C++DDOopt : The CODD Solver

Laurent Michel

April 20, 2024

Contents

1 Introduction

1.1 What is CODD?

CODD is a system for modeling and solving combinatorial optimization problems using decision diagram technology. Problems are represented as state-based dynamic programming models using the CODD language specification. The model specification is used to automatically compile relaxed and restricted decision diagrams, as well as to guide the search process. **CODD** introduces abstractions that allow to generically implement the solver components while maintaining overall execution efficiency. We demonstrate the functionality of **CODD** on a variety of combinatorial optimization problems and compare its performance to other state-based solvers as well as integer programming and constraint programming solvers.

We consider discrete optimization problems of the form

$$\begin{array}{lll} P: & \max & f(x) \\ & \text{s.t.} & C_i(x), i = 1, \ldots, m, \\ & & x \in D \end{array}$$

where $x = (x_1, \ldots, x_n)$ is a tuple of decision variables, f is a real-valued objective function over x and $C_1, \ldots, C_m$ are constraints over x with $D = D_1 \times \ldots \times D_n$ denoting the cartesian product of the domains of the variables x_i $(1 \leq i \leq n)$.

1.2 Dynamic Programming as a Computational Model

Dynamic programming can be understood as a labeled transition system where sequences of decisions $(x_1, \ldots, x_n)$ lead to sequences of states $(s_1, \ldots, s_{n+1})$ with, at each step i, a transition $s_i \xrightarrow{x_i} s_{i+1}$ labeled by decision x_i. Each such transition induces a cost c_i. The DP formulation then boils down to:

- the definition of the state space $\mathcal{S}$ with $s_i \in \mathcal{S}$.

- two distinguished states $s_\top$ and $s_\perp$ in $\mathcal{S}$ encoding, respectively, the start state for an empty sequence of decision and the sink state for the full problem
- the defition of the labels used on transition and drawn from $\mathcal{L} = \bigcup_{i=1}^{n} D_i$
- The state transition function $\tau : \mathcal{S} \times \mathcal{L} \to \mathcal{S}$ modeling the decisions
- The transition cost function $c : \mathcal{S} \times \mathcal{L} \to \mathbb{R}$.

Then, the Dynamic Program over the state sequence $(s_1, \ldots, s_{n+1})$ and the decision sequence $(x_1, \ldots, x_n)$ is

$$\begin{array}{rll} \max & \sum_{i=1}^{n} c(s_i, x_i) & \\ \text{s.t.} & s_{i+1} = \tau(s_i, x_i) & \text{for all } x_i \in D_i, i = 1, \ldots, n, \\ & s_i \in \mathcal{S} & \text{for all } i = 1, \ldots, n+1. \end{array}$$

Observe that the constraints $(C_1, \ldots, C_m)$ are captured by membership in the state space. Indeed, each state $s_i \in \mathcal{S}$ is expected to satisfy $\bigwedge_{i=1}^{m} C_i$. In particular, the requirement is true for the root state $s_0 \in \mathcal{S}$ and is preserved by the transition function τ which, when starting from a state in $\mathcal{S}$, only consider *legal* labeled transition leading to members of $\mathcal{S}$. Clearly $s_1 = s_\top$ and s_{n+1} may be equal to $s_\perp$, a state satisfying all the constraints and corresponding to the full problem (all decisions were made).

The objective function simply accumulates the cost incurred along each transition and modeled by function c.

Path whole last state is $s_\top$ are feasible for the full problem P and the cost of the path is exactly the objective function. The path with the globally optimal cost is the global optimum to the original maximization problem P. Naturally, there are potentially exponentially many such path.

1.2.1 Exact Decision Diagrams

CODD provides (for generality's sake) an *exact* decision diagram that builds the state of the LTS just described and can therefore produce a globally optimal solution. While it is relatively direct, it requires the construction of an exponentially size structure and is therefore only useful as a proof of concept. Formally, the solution set of the exact decision diagram induced by $DD_{Exact}(P)$ is identical to the solution set of P, namely $\mathcal{S}\wr\updownarrow(DD_{Exact}(P)) = \mathcal{S}\wr\updownarrow(P)$.

1.2.2 Restricted Decision Diagrams

CODD provides *restricted* decision diagrams. Those differ from exact diagram by discarding (during their top-down construction) nodes whose presence would lead to overflowing a maximal imposed *width*. Since the approach throws away states, and therefore *paths* it runs the risk of loosing the optimal solution. Yet, if they yield paths leading to $s_\perp$, the best of them is a primal bound to the original problem. Formally, the solution set of the restricted decision diagram is a subset of the solution set of the original problem P. Namely, $\mathcal{S}\wr\updownarrow(DD_{Restricted}(P)) \subseteq \mathcal{S}\wr\updownarrow(P)$.

1.2.3 Relaxed Decision Diagrams

CODD provides *relaxed* decision diagrams. Those differ from exact diagram by *merging* states whose presence would induce a diagram *width* exceeding a given bound. The state merge operator must be a legit relaxation in the following sense: it yields a state that no longer satisfies all constraints in $(C_1, \ldots, C_m)$. While this measures ensure that the size of the diagram remains under control, it *introduces* new states that are no longer satisfiable. Path leading to $s_\perp$ going through at least one such unsatisfiable state are no longer modeling a solution of the original problem. Formally, the solution set of the relaxed diagram is now a super set of the solution set of the original problem P. Namely, $\mathcal{S}\wr\updownarrow(DD_{Relaxed}(P)) \supseteq \mathcal{S}\wr\updownarrow(P)$.

From the above, one get derive primal bounds using the restricted diagrams and dual bounds using the relaxed one.

Note how all three diagrams are based on the same LTS abstractions of states, labels, transitions, merge and costs and equality to $s_\perp$. These abstractions are the core modeling facilities offered by **CODD**.

1.3 CODD Modeling

Unsurprisingly, the abstractions presented above match exactly with the **CODD** API that expects:

- A structure to define a state
- Two distinguished states $s_\top$ and $s_\perp$ used to represent a state with no decisions made ($s_\top$) and all possible decision made ($s_\perp$).
- A label generation function which, given a state s_i produces the set of potentially viable transitions

- A state transition function τ which, given a state s_i and a potentially viable label ℓ produces either nothing ($\perp$) or a new state s_{i+1} such that $s_i \xrightarrow{x_i=\ell} s_{i+1} \in \tau$.

- A cost function c which, given a state s_i and a viable label ℓ produces the cost of the transition $s_i \xrightarrow{x_i=\ell} s_{i+1}$

- An equality to $s_\perp$ function that returns true whenever its input state *is* $s_\perp$.

An *optional* local dual bounding function which, given a state s_i compute a coarse dual bound for completing s_i. *this optional abstraction* is helpful to quickly assess whether a state has any hope of leading to an improving $s_\perp$. If a state does not, it can be discarded from the relaxation. Thankfully, C++ is a rich language supporting polymorphic types, first order and higher order functions. Those provide the basis to naturally convey the formal abstractions given above. The *state* become a *type* in C++ and all the other abstractions becomes first-order functions (lambdas in C++ parlance).

1.3.1 TSP Example High Level

Consider the task of solving instances of the traveling salesman problem. With the abstraction defined above, to express a TSP over a set of cities $C = \{1..n\}$ we define:

- Let a state $s \in \mathcal{S}$ be a triplet $\langle V, l, h\rangle$ with $V \subseteq C$ the set of cities visited thus far, l the name of the city last visited (or 0 at the start), and h the number of hops made thus far (or 0 at the start).

- Let $s_\top = \langle \emptyset, 0, 0\rangle$ since no cities have been visited, hence last city is the fake city 0 and the salesman did not do any "hops"

- Let $s_\perp = \langle V, 1, n\rangle$. Indeed, without loss of generality, we will start from city 1 and thus *return* to city 1 (which will thus be the last visited). The trip will have carried out n hops and visited all cities in V.

- The label generator is a function $L : \mathcal{S} \to \mathcal{L}(\langle V, l, h\rangle) = \{1\} \cup C \setminus V$

- The function $\tau : \mathcal{S} \times \mathcal{L} \to \mathcal{S}$ is

$$
\begin{array}{rcll}
\tau(\langle V,l,h\rangle,\ell) & = & \bot & \Leftrightarrow \ell = 1 \wedge h < n-1 \\
 & & \bot & \Leftrightarrow \ell \neq 1 \wedge h \geq n-1 \\
 & & \langle C,\ell,h+1\rangle & \Leftrightarrow \ell \notin V \wedge \ell \neq l \wedge \ell = 1 \wedge h \geq n-1 \\
 & & \langle V \cup \{\ell\},\ell,h+1\rangle & \Leftrightarrow \ell \notin V \wedge \ell \neq l \wedge h < n \\
 & & \bot & \Leftrightarrow \text{otherwise}
\end{array}
$$

The condition $\ell = 1 \wedge h < n-1$ identifies an attempt to return to the first city too early. Likewise, the condition $\ell \neq 1 \wedge h \geq n-1$ is indicative of an attempt to build extend a tour with more than $n-1$ hops without returning to the initial city. The third case is a valid attempt at closing a tour with $n-1$ hops by returning to the first city. The fourth case is the vanilla situation of extending a known tour with one more hop.

1.3.2 TSP Example in CODD

Modeling with **CODD** first requires the addition of a type to represent a state. Since a state is a triplet, the model uses a `C` structure.

[]c++ struct TSP GNSet s; int last; int hops; friend std::ostream operator«(std::ostream os,const TSP m) return os « "<" « m.s « ',' « m.last « ',' « m.hops « ">"; ; Note the additional output operator that is used to inspect state. In addition to the state, **CODD** expects to provide 2 standard operations on state. Equality testing and hashing. Namely, []c++ template<> struct std::equal$_t o < TSP > constexprbooloperator()(constTSPs1, constTSPs2)constreturns1.last =$
Is a STL compliant implementation of equality testing for a type `T` which, here, is none other than `TSP`. Likewise, the STL compliant fragment below []c++ template<> struct std::hash<TSP> std::size$_t operator()(constTSPv)constnoexceptreturn(std :$
defines a `hash` operator for our state type `TSP`. Both state equality and state hashing are used internally by **CODD**.

It is now possible to define both $s_\top$ and $s_\bot$ with: []c++ const auto init = []() // The root state return TSP GNSet,1,0 ; ; const auto target = [C,sz]() // The sink state return TSP C,1,sz ; ; Note how the `init` and the `target` closures can be called to manufacture the desired states.

The label generation function is also defined with a simple closure []c++ const auto lgf = [C](const TSP s) auto r = C; r.diffWith(s.s); r.insert(1); return r; ; The body of the closure uses the capture set of cities `C` and removes from it those already visited (and held in `s.s`) but retains the first city (1) to be able to close the tour.

The function τ is, unsurprisingly, another closure (which captures `sz` and `C` denoting, respectively, the number of cities and the set of cities.) []c++ const auto stf = [sz,C](const TSP s,const int label) -> std::optional<TSP> bool bad = (label == 1 s.hops < sz-1) || (label != 1 s.hops >= sz-1); if (bad) return std::nullopt; else bool close = label==1 s.hops >= sz-1; if (close) return TSP C,1,sz; else if (!s.s.contains(label) s.last != label) return TSP s.s | GNSetlabel,label,s.hops + 1; else return std::nullopt; ; The case analysis carried out in the code mirrors exactly the formal definition. The `bad` Boolean matches the first two conditions of τ and the code return `std::nullopt` to report $\perp$. The `else` block cover the two main cases (good closing of a tour, or extension with a city). Note how a C++ expression like `s.s | GNSet{label}` is the direct encoding of $V \cup \{\ell\}$ as the `|` operator is used for set union.

The cost of a transition is also modeled with a closure that captures the edge set `es` and reads: []c++ const auto scf = [es](const TSP s,int label) // partial cost function return es.at(GE s.last,label); ; It simply looks up in the edge *map* `es` the weight associated to the edge originating at vertex `s.last` and ending at vertex `label`.

The state merge capability to support relaxed diagram is simply: []c++ const auto smf = [](const TSP s1,const TSP s2) -> std::optional<TSP> if (s1.last == s2.last s1.hops == s2.hops) return TSP s1.s s2.s , s1.last, s1.hops; else return std::nullopt; // return the empty optional ; Observe how this closure takes two states s_1 and s_2 and considers them mergeable if they both end in the same city and have the same number of hops. The relaxation stems from the fact that the set *intersection* between $s_1.s$ and $s_2.s$ will only retain cities that were visited in both.

Finally, the equality to the sink (target) state is another closure: []c++ const auto eqs = [sz](const TSP s) -> bool return s.last == 1 s.hops == sz; ; Which deems state s equal to the sink if the last city is 1 and we have the desired number of hops sz.

1.4 CODD Solving

Solving the TSP then reduces to *instantiating* the generic solver with all the closures defined earlier. Namely: []c++ BAndB engine(DD<TSP,Minimize<double>, // to minimize decltype(target), decltype(lgf), decltype(stf), decltype(scf), decltype(smf), decltype(eqs) >::makeDD(init,target,lgf,stf,scf,smf,eqs,labels),1); engine.search(bnds); Note how the `DD` template is specialized with the state type `TSP`, the type `Minimize<double>` to convey that this is a minimization, and the types of the various closures using the C++-23 `decltype` operator. The solver is invoked with the last line.   Note that **CODD** users never directly call any of those closures. Calls to the closures are choreographed by the solver to build the diagrams and reason with them. **CODD** users are strictly focused on defining the DP LTS in a mathematical sense and then doing the 1-1 translation to C++.

2 Installing CODD

2.1 Download

The CODD solver is available on github.

2.2 Compilation

The CODD C++ library implements both the modeling and the solving framework. It extensively relies on functional closures to deliver concise, declarative and elegant models. It has:

- Restricted / exact / relax diagrams
- State definition, initial, terminal, tranistion and state merging functions separated
- Label generation functions
- Equivalence predicate for sink.

It allows to define model to solve problems using a dynamic programming style with the support of underlying decision diagrams.

2.2.1 Dependencies

You need `graphviz` (The `dot` binary) to create graph images. It happens automatically when the `display` method is called. Temporary files are created in `/tmp` and the macOS `open` command is used (via `fork/execlp`) to open the generated PDF. The rest of the system is vanilla C++-23 and `cmake`. Keep in mind that this is optional and only needed if you wish to generate images of diagrams (typically to debug models).

2.2.2 C++ Standard

You need a C++-23 capable compiler. gcc and clang should both work. I work on macOS where I use the mainline clang coming with Xcode. The implementation uses templates and concepts to factor the code.

2.2.3 Build system

This is `cmake`. Simply do the following []bash mkdir build cd build cmake .. make -j4 And it will compile the whole thing. To compile in optimized mode, simply change the variable `CMAKE_BUILD_TYPE` from `Debug` to `Release` as shown below: []bash cmake .. -DCMAKE$_BUILD_TYPE = Release$

2.2.4 Unit tests

oIn the `test` folder. Mostly for the low-level containers.

2.2.5 Library

All of it in the `src` folder. []shell scc -i cpp,org,h,hpp ..

```
Language                 Files     Lines   Blanks  Comments     Code Complexity

C++                         29      4288      298       274     3716        670
C++ Header                  15      3098      132       338     2628        431
Org                          3       778       99         0      679        100

Total                       47      8164      529       612     7023       1201

Estimated Cost to Develop (organic) $209,143
Estimated Schedule Effort (organic) 7.59 months
Estimated People Required (organic) 2.45

Processed 270533 bytes, 0.271 megabytes (SI)
```

2.3 Examples

To be found in the `examples` folder

- `coloringtoy` tiny coloring bench (same as in python)
- `foo` maximum independent set (toy size)
- `tstpoy` tiny TSP instance (same as in python)
- `gruler` golomb ruler (usage <size> <ubOnLabels>)

- `misp` the maximum independent set problem

3 The Maximum Independent Set Problem (MISP)

It is, perhaps, most effective to look at some models to get a reasonable sense of the effort it takes to model and solve a problem with **CODD**.

3.1 Preamble

To start using CODD, it is sufficent to include its main header as follow []c++ include "codd.hpp"

3.2 Reading the instance

[]c++ struct GE int a,b; friend bool operator==(const GE e1,const GE e2) return e1.a == e2.a e1.b == e2.b; friend std::ostream operator«(std::ostream os,const GE e) return os « e.a « "–>" « e.b; ;

struct Instance int nv; int ne; std::vector<GE> edges; FArray<GNSet> adj; Instance() : adj(0) Instance(Instance i) : nv(i.nv),ne(i.ne),edges(std::move(i.edges)) GNSet vertices() return setFrom(std::views::iota(0,nv)); auto getEdges() const noexcept return edges; void convert() adj = FArray<GNSet>(nv+1); for(const auto e : edges) adj[e.a].insert(e.b); adj[e.b].insert(e.a); ;

Instance readFile(const char* fName) Instance i; using namespace std; ifstream f(fName); while (!f.eof()) char c; f » c; if (f.eof()) break; switch(c) case 'c': std::string line; std::getline(f,line); break; case 'p': string w; f » w » i.nv » i.ne; break; case 'e': GE edge; f » edge.a » edge.b; edge.a–,edge.b–; // make it zero-based assert(edge.a >=0); assert(edge.b >=0); i.edges.push$_back(edge); break; f.close(); i.convert(); return i;$

The C structure `GE` is meant to represent a *graph edge*. It inlines a friend function to print edges and an equality operator. The C struct `Instance` is used to encapsulate an instance of the MISP problem read from a text file. It holds the number of vertices `nv`, the number of edges `ne`, the list of `edges` and computes and holds the adjacency list `adj`. The latter is computed by the `convert` method which simply scans the edges in `edges` and adds the endpoints in the respective sets of the adjacency vector. Note that the vertices are numbered from 0 onward (so the last vertex number is `nv - 1`).

The `readFile` function produces an `Instance` from a named file `fName`. Note how it shifts the vertex identifiers of edges down by 1 since the standard instances use a 1-based numbering scheme rather than a 0-based numbering scheme.

3.3 State definition

[]c++

struct MISP GNSet sel; int n; friend std::ostream operator«(std::ostream os,const MISP m) return os « "<" « m.sel « ',' « m.n « ',' « ">"; ;

template<> struct std::equal$_to < MISP > booloperator()(constMISPs1, constMISPs2)constret$

template<> struct std::hash<MISP> std::size$_toperator()(constMISPv)constnoexceptreturnstd ::$

The `MISP` struct defines the state representation for the DP model. For the *maximum independent set problem* the state is simply a set of integers named `sel` and an integer `n` representing the index of the next vertex to consider for inclusion (or exclusion) from the independent set. The next two classe are standard C++ and define the following:

- `std::equal_to<MISP>`: this structure conforms to the STL and defines as equality operator over the state `MISP`
- `sth::hash<MISP>`: this structure conforms to the STL and defines a hash function for

  the state `MISP`. Note how it uses the hash functions for the `int` type and the `GNSet` types provided by the STL or the `CODD` library.

3.4 Main Model

3.4.1 Getting started

[]c++ int main(int argc,char* argv[]) // using STL containers for the graph if (argc < 2) std::cout « "usage: coloring <fname> <width>"; exit(1); const char* fName = argv[1]; const int w = argc==3 ? atoi(argv[2]) : 64; auto instance = readFile(fName);

const GNSet ns = instance.vertices(); const int top = ns.size(); const std::vector<GE> es = instance.getEdges();

const auto labels = ns | GNSet top ; // using a plain set for the labels std::vector<int> weight(ns.size()+1); weight[top] = 0; for(auto v : ns) weight[v] = 1; ...

The `main` program simply gets the filename from the command line and reads the instance from the file. It then extract in `ns` the set of vertices, in `top` its cardinality and in `es` the list of edges. The last four lines define the `labels` to be used (the identifier of all vertices together with `top` to encode the transition to the final state in the decision diagram). They also define the weights of the vertices. Since the instances are cliques (from the DIMACS challenge), the `weight` of every vertex is 1 while the `weight` of `top` is 0.

3.4.2 The Bound Tracker

The maximum independent set is an optimization (maximization) problem. CODD needs to track solutions as they get produces and offers the opportunity to execute an arbitrary code fragment when a new solution somes forth. This code fragment can be used, for instance, to check the correctness of the solution.

This task is the responsibility of the `Bounds` object. Minimally, one simply must declare an instance as follows: []c++ Bounds bnds;

In the MISP example, we illustrate how to respond to incoming solutions. In this case the `Bounds` instance uses a C++ lambda (a closure) that will be fed a solution `inc`, i.e., a vector of labels.

Consider the example below: []c++ Bounds bnds([es](const std::vector<int> inc) bool ok = true; for(const auto e : es) bool v1In = (e.a < (int)inc.size()) ? inc[e.a] : false; bool v2In = (e.b < (int)inc.size()) ? inc[e.b] : false; if (v1In v2In) std::cout « e « " BOTH ep in inc: " « inc « ""; assert(false); ok = !(v1In v2In); std::cout « "CHECKER is " « ok « "";); The closure first *captures* the set of edges (by reference) as checking the validity of a solution simply entails looping over all edges and making sure that not both endpoints of an edge are included in the solution. The `for` loop binds `e` to an edge and, provided that the endpoints are mentioned in the solution, looks up the Boolean associated to the vertex in the solution. Note how the solution `inc` can be a prefix of the full vertex list (hence the conditional expression). If both endpoints are mentionned in the solution, the computation is aborted as this would indicate a bug in the model.

3.4.3 Defining neighbors

The main model will make use of the adjacency list, so it is advisable to hold into a variable `neighbors` the adjacency list for the graph. []c++ auto neighbors = instance.adj;

3.4.4 The actual CODD Model

A CODD model capture a label transition system (LTS). This LTS operates on nodes holding states for the problem. In the case of the maximum independent set, the states are `MISP` instances. The LTS starts from a *source* node and forms paths that ultimately target a *sink* state. A path in the LTS moves from state to state by *generating labels* and using a *transition* function. Each such transition can incur a *cost*.

The CODD solver uses a branch & bound strategy with both a primal and a dual bound. Primal bounds are produced as a matter of course each time an incumbent solution is found, but also through the used of **restricted decision diagrams**. Likewise, dual bounds are produced by **relaxed decision diagrams**. Such relaxed diagrams rely on *merge* operations to collapse state.

CODD models all the italicized concepts outlined in the prior paragraphs with C++ closures. The remainder of this section presents them, one at a time.

1. The Start State Closure The root, start or source state in the MISP application simly holds in the `sel` property of the state the indices of all legal vertices and holds in `n` the value 0 to report that the next decision is to be about vertex 0. Note how the code below uses the STL `std::views::iota` to loop over the closed range [0,top] and insert each value `i` in the set `U` that is then used to create and return the root state.

   []c++ const auto myInit = [top]() // The root state GNSet U = ; for(auto i : std::views::iota(0,top)) U.insert(i); return MISP U , 0; ;

2. The Sink State Closure The sink state is chosen, by convention, to hold an empty set for the remaining legal vertices (so no more decision beyond this point) and `top` as the next vertex to consider since top is the index of the last vertex plus 1. Note how the closure capture the `top` local variable. []c++ const auto myTarget = [top]() // The sink state return MISP GNSet ,top; ;

3. The Label Generation Closure Moving from one state to the next involves making a decision about the next vertex to be consider for inclusion. Observe that the identity of that vertex is held in the `n` property of the state we are departing. The decision, in this case, is a binary choice. Either we include `n` or we do not. So the label generation function returns the closed range [0,1] as the valid outgoing labels. Note how the closure takes as input the source state `s` (yet, for the MISP model, the source state is not used for any purposes.). []c++ const auto lgf = [](const MISP s) return Range::close(0,1); ;

4. The State transition Closure the state transition closure is the heart of the model. It specifies what state to go to when leaving `s` through label `label`. Observe that `s` dictates which vertex to consider in `s.n`. Two cases arise:

- $s.n \geq top$ In this situation, we ran out of vertices. If the remaining legal set is empty ($s.sel = \emptyset$) then we ought to transition to the sink as we are closing a viable path. Otherwise, this is a "dead-end" and the code return nothing (`std::nullopt`) as the API uses the C++ optional type to convey the absence of a transition.
- $s.n < top$ In this situation, we can decide to include `s.n` in the MISP provided that it is still legal (i.e., provided that $s.n \in s.sel$). The first line of the second case therefore commits to not transitioning whenever $s.n \notin s.sel \wedge label = True$ and thus return `std::nullopt`. Otherwise, `s.n` is eligible (because it is either to be excluded, or it is still legal) and the new state is computed. The new state $out = s.sel \setminus N(s.n) \setminus \{s.n\}$ where $N(s.n)$ refers to the neighbors of $s.n$. The `diffWith` method implements the set difference calculation. Finally, the result state holds `out` and sets the next vertex to consider to be *top* if $out = \emptyset$ or $s.n + 1$ otherwise.

[]c++ auto myStf = [top,neighbors](const MISP s,const int label) -> std::optional<MISP> if (s.n >= top) if (s.sel.empty()) return MISP GNSet , top; else return std::nullopt; else if (!s.sel.contains(s.n) label) return std::nullopt; GNSet out = s.sel; out.remove(s.n); // remove n from state if (label) out.diffWith(neighbors[s.n]); const bool empty = out.empty(); // find out if we are done! return MISP std::move(out),empty ? top : s.n + 1; // build state accordingly ;

5. The transition Cost Closure Each transition from a state to another incurs a cost based on the source state and the label used to transition out. CODD once again relies on a closure to report this cost. In the code fragment below, note how the closure capture a reference to the `weight` vector and uses the identity of the vertex being labeled `s.n` as well as the `label` itself (a 0/1 value) to compute and return the actual cost. []c++ const auto scf = [weight](const MISP s,int label) // cost function return label * weight[s.n]; ;

6. The State Merge Closure CODD computes its dual bound with a relaxation that *merges* state. The implementation uses a closure which, given two states s_1 and s_2 determines whether the two states are mergeable and returns a new state if they are, or the `std::nullopt` optional if they are not.

   In the MISP case, states are always mergeable and the merge result is none other than the union of the two eligible sets of vertices and the minimum identifier for the next vertex. In the code below the C++ operator `|` conveys the union of the two sets. []c++ const auto smf = [](const MISP s1,const MISP s2) -> std::optional<MISP> return MISP s1.sel | s2.sel,std::min(s1.n,s2.n); ;

7. Local Bound Closure CODD can use an *optional* closure to quickly compute dual bounds associated to states of the LTS. The optional `local` closure is therefore typically lightweight. In the case of MISP, a simple dual bound consist of summing up the weight of all the vertices that are still eligible. This is clearly an overestimate as some of these vertices will be ruled out. But it's cheap to compute! []c++ const auto local = [weight](const MISP s) -> double return sum(s.sel,[weight](auto v) return weight[v];); ;

8. Recognizing the Sink CODD uses one last (mandatory) closure to establish equality to the sink. It does not rely on the equality operator as recognizing the sink may not require to test all its attributes for equality, but only a subset of them. []c++ const auto eqs = [](const MISP s) -> bool return s.sel.size() == 0; ;

9. Wrapping up Now that all the mandatory (and optional) closures are defined, it only remains to instantiate the generic solver with the closures given above and invoke it.

   The type `Maximize<double>` is used to convey that this is a maximization problem while the nested `double` type is the type used to track the objective function value and is currently always `double`. CODD supports `Minimize<double>` as well.

   []c++ BAndB engine(DD<MISP,Maximize<double>, // to maximize decltype(myTarget), decltype(lgf), decltype(myStf), decltype(scf), decltype(smf), decltype(eqs), decltype(local) >::makeDD(myInit,myTarget,lgf,myStf,scf,smf,eqs,labe
   engine.search(bnds); return 0;

4 Papers

5 Related Systems

5.1 DDO

DDO was created by Pierre Schauss and Xavier Gillard. It is a generic and efficient framework for MDD-based optimization written in Rust.

5.2 Domain Independent DP

DIDP is the brainchild of Ryo Kuroiwa, J. Christopher Beck. It can be found here and offers both a modeling and a solving API for dynamic programming.