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 suitable 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.