coût de transition.

Au coeur de DDO, on trouve l'idée selon laquelle un système de transition DP se matérialise facilement sous la forme d'un diagramme de décision (réduit). Toutefois, bien qu'ils soient compacts, la construction de ces DD peut requérir une quantité de mémoire (et de temps) exponentielle. C'est pourquoi il est impossible d'encoder exactement l'espace d'états pour des instances de problèmes réelles. Pour palier à cela, DDO utilise des DDs ayant une taille maximum bornée. Ceux-ci permettent de fournir deux types d'approximations pour le problème à résoudre. En supposant une fonction d'objectif à maximiser, les DD *restreints* et *relâchés* [1] permettent respectivement de dériver une borne inférieure et supérieure pour le problème à résoudre.

Compiler un DD restreint à partir d'une formulation DP est assez simple : il suffit de s'assurer que la lar-

*Papier doctorant : Xavier Gillard est auteur principal.
1. https ://github.com/xgillard/ddo

geur des niveaux du DD soit limitée en supprimant les nœuds les moins prometteurs de chaque niveau. Ceci supprime un certain nombre de solutions du DD, mais n'introduit jamais de non-solutions dans celui-ci. La compilation d'un DD relâché est un peu différente car elle nécessite qu'on lui fournisse une relaxation permettant la *fusionner* des noeuds surnuméraires. C'est pourquoi, lorsqu'on souhaite résoudre un problème avec DDO, il est nécessaire de donner à la fois un modèle DP et une relaxation du problème.

3 La bibliothèque *ddo*

C'est là tout ce dont notre bibliothèque *ddo* a besoin pour résoudre un problème automatiquement et efficacement : la définition de ce problème et une relaxation. Naturellement, *ddo* permet en outre de guider la résolution via des heuristiques propres au problème. Mais leur emploi n'est pas *requis*.

Nous illustrons notre propos via un exemple minimaliste mais complet. Celui-ci montre comment modéliser et résoudre un problème de sac à dos avec *ddo*. Le Listing 1, montre bien à quel point le modèle *ddo* ressemble aux abstractions mathématiques évoquées dans la section précédente. En particulier, l'implémentation du trait `Problem<usize>` par la structure `Sac` décrit la formulation DP du problème de sac à dos dont l'état consiste en un entier non signé (`usize`). L'espace de solution du problème (a) est caractérisé par les méthodes `nb_vars()` et `domain_of()` (lignes 8–15). De même les quatre autres éléments constitutifs d'un modèle DP (état init. (b), valeur init. (c), fonction de transition (d) et fonction de coût (e)) sont tous impléméntés par leurs fonctions éponymes (lignes 16–29). Aussi, la structure `SacRelax` qui implémente le trait `Relaxation<usize>` montre un exemple de fusion d'états (lines 36–45). Dans notre exemple, le nouvel état relâché est obtenu en gardant la capacité maximum des états à fusionner et il n'est pas nécessaire de modifier le poids des arcs entrants du noeud ainsi relâché.

```
1  #[derive(Clone, Debug)]
2  struct Sac {
3      capacite: usize,
4      profit: Vec<usize>,
5      poids: Vec<usize>
6  }
7  impl Problem<usize> for Sac {
8      fn nb_vars(&self) -> usize {
9          self.profit.len()
10     }
11     fn domain_of<'a>(&self,
12         state: &'a usize,
13         var: Variable) -> Domain<'a> {
14         vec![0, 1].into()
15     }
16     fn initial_state(&self) -> usize {
17         self.capacite
18     }
19     fn initial_value(&self) -> isize {
20         0
21     }
22     fn transition(&self, state: &usize,
23         vars: &VarSet, d: Decision) -> usize {
24         let var = d.variable.id();
25         state - self.poids[var] * d.value as usize
26     }
27     fn transition_cost(&self, state: &usize,
28         vars: &VarSet, d: Decision) -> isize {
29         let var = d.variable.id();
30         self.profit[var] as isize * d.value
31     }
32 }
33 #[derive(Clone, Copy)]
34 struct SacRelax;
35 impl Relaxation<usize> for SacRelax {
36     fn merge_states(&self,
37         states: &mut dyn Iterator<Item=&usize>)
38      -> usize {
39         states.copied().max().unwrap_or(0)
40     }
41     fn relax_edge(&self, src: &usize, dst: &usize,
42         relaxed: &usize, d: Decision, cost: isize)
43      -> isize {
44         cost
45     }
46 }
47 fn main() {
48     let problem = Sac {/*omis*/};
49     let config = config_builder(&problem, SacRelax)
50       .build();
51     let mdd = DeepMDD::from(config);
52     let mut solver = ParallelSolver::new(mdd);
53     solver.maximize();
54 }
```

Listing 1 – Detailed example

Enfin, les lignes 48–53 du Listing 1 montrent comment instancier le solveur et l'utiliser pour résoudre une instance du problème de sac à dos binaire en utilisant tous les fils d'exécution matériels de la machine.

4 Résultats expérimentaux et conclusion

Nous voudrions clôturer notre présentation de *ddo* en pointant certains résultats expérimentaux très encourageants. En effet, celui-ci montre que malgré que *ddo* soit entièrement générique il parvient à être extrêmement performant. En l'occurrence, la Figure 1 montre que lors de nos expériences, notre solveur a su résoudre les 265 instances de MCP testées en un peu moins de 800 secondes sur 24 fils d'exécution alors que dans les mêmes conditions, Gurobi 9.0.2 n'est pas parvenu à toutes les résoudre en 30 minutes.

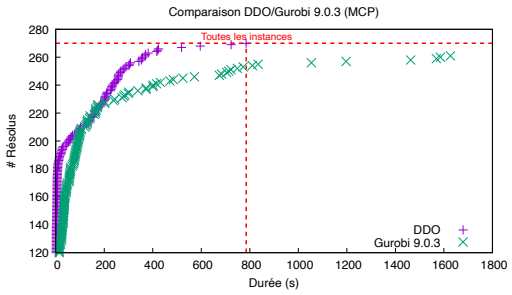

FIGURE 1 – Comparaison *ddo*/Gurobi

Références

[1] H. R. Andersen, T. Hadzic, J. N. Hooker et P. Tiedemann : A constraint store based on multivalued decision diagrams. *In* Christian Bessière, éditeur : *Principles and Practice of Constraint Programming*, volume 4741 de *LNCS*, pages 118–132. Springer, 2007.

[2] David Bergman et Andre A. Cire : Theoretical insights and algorithmic tools for decision diagram-based optimization. *Constraints*, 21(4):533–556, 2016.

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

[4] J.R. Burch, Clarke E.M., McMillan K.L., Dill D.L. et Hwang H.L. : Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] Xavier Gillard, Pierre Schaus et André Ciré Coppé, Vianney : Improving the filtering of branch-and-bound mdd solver. 2021.

[6] Xavier Gillard, Pierre Schaus et Vianney Coppé : Ddo, a generic and efficient framework for mdd-based optimization. 2020.

[7] Guillaume Perez et Jean-Charles Régin : Efficient operations on mdds for building constraint programming models. *In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*, pages 374–380, 2015.

[8] Hélène Verhaeghe, Christophe Lecoutre et Pierre Schaus : Compact-mdd : Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. *In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1383–1389, 2018.

Ddo, a Generic and Efficient Framework for MDD-Based Optimization

Xavier Gillard [*], **Pierre Schaus** and **Vianney Coppé**
UCLouvain
{xavier.gillard, pierre.schaus, vianney.coppe}@uclouvain.be

Abstract

This paper presents *ddo*, a generic and efficient library to solve constraint optimization problems with decision diagrams. To that end, our framework implements the branch-and-bound approach which has recently been introduced by [Bergman *et al.*, 2016b] to solve dynamic programs to optimality. Our library allowed us to successfully reproduce the results of Bergman et al. for MISP, MCP and MAX2SAT while using a single generic library. As an additional benefit, *ddo* is able to exploit parallel computing for its purpose without imposing any constraint on the user (apart from memory safety). *Ddo* is released as an open source[1] rust library (crate) alongside with its companion example programs to solve the aforementioned problems. To the best of our knowledge, this is the first public implementation of a generic library to solve combinatorial optimization problems with branch-and-bound MDD.

1 Introduction

Multivaluated Decision Diagrams (MDD) are a generalization of *Binary Decision Diagrams* (BDD) which have long been used in the verification community, e.g. for model checking purposes [Burch *et al.*, 1992]. More recently, these graphical models have drawn the attention of researchers from the CP and OR communities. The popularity of these decision diagrams (DD) stems from their ability to provide a compact representation of large solution spaces as in the case of the table constraint [Perez and Régin, 2015; Verhaeghe *et al.*, 2018]. One of the research streams which emerged from this increased interest about MDDs is *decision-diagram-based optimization* (DDO) [Bergman and Cire, 2016]. Its purpose is to efficiently solve combinatorial optimization problems by exploiting the structure of the problem being solved through the use of DDs. So far, the techniques developed in this context have largely been successful and outperforms state-of-the-art IP solvers for the problems where they are applicable. This paper belongs to the DDO subfield and intends to broaden the DDO-awareness and facilitate its integration with other solvers and techniques through the release of a generic and efficient open-source rust library implementing these algorithms and data structures.

[*]Contact Author

[1]https://github.com/xgillard/ddo

2 Background

A discrete optimization problem is first and foremost a constraint *satisfaction* problem with an associated objective function to be maximized. Among these problems, some exhibit an *optimal subproblem structure* making them suitable for a dynamic programming (DP) formulation. Even though DP models are typically thought of in terms of recursion, it is also natural to consider them as transition systems. In that case, a DP model consists of: (a) a solution space defined by the problem variables and their domains; (b) an initial state, (c) an initial value; (d) a transition function and (e) a transition cost function.

At the heart of DDO, is the idea that DP transition systems naturally lend themselves to materialization in the form of a (reduced) decision diagram. However, despite their compactness, the construction of DD suffers from a potentially exponential time and memory requirements. Using DDs to exactly encode the solution space of a problem is thus out of reach for any practical problem instance. This is why, DDO relies on the use of bounded-size DDs to approximate a solution of the actual problem. Two types of approximate bounded-size DDs have been devised for that purpose: *relaxed* and *restricted* DDs. These respectively encode an over- and under-approximation of the solution-space. Assuming a maximization problem, relaxed DDs [Andersen *et al.*, 2007] are thus capable of providing an upper bound on the optimal solution. Conversely, restricted DDs yield good lower bounds, as they contain a subset of the feasible solutions of the problem.

Deriving a restricted MDD from the DP formulation of a problem is quite simple. For that purpose, it suffices to limit the width of the MDD layers by simply dropping the less promising nodes of that layer. This process only removes solutions from the set of solutions represented by the MDD but it does not create any infeasible solution. Deriving a relaxed MDD from the same DP formulation is a different matter though. For that purpose, one needs to provide a relaxation to *merge nodes* that exceed the maximum width bound. For that reason, anyone willing to use DDO to solve a new kind of problem must provide both a DP formulation and a suit-

able relaxation for the problem; and in an ideal world, these would be the only two required inputs.

3 The *ddo* Library

This is exactly what our *ddo* framework aims to do: it starts from the definition of a problem and its relaxation to automatically and efficiently solve the problem to optimality. Furthermore, it allows a user to specify and use custom heuristics. But these are not mandatory, and the framework readily provides default heuristics.

We illustrate our point going through a minimalistic yet extensive example. Which one shows how to model and solve the binary knapsack problem with *ddo*. From Listing 1, one can observe how closely the *ddo* model matches with the mathematical abstractions outlined in the previous section. In particular, the implementation of the `Problem<usize>` trait by `Knapsack` describes the DP formulation of a binary knapsack problem whose state consists of a single unsigned integer (`usize`). The solution space (a) of the problem is characterized by the methods `nb_vars()` and `domain_of()` (lines 9–16). Similarly, the other four elements constitutive of a DP model (initial state (b), initial value (c), transition function (d) and transition cost function (e)) are all implemented by their eponymous method (lines 16–32). Also, `KPRelax` implementing the trait `Relaxation<usize>` shows what it takes to merge several nodes so as to derive a new relaxed node standing for them all (lines 38–49). In our example, the relaxed state of the new node is obtained by taking the maximum remaining capacity available in any of the merged nodes. The arcs towards the new relaxed node are obtained by (approximately) considering that the longest path to any of the merged nodes yields the relaxed node.

```
1  /// Lines 1-33 describe the problem DP formulation
2  #[derive(Debug, Clone)]
3  struct Knapsack {
4    capacity: usize,
5    profit  : Vec<usize>,
6    weight  : Vec<usize>
7  }
8  impl Problem<usize> for Knapsack {
9    fn nb_vars(&self) -> usize {
10     self.profit.len()
11   }
12   fn domain_of<'a>(&self,state: &'a usize,
13                            var  : Variable)
14   ->Domain<'a> {
15     vec![0, 1].into()
16   }
17   fn initial_state(&self) -> usize {
18     self.capacity
19   }
20   fn initial_value(&self) -> i32 {
21     0
22   }
23   fn transition(&self,state:&usize,
24                       vars :&VarSet,
25                       dec  :Decision) -> usize {
26     state - self.weight[dec.variable.id()]
27   }
28   fn transition_cost(&self,state:&usize,
29                            vars :&VarSet,
30                            dec  :Decision) -> i32 {
31     self.profit[dec.variable.id()] as i32 * dec.value
32   }
33 }
34 /// Lines 34-50 implement the problem relaxation
```

```
35 #[derive(Debug, Clone)]
36 struct KPRelax;
37 impl Relaxation<usize> for KPRelax {
38   fn merge_nodes(&self, nodes: &[Node<usize>])
39   -> Node<usize> {
40     let lp_info = nodes.iter()
41                        .map(|n| &n.info)
42                        .max_by_key(|i| i.lp_len);
43     let max_capa= nodes.iter()
44                        .map(|n| n.state)
45                        .max();
46     Node::merged(max_capa,
47                  lp_info.lp_len,
48                  lp_info.lp_arc.clone())
49   }
50 }
51 fn main() {
52   let problem = Knapsack {/* elided */};
53   let mdd = mdd_builder(&problem, KPRelax).build();
54   let mut solver = ParallelSolver::new(mdd);
55   let (optimal, solution) = solver.maximize();
56 }
```

Listing 1: Detailed example

Finally, the last fragment (lines 51–56) of Listing 1 show what it takes to instantiate the solver and use it to solve a knapsack problem instance with *ddo* using all the hardware threads available on the machine.

4 Experimental Results

To conclude our brief presentation of *ddo*, we would like to showcase some experimental results (Table 1). These figures measure the time it took (in seconds) to solve a subset of the well known MISP/Max-Clique instances from the DIMACS challenge. These measurements have been taken on a machine equipped with two Intel E5-2640v3 CPU (2.60GHz, 8 cores, 2 threads/core for a total of 32 available hardware threads) and 128G of RAM. The timeout for each run was set to 600 seconds and we set a maximum width of 100 nodes per layer of our restricted and relaxed MDDs.

These results are very promising as they indicate that even though our library is truly generic, it delivers an overall performance on par with that of DDX10[Bergman *et al.*, 2014; Bergman *et al.*, 2016a]. The latter having been favorably compared by its authors to IBM ILOG CPLEX 12.5.1.

Instance	1 thread	16 threads	32 threads
hamming8-4.clq	25.45	2.58	2.17
brock200_4.clq	18.65	1.78	1.56
san400_0.7_1.clq	48.67	4.98	4.35
p_hat300-2.clq	14.98	1.88	1.64
san1000.clq	124.46	23.18	21.78
p_hat1000-1.clq	73.98	20.07	19.58
sanr400_0.5.clq	74.07	6.80	6.21
san200_0.9_2.clq	64.94	3.13	2.62
sanr200_0.7.clq	69.67	5.81	4.91
san400_0.7_2.clq	250.07	19.74	15.94
p_hat1500-1.clq	timeout	89.28	88.40
brock200_1.clq	316.30	25.64	21.01

Table 1: Runtime (seconds) to solve MISP/Max-Clique instances from the DIMACS challenge. Timeout 600 seconds.

5 Demonstration

This demonstration will focus on how a practician can use our library to solve combinatorial optimization problems. Starting from the above knapsack example, we will show how one can tune the behavior of the solver to make the most of the available resources and problem knowledge. In particular, we will show how to opt for a static vs dynamic maximum layer width; how to opt for a single vs multi-threaded resolution and how to specify a custom variable selection heuristic in replacement of the default (natural-order) one.

References

[Andersen *et al.*, 2007] Henrik Reif Andersen, Tarik Hadzic, John Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In Christian Bessière, editor, *Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.

[Bergman and Cire, 2016] David Bergman and Andre Cire. Theoretical insights and algorithmic tools for decision diagram-based optimization. *Constraints*, 21(4):533–556, 2016.

[Bergman *et al.*, 2014] David Bergman, Andre Cire, Ashish Sabharwal, Samulowitz Horst, Saraswat Vijay, and Willem-Jan and van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, volume 8451, pages 351–367. Springer, 2014.

[Bergman *et al.*, 2016a] David Bergman, Andre Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization*. Springer, 2016.

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

[Burch *et al.*, 1992] Jerry Burch, Clarke Edmund, McMillan Kenneth, Dill David, and Hwang H.L. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Perez and Régin, 2015] Guillaume Perez and Jean-Charles Régin. Efficient operations on mdds for building constraint programming models. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*, pages 374–380, 2015.

[Verhaeghe *et al.*, 2018] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1383–1389, 2018.

Improving the filtering of Branch-And-Bound MDD solver

Xavier Gillard[1 [0000−0002−4493−6041]], Vianney Coppé[1 [0000−0001−5050−0001]], Pierre Schaus[1 [0000−0002−3153−8941]], and André Augusto Cire[2 [0000−0001−5993−4295]]

Université Catholique de Louvain, BELGIUM
University of Toronto Scarborough and Rotman School of Management, CANADA
{xavier.gillard, pierre.schaus, vianney.coppe}@uclouvain.be,
andre.cire@rotman.utoronto.ca

Abstract. This paper presents and evaluates two pruning techniques to reinforce the efficiency of constraint optimization solvers based on multi-valued decision-diagrams (MDD). It adopts the branch-and-bound framework proposed by Bergman et al. in 2016 to solve dynamic programs to optimality. In particular, our paper presents and evaluates the effectiveness of the local-bound (LocB) and rough upper-bound pruning (RUB). LocB is a new and effective rule that leverages the approximate MDD structure to avoid the exploration of non-interesting nodes. RUB is a rule to reduce the search space during the development of bounded-width-MDDs. The experimental study we conducted on the Maximum Independent Set Problem (MISP), Maximum Cut Problem (MCP), Maximum 2 Satisfiability (MAX2SAT) and the Traveling Salesman Problem with Time Windows (TSPTW) shows evidence indicating that rough-upper-bound and local-bound pruning have a high impact on optimization solvers based on branch-and-bound with MDDs. In particular, it shows that RUB delivers excellent results but requires some effort when defining the model. Also, it shows that LocB provides a significant improvement automatically; without necessitating any user-supplied information. Finally, it also shows that rough-upper-bound and local-bound pruning are not mutually exclusive, and their combined benefit supersedes the individual benefit of using each technique.

Introduction

Multi-valued Decision Diagrams (MDD) are a generalization of *Binary Decision Diagrams* (BDD) which have long been used in the verification, e.g., for model checking purposes [10]. Recently, these graphical models have drawn the attention of researchers from the CP and OR communities. One of the research streams which emerged from this increased interest about MDDs is *decision-diagram-based optimization* (DDO) [5]. Its purpose is to efficiently solve combinatorial optimization problems by exploiting problem structure through DDs. This paper belongs to the DDO sub-field and intends to further improve the

efficiency of DDO solvers through the introduction of two bounding techniques: local-bounds pruning (LocB) and rough-upper-bound pruning (RUB).

This paper starts by covering the necessary background on DDO. Then, it presents the local-bound and rough-upper-bound pruning techniques in Sections 2.1 and 2.2. After that, it presents an experimental study which we conducted using 'ddo' [17][1], our open source fast and generic MDD-based optimization library. This experimental study investigates the relevance of RUB and LocB through four disinct NP-hard problems: the Weighted Maximum Independent Set Problem (MISP), Maximum Cut Problem (MCP), Maximum 2 Satisfiability Problem (MAX2SAT) and the Traveling Salesman Problem with Time Windows (TSPTW). Finally, section 4 discusses previous related work before drawing conclusions.

1 Background

The coming paragraphs give an overview of discrete optimization with decision diagrams. Most of the formalism presented here originates from [8]. Still, we reproduce it here for the sake of self-containedness.

Discrete optimization. A discrete optimization problem is a constraint *satisfaction* problem with an associated objective function to be maximized. The discrete optimization problem $\mathcal{P}$ is defined as $\max\{f(x) \mid x \in D \wedge C(x)\}$ where C is a set of constraints, $x = \langle x_0, \ldots, x_{n-1}\rangle$ is an assignment of values to variables, 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. In that setup, the function $f : D \to \mathbb{R}$ is the objective to be maximized.

Among the set of 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 $\forall x \in Sol(\mathcal{P}) : f(x^*) \geq f(x)$.

Dynamic programming. Dynamic programming (DP) was introduced in the mid 50's by Bellman [3]. This strategy is significantly popular and is at the heart of many classical algorithms (e.g., Dijkstra's algorithm [12, p.658] or Bellman-Ford's [12, 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 of state-spaces $S_0, \ldots, S_n$ among which one distinguishes the *initial state* r, the *terminal state* t and the *infeasible state* $\perp$.
- a set of transition functions $t_i : S_i \times D_i \to S_{i+1}$ 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 $\perp$ if assigning $x_i = d$ is infeasible). These functions should never allow one to recover from infeasibility ($t_i(\perp, d) = \perp$ for any $d \in D_i$).

[1] https://github.com/xgillard/ddo

- 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:

$$\begin{aligned}
&\text{maximize } f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i) \\
&\text{subject to} \\
&s^{i+1} = t_i(s^i, x_i) \text{ for } i = 0, \ldots, n-1; x_i \in D_i \wedge C(x_i) \\
&s^i \in S_i \text{ for } i = 0, \ldots, n
\end{aligned}$$

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 $Sol(\mathcal{P})$.

1.1 Decision diagrams

Because DDO aims at solving constraint *optimization* problems and not just constraint *satisfaction* problems, it uses a particular DD flavor known as reduced weighted DD – DD as of now. As initially posed by Hooker[21], DDs can be perceived as a compact representation of the search trees. This is achieved, in this context, by superimposing isomorphic subtrees.

To define our DD more formally, we will slightly adapt the notation from [5]. 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 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 $t_3(\sigma(u_1), 6) = \sigma(u_2)$ and the benefit of that assignment is $v(a) = h_3(\sigma(u_1), 6) = 42$.

Because each r-t path describes an assignment that satisfies $\mathcal{P}$, 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 longest r-t path in $\mathcal{B}$ (according to the relation v on arcs).

Exact-MDD. For a given problem $\mathcal{P}$, an exact MDD $\mathcal{B}$ is an MDD 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 MDD for $\mathcal{P}$ can be compiled in a top-down fashion[2]. 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.

1.2 Bounded-Size Approximations

In spite of the compactness of their encoding, the construction of DD suffers from a potentially exponential memory requirement in the worst case[3]. Thus, using DDs to exactly encode the solution space of a problem is often intractable. Therefore, one must resort to the use of *bounded-size* approximation of the exact MDD. These are compiled generically by inserting a call to a width-bounding procedure 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. Depending on the behavior of that procedure, one can either compile a restricted-MDD (= an under-approximation) or a relaxed-MDD (= an over-approximation).

Restricted-MDD: Under-approximation. A restricted-MDD provides an under-approximation of some exact-MDD. As such, all paths of a restricted-MDD encode valid solutions, but some solutions might be missing from the MDD. This is formally expressed as follows: given the DP formulation of a problem $\mathcal{P}$, $\mathcal{B}$ is a restricted-MDD iff $Sol(\mathcal{B}) \subseteq Sol(\mathcal{P})$.

To compile a restricted-MDD, it is sufficient to simply delete certain nodes from the current layer until its width fits within the specified bound W. To that end, the width-bounding procedure simply selects a subset of the nodes from L_i which are heuristically assumed to have the less impact on the tightness of the bound. Various heuristics have been studied in the literature [7], and *minLP* was shown to be the heuristic that works best in practice. This heuristic decides to select (hence remove) the nodes having the shortest longest path from the root.

[2] An incremental refinement *a.k.a. construction by separation* procedure is detailed in [11, pp. 51–52] but we will not cover it here for the sake of conciseness.

[3] Consequently, it also suffers from a potentially exponential time requirement in the worst case. Indeed, time is constant in the final number of nodes (unless the transition functions themselves are exponential in the input).

Relaxed-MDD: Over-approximation. A relaxed-MDD $\mathcal{B}$ provides a bounded-width over-approximation of some exact-MDD. As such, it may hold paths that are no solution to $\mathcal{P}$, the problem being solved. We have thus formally that $Sol(\mathcal{B}) \supseteq Sol(\mathcal{P})$.

Compiling a relaxed-MDD requires one to be able to *merge* several nodes into an inexact one. To that end, we use two operators:

- $\oplus$ which yields a new node combining the states of a selection of nodes so as to over-approximate the states reachable in the selection.
- Γ which is used to possibly relax the weight of arcs incident to the selected nodes.

These operators are used as follows. Similar to the restricted-MDDs case, the width-bounding procedure starts by heuristically selecting the least promising nodes and removing them from layer L_i. Then the states of these selected nodes are combined with one another so as to create a merged node $\mathcal{M} = \oplus(selection)$. After that, the inbound arcs incident to all selected nodes are Γ-relaxed and redirected towards $\mathcal{M}$. Finally, the result of the merger ($\mathcal{M}$) is added to the layer in place of the initial selection of nodes.

Summary. Fig. 1 summarizes the information from sections 1.1 and 1.2. It displays the three MDDs corresponding to one same example problem having four variables. The exact MDD (a) encodes the complete solution set and, equivalently, the state space of the underlying DP encoding. One easily notices that the restricted DD (b) is an under approximation of (a) since it achieves its width boundedness by removing nodes d and e and their children (i, j). Among others, it follows that the solution $[x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0]$ is not represented in (b) even though it exists in (a). Conversely, the relaxed diagram (c) achieves a maximum layer with of 3 by merging nodes d, e and h into a new inexact node $\mathcal{M}$ and by relaxing all arcs entering one of the merged nodes. Because of this, (c) introduces solutions that do not exist in (a) as is for instance the case of the assignment $[x_0 = 0, x_1 = 0, x_2 = 3, x_3 = 1]$. Moreover, because the operators $\oplus$ and Γ are correct[4], the length of the longest path in (c) is an upper bound on the optimal value of the objective function. Indeed, one can see that the length of the longest path in (a) (= the exact optimal solution) has a value of 25 while it amounts to 26 in (c).

1.3 The Dynamics of Branch-and-Bound with DDs

Being able to derive good lower and upper bounds for some optimization problem $\mathcal{P}$ is useful when the goal is to use these bounds to strengthen algorithms [13, 31, 32]. But it is not the only way these approximations can be used. A complete and efficient branch-and-bound algorithm relying on those approximations was proposed in [8] which we hereby reproduce (Alg. 1).

[4] The very definition of these operators is problem-specific. However, [22] formally defines the conditions that are necessary to correctness.

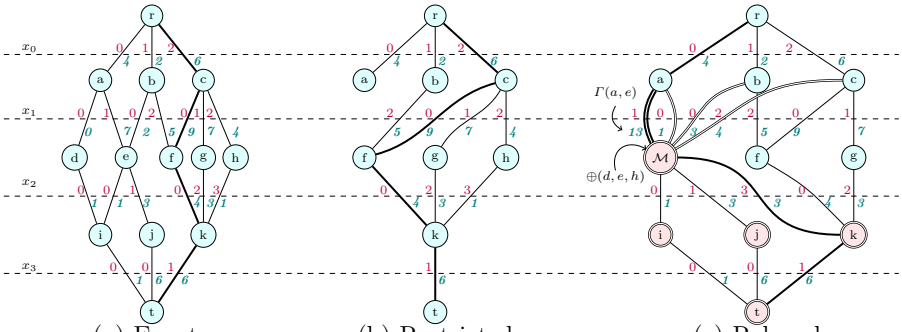

Fig. 1. The exact (a), restricted (b) and relaxed (c) versions of an MDD with four variables. The width of MDDs (b) and (c) have been bounded to a maximum layer width of three. The decision labels of the arcs are shown above the layers separation lines (dashed). The arc weights are shown below the layer separation lines. The longest path of each MDD is boldfaced. In (c), the node $\mathcal{M}$ is the result of merging nodes d, e and h with the $\oplus$ operator. Arcs that have been relaxed with the Γ operator are pictured with a double stroke. Note, because these arcs have been Γ-relaxed, their value might be greater than that of corresponding arcs in (a), (b). Similarly, all "inexact" nodes feature a double border.

This algorithm works as follows: at start, the node r is created for the initial state of the problem and placed onto the *fringe* – a global priority queue that tracks all nodes remaining to explore and orders them from the most to least promising. Then, a loop consumes the nodes from that fringe (line 1), one at a time and explores it until the complete state space has been exhausted. The *exploration* of a node u inside that loop proceeds as follows: first, one compiles a restricted DD $\underline{\mathcal{B}}$ for the sub-problem rooted in u (line 5). Because all paths in a restricted DD are feasible solutions, when the lower bound $v^*(\underline{\mathcal{B}})$ derived from the restricted DD $\underline{\mathcal{B}}$ improves over the current best known solution $\underline{v}$; then the longest path of $\underline{\mathcal{B}}$ (best sol. found in $\underline{\mathcal{B}}$) and its length $v^*(\underline{\mathcal{B}})$ are memorized (lines 7-9).

In the event where $\underline{\mathcal{B}}$ is exact (no restriction occurred during the compilation of $\underline{\mathcal{B}}$), it covers the complete state space of the sub-problem rooted in u. Which means the processing of u is complete and we may safely move to the next node. When this condition is not met, however, some additional effort is required.In that case, a *relaxed* DD $\overline{\mathcal{B}}$ is compiled from u(line 11). That relaxed DD serves two purposes: first, it is used to derive an upper bound $v^*(\overline{\mathcal{B}})$ which is compared to the current best known solution (line 12). This gives us a chance to prune the unexplored state space under u when $v^*(\overline{\mathcal{B}})$ guarantees it does not contain any better solution than the current best. The second use of $\overline{\mathcal{B}}$ happens when $v^*(\overline{\mathcal{B}})$ cannot provide such a guarantee. In that case, the exact cutset of $\overline{\mathcal{B}}$ is used to enumerate residual sub-problems which are enqueued onto the fringe (lines 13-14).

A cutset for some relaxed DD $\overline{\mathcal{B}}$ is a subset $\mathcal{C}$ of the nodes from $\overline{\mathcal{B}}$ such that any $r-t$ path of $\overline{\mathcal{B}}$ goes through at least one node $\in \mathcal{C}$. Also, a node u is said to be exact iff all its incoming paths lead to the same state $\sigma(u)$. From there, an exact cutset of $\overline{\mathcal{B}}$ is simply a cutset whose nodes are all exact. Based on this definition, it is easy to convince oneself that an exact cutset constitutes a frontier up to which the relaxed DD $\overline{\mathcal{B}}$ and its exact counterpart $\mathcal{B}$ have not diverged. And, because it is a cutset, the nodes composing that frontier cover all paths from both $\mathcal{B}$ and $\overline{\mathcal{B}}$; which guarantees the completeness of Alg. 1 [8].

Any relaxed-MDD admits at least one exact cutset – e.g. the trivial $\{r\}$ case. Often though, it is not unique and different options exist as to what cutset to use. It was experimentally shown by [8] that most of the time, the Last Exact Layer (LEL) is superior to all other exact cutsets in practice. LEL consists of the *deepest* layer of the relaxed-MDD having all its nodes exact.

Example 2. In Fig.-1 (c), the first inexact node $\mathcal{M}$ occurs in layer L_2. Hence, the LEL cutset comprises all nodes (a, b, c) from the layer L_1. Because $\mathcal{M}$ is inexact, and because it is a parent of nodes i, j and k, these three nodes are considered inexact too.

Algorithm 1 Branch-And-Bound with DD

```
1: Create node r and add it to Fringe
2: x ← ⊥
3: v ← −∞
4: while Fringe is not empty do
5:    u ← Fringe.pop()
6:    B ← Restricted(u)
7:    if v*(B) > v then
8:       v ← v*(B)
9:       x ← x*(B)
10:   if B is not exact then
11:      B̄ ← Relaxed(u)
12:      if v*(B̄) > v then
13:         for all u' ∈ B̄.exact_cutset() do
14:            Fringe.add(u')
15: return (x, v)
```

Algorithm 2 Local bound pruning

```
1: Create node r and add it to Fringe
2: x ← ⊥
3: v ← −∞
4: while Fringe is not empty do
5:    u ← Fringe.pop()
6:    if v|*_u ≤ v then
7:       continue
8:    B ← Restricted(u)
9:    if v*(B) > v then
10:      v ← v*(B)
11:      x ← x*(B)
12:   if B is not exact then
13:      B̄ ← Relaxed(u)
14:      if v*(B̄) > v then
15:         for all u' ∈ B̄.exact_cutset() do
16:            if v|*_u' > v then
17:               Fringe.add(u')
18: return (x, v)
```

2 Improving the filtering of branch-and-bound MDD

In the forthcoming paragraphs, we introduce the local bound and present the rough upper bound: two reasoning techniques to reinforce the pruning strength of Alg. 1.

2.1 Local bounds (LocB)

Conceptually, pruning with local bounds is rather simple: a relaxed MDD $\overline{\mathcal{B}}$ provides us with *one* upper bound $v^*(\overline{\mathcal{B}})$ on the optimal value of the objective function for some given sub-problem. However, in the event where $v^*(\overline{\mathcal{B}})$

is greater than the best known lower bound $\underline{v}$ (best current solution) nothing guarantees that all nodes from the exact cutset of $\overline{\mathcal{B}}$ admit a longest path to t with a length of $v^*(\overline{\mathcal{B}})$. Actually, this is quite unlikely. This is why we propose to attach a *"local" upper bound* to each node of the cutset. This local upper bound – denoted $v|^*_u$ for some cutset node u – simply records the length of the longest r-t path passing through u in the relaxed MDD $\overline{\mathcal{B}}$.

In other words, LocB allows us to refine the information provided by a relaxed DD $\overline{\mathcal{B}}$. On one hand, $\overline{\mathcal{B}}$ provides us with $v^*(\overline{\mathcal{B}})$ which is the length of the longest r-t path in $\overline{\mathcal{B}}$. As such, it provides an upper bound on the optimal value that can be reached from the root node of $\overline{\mathcal{B}}$. With the addition of LocB, the relaxed DD provides us with an additional piece of information. For each individual node u in the exact cutset of $\overline{\mathcal{B}}$, it defines the value $v|^*_u$ which is an upper bound on the value attainable from that node.

As shown in Alg. 2, the value $v|^*_u$ can prove useful at two different moments. First, in the event where $v|^*_u \leq \underline{v}$, this value can serve as a justification to not enqueue the subproblem u (line 16) since exhausting this subproblem will yield no better solution than $\underline{v}$. More formally, by definition of a cutset and of LocB, it must be the case that the longest r-t path of $\overline{\mathcal{B}}$ traverses one of the cutset nodes u and thus that $v^*(\overline{\mathcal{B}}) = v|^*_u$ (where $v|^*_u$ is the local bound of u). Hence we have: $\exists u \in$ cutset of $\overline{\mathcal{B}} : v^*(\overline{\mathcal{B}}) = v|^*_u$. However, because $v^*(\overline{\mathcal{B}})$ is the length of the *longest* r-t path of $\overline{\mathcal{B}}$, there may exist cutset nodes that only belong to r-t paths shorter than $v^*(\overline{\mathcal{B}})$. That is: $\forall u' \in$ cutset of $\overline{\mathcal{B}} : v^*(\overline{\mathcal{B}}) \geq v|^*_{u'}$. Which is why $v|^*_{u'}$ can be stricter than $v^*(\overline{\mathcal{B}})$ and hence let LocB be stronger at pruning nodes from the frontier.

The second time when $v|^*_u$ might come in handy occurs when the node u is popped out of the fringe (line 6). Indeed, because the fringe is a global priority queue, any node that has been pushed on the fringe can remain there for a long period of time. Thus, chances are that the value $\underline{v}$ has increased between the moment when the node was pushed onto the fringe (line 17) and the moment when it is popped out of it. Hence, this gives us an additional chance to completely skip the exploration of the sub-problem rooted in u.

Let us illustrate that with the relaxed MDD shown on Fig.2, for which the exact cutset comprises the highlighted nodes a and b. Please note that because this scenario may occur at any time during the problem resolution, we will assume that the fringe is not empty when it starts. Assuming that the current best solution $\underline{v}$ is 20 when one explores the pictured subproblem, we are certain that exploring the subproblem rooted in a is a waste of time, because the local bound $v|^*_a$ is only 16. Also, because the fringe was not empty, it might be the case that b was left on the fringe for a long period of time. And because of this, it might be the case that the best known value $\underline{v}$ was improved between the moment when b was pushed on the fringe and the moment when it was popped out of it. Assuming that $\underline{v}$ has improved to 110 when b is popped out of the fringe, it may safely be skipped because $v|^*_b$ guarantees that an exploration of b will not yield a better solution than 102.

Alg. 3 describes the procedure to compute the local bound $v|^*_u$ of each node u belonging to the exact cutset of a relaxed MDD $\overline{\mathcal{B}}$. Intuitively, this is achieved by doing a bottom-up traversal of $\overline{\mathcal{B}}$, starting at t and stopping when the traversal crosses the last exact layer (line 5). During that bottom-up traversal, the algorithm marks the nodes that are reachable from t. This way, it can avoid the traversal of dead-end nodes. Also, Alg. 3 maintains a value $v^*_{\uparrow_t}(u)$ for each node u it encounters. This value represents the length of the longest u-t path. Afterwards (line 13), it is summed with the length of the longest r-u path v^*_{r-u} to derive the exact value of the local bound $v|^*_u$.

Algorithm 3 Computing the local bounds

```
1: lel ← Index of the last exact layer
2: v*↑t(u) ← −∞ for each node u ∈ B̄                    // init. longest u-t path
3: mark(t) ←true
4: v*↑t(t) ← 0                                          // longest t-t path
5: for all i = n to lel do
6:    for all node u ∈ L_i do
7:       if mark(u) then
8:          for all arc a = (u', u) incident to u do
9:             mark(u') ← true
10:            v*↑t(u') ← max(v*↑t(u'), v*↑t(u) + v(a))      // longest u'-t path
11: for all node u ∈ B̄.exact_cutset() do
12:   if mark(u) then
13:      v|*u ← v*r−u + v*↑t(u)          // longest r-u path + longest u-t path
14:   else
15:      v|*u ← −∞
```

2.2 Rough upper bound (RUB)

Rough upper bound pruning departs from the following observation: assuming the knowledge of a lower bound $\underline{v}$ on the value of v^*, and assuming that one is able to swiftly compute a rough upper bound $\overline{v_s}$ on the optimal value v^*_s of the subproblem rooted in state s; any node u of a MDD having a rough upper bound $\overline{v_{\sigma(u)}} \leq \underline{v}$ may be discarded as it is guaranteed not to improve the best known solution. This is pretty much the same reasoning that underlies the whole branch-and-bound idea. But here, it is used to prune portions of the search space explored *while compiling* approximate MDDs.

To implement RUB, it suffices to adapt the MDD compilation procedure (top-down , iterative refinement, ...) and introduce a check that avoids creating a node u' with state $next$ when $\overline{v_{next}} \leq \underline{v}$.

The key to RUB effectiveness is that RUB is used while compiling the restricted and relaxed DDs. As such, its computation does not directly appear in Alg. 1, but rather is accounted within the compilations of $Restricted(u)$ and

Relaxed(u) from Alg. 1. Thus, it really is not used as yet-an-other-bound competing with that of line 12, but instead to speed up the computation of restricted and relaxed DDs. More precisely, this speedup occurs because the compilation of the DDs discards some nodes that would otherwise be added to the next layer of the DD and then further expanded, which are ruled out by RUB. A second benefit of using RUBs is that it helps tightening the bound derived from a relaxed DD (Alg.1 line 12). Because the layers that are generated in a relaxed DD are narrower when applying RUB, there are fewer nodes exceeding the maximum layer width. The operator $\oplus$ hence needs to merge a smaller set of nodes in order to produce the relaxation.

The dynamics of RUB is graphically illustrated by Fig.-3 where the set of highlighted nodes can be safely elided since the (rough) upper bound computed in node s is lesser than the best lower bound.

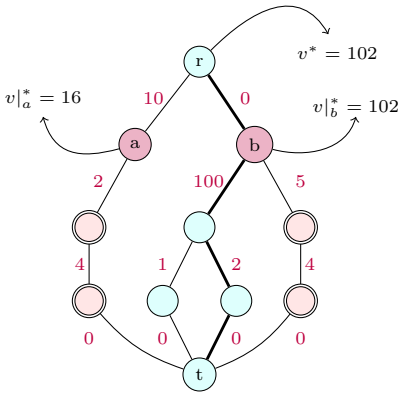

Fig. 2. An example relaxed-MDD having an exact cutset $\{a, b\}$ with local bounds $v|_a^*$ and $v|_b^*$. The nodes with a simple border represent exact nodes and those with a double border represent "inexact" nodes. The edges along the longest path are displayed in bold.

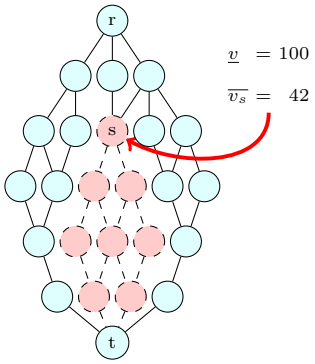

Fig. 3. Assuming a lower bound $\underline{v}$ of 100 and a rough upper bound $\overline{v_s}$ of 42 for the node s, all the highlighted nodes (in red, with a dashed border) may be pruned from the MDD.

Important Note It is important to understand that because the RUB is computed at each node of each restricted and relaxed MDD compiled during the instance resolution, it must be extremely inexpensive to compute. This is why RUB is best obtained from a fast and simple problem specific procedure.

3 Experimental Study

In order to evaluate the impact of the pruning techniques proposed above, we conducted a series of experiments on four problems. In particular, we conducted experiments on the Maximum Independent Set Problem (MISP), the Maximum Cut Problem (MCP), the Maximum Weighted 2-Satisfiablility Problem (MAX2SAT) and the Traveling Salesman Problem with Time Windows (TSPTW). For the first three problems, we generated sets of random instances which we attempted to solve with different configurations of our own open source solver written in Rust [17][5]. For TSPTW, we reused openly available sets of benchmarks which are usually used to assess the efficiency of new solvers for TSPTW[27]. Thanks to the generic nature of our framework, the model and all heuristics used to solve the instances were the same for all experiments. This allowed us to isolate the impact of RUB and LocB on the solving performance and neutralize unrelated factors such as variable ordering. Indeed, the only variations between the different solver flavors relate to the presence (or absence) of RUB and LocB. All experiments were run on the same physical machine equipped with an AMD6176 processor and 48GB of RAM. A maximum time limit of 1800 seconds was allotted to each configuration to solve each instance.

The details of the DP models and RUBs we formulated for all four problems are given in the appendices to the extended version of this paper [6].

MISP. To assess the impact of RUB and LocB on MISP, we generated random graphs based on the Erdos-Renyi model G(n, p) [15] with the number of vertices n = 250, 500, 750, 1000, 1250, 1500, 1750 and the probability of having an edge connecting any two vertices p = 0.1, 0.2, ... , 0.9. The weight of the edges in the generated graphs were drawn uniformly from the set $\{-5, -4, -3, -2, -1, 1, 2, 3, 4, 5\}$. We generated 10 instances for each combination of size and density (n, p).

MCP. In line with the strategy used for MISP, we generated random MCP instances as random graphs based on the Erdos-Renyi model G(n, p). These graphs were generated with the number of vertices n = 30, 40, 50 and the probability p of connecting any two vertices = 0.1, 0.2, 0.3, .., 0.9. The weights of the edges in the generated graphs were drawn uniformly among $\{-1, 1\}$. Again, we generated 10 instances per combination n, p.

MAX2SAT. Similar to the above, we used random graphs based the Erdos-Renyi model G(n, p) to derive MAX2SAT instances. To this end, we produced graphs with n = 60, 80, 100, 200, 400, 1000 (hence instances with 30, 40, 50, 100, 200 and 500 variables) and p = 0.1, 0.2, 0.3, .. , 0.9. For each combination of size (n) and density (p), we generated 10 instances. The weights of the clauses in the generated instances were drawn uniformly from the set $\{1, 2, 3, 5, 6, 7, 8, 9, 10\}$.

[5] https://github.com/xgillard/ddo
[6] Available online at: http://hdl.handle.net/2078.1/245322

TSPTW. To evaluate the effectiveness of our rules on TSPTW, we used the 467 instances from the following suites of benchmarks, which are usually used to assess the efficiency of new TSPTW solvers. AFG [2], Dumas [14], Gendreau-Dumas [16], Langevin [26], Ohlmann-Thomas [28], Solomon-Pesant [29] and Solomon-Potvin-Bengio [30].

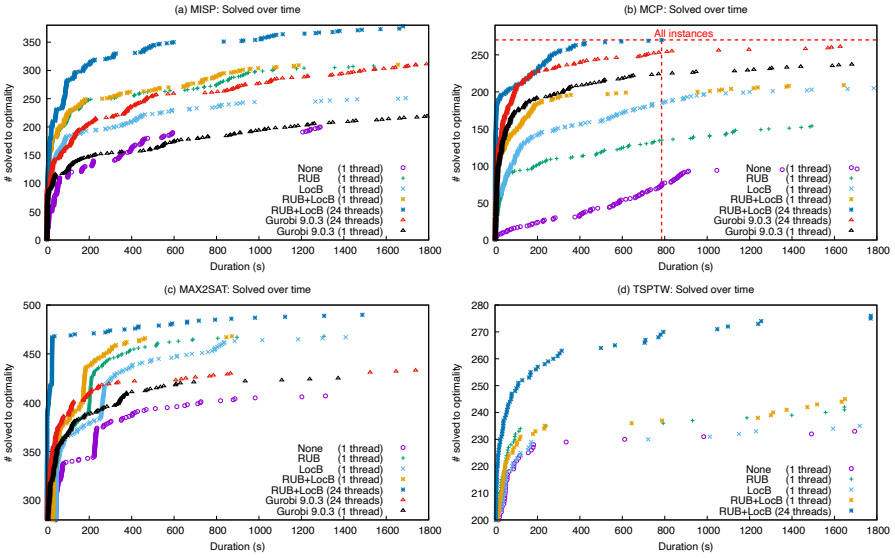

Fig. 4. Number of solved instances over time for each considered problem

Figure 4 gives an overview of the results from our experimental study. It respectively depicts the evolution over time of the number of instances solved by each technique for MISP (a), MCP (b) and MAX2SAT (c) and TSPTW (d).

As a first step, our observation of the graphs will focus on the differences that arise between the single threaded configurations of our *ddo* solvers. Then, in a second phase, we will incorporate an existing state-of-the-art ILP solver (Gurobi 9.0.3) in the comparison. Also, because both Gurobi and our *ddo* library come with built-in parallel computation capabilities, we will consider both the single threaded and parallel (24 threads) cases. This second phase, however, only bears on MISP, MCP and MAX2SAT by lack of a Gurobi TSPTW model.

DDO configurations The first observation to be made about the four graphs in Fig.4, is that for all considered problems, both RUB and LocB outperformed the 'do-nothing' strategy; thereby showing the relevance of the rules we propose. It is not clear however which of the two rules brings the most improvement to the problem resolution. Indeed, RUB seems to be the driving improvement factor for MISP (a) and TSPTW (d) and the impact of LocB appears to be moderate or

weak on these problems. However, it has a much higher impact for MCP (b) and MAX2SAT (c). In particular, LocB appears to be the driving improvement factor for MCP (b). This is quite remarkable given that LocB operates in a purely black box fashion, without any problem-specific knowledge. Finally, it should also be noted that the use of RUB and LocB are not mutually exclusive. Moreover, it turns out that for all considered problems, the combination RUB+LocB improved the situation over the use of any single rule.

Furthermore, Fig.5 confirms the benefit of using both RUB *and* LocB together rather than using any single technique. For each problem, it measures the "performance" of using RUB+LocB vs the best single technique through the end gap. The end gap is defined as $\left(100 * \frac{|UB|-|LB|}{|UB|}\right)$. This metric allows us to account for all instances, including the ones that could not be solved to optimality. Basically, a small end gap means that the solver was able to confirm a tight confidence interval of the optimum. Hence, a smaller gap is better. On each subgraphs of Fig.5, the distance along the x-axis represents the end gap for reach instance when using both RUB and LocB whereas the distance along y-axis represents the end gap when using the best single technique for the problem at hand. Any mark above the diagonal shows an instance for which using both RUB and LocB helped reduce the end gap and any mark below that line indicates an instance where it was detrimental.

From graphs 5-a, 5-c and 5-d it appears that the combination RUB+LocB supersedes the use of RUB only. Indeed the vast majority of the marks sit above the diagonal and the rest on it. This indicates a beneficial impact of using both techniques even for the hardest (unsolved) instances. The case of MCP (graph 5-b) is less clear as most of the marks sit on the diagonal. Still, we can only observe three marks below the diagonal and a bit more above it. Which means that even though the use of RUB in addition to LocB is of little help in the case of MCP, its use does not degrade the performance for that considered problem.

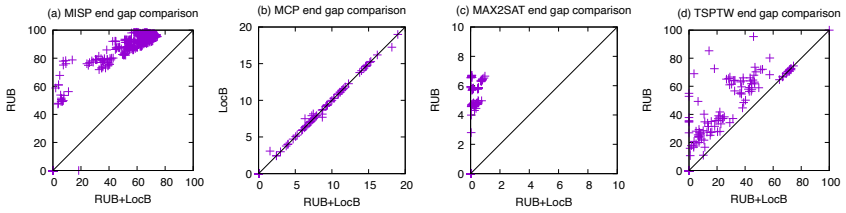

Fig. 5. End gap: The benefit of using both techniques vs the best single one

Comparison with Gurobi 9.0.3 The first observation to be made when comparing the performance of Gurobi vs the DDO configurations, is that when running on a single thread, ILP outperforms the basic DDO approach (without RUB and LocB). Furthermore, Gurobi turns out to be the best single threaded solver for

MCP by a fair margin. However, in the MISP and MAX2SAT cases, Fig. 4 shows that the DDO solvers benefitting from RUB and LocB were able to solve more instances and to solve them faster than Gurobi. Which underlines the importance of RUB and LocB.

When lifting the one thread limit, one can see that the DD-based approach outperform ILP on each of the considered problems. In particular, in the case of MCP for which Gurobi is the best single threaded option; our DDO solver was able to find and prove the optimality of all tested instances in a little less than 800 seconds. The ILP solver, on the other end, was not able to prove the optimality of the 9 hardest instances within 30 minutes. Additionally, we also observe that in spite of the performance gains of MIP when running in parallel, Gurobi fails to solve as many MISP and MAX2SAT instances and to solve them as fast as the single threaded DDO solvers with RUB and LocB. This emphasizes once more the relevance of our techniques. It also shows that the observation from [9] still hold today: despite the many advances of MIP the DDO approach still scales better than MIP on the considered problems when invoked in parallel.

4 Previous work

DDO emerged in the mid' 2000's when [24] proposed to use decision diagrams as a way to solve discrete optimization problems to optimality. More or less concomitantly, [1] devised relaxed-MDD even though the authors envisioned its use as a CP constraint store rather than a means to derive tight upper bounds for optimization problems. Then, the relationship between decision diagrams and dynamic programming was clarified by [21].

Recently, Bergman, Ciré and van Hoeve investigated the various ways to compile decision diagrams for optimization (top-down, construction by separation) [11]. They also investigated the heuristics used to parameterize these DD compilations. In particular, they analyzed the impact of variable ordering in [11, 7] and node selection heuristics (for merge and deletion) in [7]. Doing so, they empirically demonstrated the crucial impact of variable ordering on the tightness of the derived bounds and highlighted the efficiency of minLP as a node selection heuristic. Later on, the same authors proposed a complete branch-and-bound algorithm based on DDs [8]. This is the algorithm which we propose to adapt with extra reasoning mechanisms and for which we provide a generic open-source implementation in Rust [17]. The impressive performance of DDO triggered some theoretical research to analyze the quality of approximate MDDs [5] and the correctness of the relaxation operators [22].

This gave rise to new lines of work. The first one focuses on the resolution of a larger class of optimization problems; chief of which multi-objective problems [4] and problems with a non-linear objective function. These are either solved by decomposition [4] or by using DDO to strengthen other IP techniques [13]. A second trend aims at hybridizing DDO with other IP techniques. For instance, by using Lagrangian relaxation [23] or by solving a MIP [6] to derive with very tight bounds. But the other direction is also under active investigation: for example,

[31, 32] use DD to derive tight bounds which are used to replace LP relaxation in a cutting planes solver. Very recently, a third hybridization approach has been proposed by Gonzàlez et al.[18]. It adopts the branch-and-bound MDD perspective, but whenever an upper bound is to be derived, it uses a trained classifier to decide whether the upper bound is to be computed with ILP or by developing a fixed-width relaxed MDD.

The techniques (ILP-cutoff pruning and ILP-cutoff heuristic) proposed by Gonzalez et al.[18] are related to RUB and LocB in the sense that all techniques aim at reducing the search space of the problem. However, they fundamentally differ as ILP-cutoff pruning acts as a replacement for the compilation of a relaxed MDD whereas the goal of RUB is to speed up the development of that relaxed MDD by removing nodes *while* the MDD is being generated. The difference is even bigger in the case of ILP-cutoff heuristic vs LocB: the former is used as a primal heuristic while LocB is used to filter out sub-problems that can bear no better solution. In that sense, LocB belongs more to the line of work started by [1, 19, 20]: it enforces the constraint $lb \leq f(x) \leq ub$ and therefore provokes the deletion of nodes and arcs that cannot lead to the optimal solution.

More recently, Horn et al explored an idea in [25] which closely relates to RUB. They use "fast-to-compute dual bounds" as an admissible heuristic to guide the compilation of MDDs in an A* fashion for the prize-collecting TSP. It prunes portions of the state space during the MDD construction, similarly to when RUB is applied. Our approach differs from that of [25] in that we attempt to incorporate problem specific knowledge in a framework that is otherwise fully generic. More precisely, it is perceived here as a problem-specific pruning that exploits the combinatorial structure implied by the state variables. It is independent of other MDD compilation techniques, e.g., our techniques are compatible with node merge ($\oplus$) operators and other methodologies defined in the DDO literature. We also emphasize that, as opposed to more complex LP-based heuristics that are now typical in A* search, we investigate quick methodologies that are also easy to incorporate in a MDD branch and bound.

Conclusion and future work

This paper presented and evaluated the impact of the local bound and rough upper bound techniques to strengthen the pruning of the branch-and-bound MDD algorithm. Our experimental study on MISP, MCP, MAX2SAT and TSPTW confirmed the relevance of these techniques. In particular, our experiments have shown that devising a fast and simple rough upper bound is worth the effort as it can significantly boost the efficiency of a solver. Similarly, our experiments showed that the use of local bound can significantly improve the efficiency of DDO solver despite its problem agnosticism. Furthermore, it revealed that a combination of RUB and LocB supersedes the benefit of any single reasoning technique. These results are very promising and we believe that the public availability of an open source DDO framework implementing RUB and LocB might serve as a basis for novel DP formulation for classic problems.

References

1. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 118–132. Springer (2007)
2. Ascheuer, N.: Hamiltonian path problems in the on-line optimization of flexible manufacturing systems (1996)
3. Bellman, R.: The theory of dynamic programming. Bulletin of the American Mathematical Society **60**(6), 503–515 (11 1954), https://projecteuclid.org:443/euclid.bams/1183519147
4. Bergman, D., Cire, A.A.: Multiobjective optimization by decision diagrams. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 9892, pp. 86–95. Springer (2016)
5. Bergman, D., Cire, A.A.: Theoretical insights and algorithmic tools for decision diagram-based optimization. Constraints **21**(4), 533–556 (2016). https://doi.org/10.1007/s10601-016-9239-9, https://doi.org/10.1007/s10601-016-9239-9
6. Bergman, D., Cire, A.A.: On finding the optimal bdd relaxation. In: Salvagnin, D., Lombardi, M. (eds.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 10335, pp. 41–50. Springer (2017)
7. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Optimization bounds from binary decision diagrams. INFORMS Journal on Computing **26**(2), 253–268 (2014). https://doi.org/10.1287/ijoc.2013.0561, https://doi.org/10.1287/ijoc.2013.0561
8. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016). https://doi.org/10.1287/ijoc.2015.0648, https://doi.org/10.1287/ijoc.2015.0648
9. Bergman, D., Cire, A.A., Sabharwal, A., Samulowitz, H., Saraswat, V., van Hoeve, W.J.: Parallel combinatorial optimization with decision diagrams. International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems pp. 351–367 (2014)
10. Burch, J., E.M., C., K.L., M., D.L., D., H.L., H.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A, https://doi.org/10.1016/0890-5401(92)90017-A
11. Cire, A.A.: Decision Diagrams for Optimization. Ph.D. thesis, Carnegie Mellon University Tepper School of Business (2014)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
13. Davarnia, D., van Hoeve, W.J.: Outer approximation for integer nonlinear programs via decision diagrams (2018)
14. Dumas, Y., Desrosiers, J., Gelinas, E., Solomon, M.M.: An optimal algorithm for the traveling salesman problem with time windows. Operations research **43**(2), 367–371 (1995)
15. Erdös, P., Rényi, A.: On random graphs i. Publicationes Mathematicae Debrecen **6**, 290 (1959)
16. Gendreau, M., Hertz, A., Laporte, G., Stan, M.: A generalized insertion heuristic for the traveling salesman problem with time windows. Operations Research **46**(3), 330–335 (1998)

17. Gillard, X., Schaus, P., Coppé, V.: Ddo, a generic and efficient framework for mdd-based optimization. Accepted at the International Joint Conference on Artificial Intelligence (IJCAI-20); DEMO track (2020)
18. Gonzalez, J.E., Cire, A.A., Lodi, A., Rousseau, L.M.: Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem. Constraints pp. 1–24 (2020)
19. Hadžić, T., Hooker, J., Tiedemann, P.: Propagating separable equalities in an mdd store. In: CPAIOR. pp. 318–322 (2008)
20. Hoda, S., Van Hoeve, W.J., Hooker, J.N.: A systematic approach to mdd-based constraint programming. In: International Conference on Principles and Practice of Constraint Programming. pp. 266–280. Springer (2010)
21. Hooker, J.N.: Decision diagrams and dynamic programming. In: Gomes, C., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 7874, pp. 94–110. Springer (2013)
22. Hooker, J.N.: Job sequencing bounds from decision diagrams. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 10416, pp. 565–578. Springer (2017)
23. Hooker, J.N.: Improved job sequencing bounds from decision diagrams. In: Schiex, T., de Givry, S. (eds.) Principles and Practice of Constraint Programming. LNCS, vol. 11802, pp. 268–283. Springer (2019)
24. Hooker, J.: Discrete global optimization with binary decision diagrams. GICOLAG 2006 (2006)
25. Horn, M., M̃aschler, J., R̃aidl, G.R., R̃önnberg, E.: A*-based construction of decision diagrams for a prize-collecting scheduling problem. Computers & Operations Research **126**, 105125 (2021). https://doi.org/https://doi.org/10.1016/j.cor.2020.105125, http://www.sciencedirect.com/science/article/pii/S0305054820302422
26. Langevin, A., Desrochers, M., Desrosiers, J., Gélinas, S., Soumis, F.: A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. Networks **23**(7), 631–640 (1993)
27. López-Ibáñez, M., Blum, C.: Benchmark instances for the travelling salesman problem with time windows. Online (2020), http://lopez-ibanez.eu/tsptw-instances
28. Ohlmann, J.W., Thomas, B.W.: A compressed-annealing heuristic for the traveling salesman problem with time windows. INFORMS Journal on Computing **19**(1), 80–90 (2007)
29. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transportation Science **32**(1), 12–29 (1998)
30. Potvin, J.Y., Bengio, S.: The vehicle routing problem with time windows part ii: genetic search. INFORMS journal on Computing **8**(2), 165–172 (1996)
31. Tjandraatmadja, C.: Decision Diagram Relaxations for Integer Programming. Ph.D. thesis, Carnegie Mellon University Tepper School of Business (2018)
32. Tjandraatmadja, C., van Hoeve, W.J.: Target cuts from relaxed decision diagrams. INFORMS Journal on Computing **31**(2), 285–301 (2019). https://doi.org/10.1287/ijoc.2018.0830, https://doi.org/10.1287/ijoc.2018.0830