Choco-Graph : a module for graph variables in the Choco CP solver

Jean-Guillaume Fages Charles Prud'homme Xavier Lorca *

September 16, 2014

Contents

*ChocoTeam, choco3-support@mines-nantes.fr, TASC research team

1 Overview

This Choco module allows you to search for a graph[1], which may be subject to constraints. The domain of a graph variable $\mathcal{G}$ is a graph interval $[\underline{\mathcal{G}}, \overline{\mathcal{G}}]$. $\underline{\mathcal{G}}$ is the graph representing vertices and edges which must belong to any single solution whereas $\overline{\mathcal{G}}$ is the graph representing vertices and edges which may belong to one solution. Therefore, any value $\mathcal{G}^*$ must satisfy the graph inclusion $\underline{\mathcal{G}} \subseteq \mathcal{G}^* \subseteq \overline{\mathcal{G}}$. One may see a strong connection with set variables. A graph variable can be subject to graph constraints to ensure global graph properties (e.g. connectedness, acyclicity) and channeling constraints to link the graph variable with some other binary, integer or set variables. The solving process consists of removing nodes and edges from $\overline{\mathcal{G}}$ and adding some others to $\underline{\mathcal{G}}$ until having $\underline{\mathcal{G}} = \overline{\mathcal{G}}$, i.e. until $\mathcal{G}$ gets instantiated. These operations stem from both constraint propagation and search. You may wonder why using a graph variable. Here are the most important motivations to do so:

- Modeling convenience :
  - When solving a graph problem, the model is close to the original problem.
  - A graph variable is a consistent graph representation (e.g. a node which does not exist has no incident arcs), which simplifies the model.
  - Stating constraints as graph properties in a declarative way is nice.
- Implementation convenience :
  - Manipulating a domain which consists of two graphs (representing respectively mandatory and potential elements) makes easy the implementation of graph-based filtering algorithms. As the implementation becomes more natural, the risk of mistakes decreases.
- Performance gains :
  - You can use optimized data structure for domains (e.g. bit sets, bipartite sets, linked lists...) which allow to reduce runtime of most algorithms and/or memory consumption. This brings significant improvement on large scale problems.
  - Having such a global variable instead of many smaller ones makes the solver lighter, which brings performance improvement.

This module can be seen as a CP framework for graph theory. In that sense, it is closer to Operations Research than Artificial Intelligence. With a minimum background in graph algorithms, this manual will help you to build your own models, constraints and search procedures so that you effectively solve your problems.

This module has been introduced in 2011 but it has been deeply refactored in September 2014. In case some bugs have been introduced during that step, please inform us.

[1] Either directed or undirected, with at most one arc/edge between any two vertices.

2 Defining a graph variable

Prior to introduce how to build graph variables, it is necessary to describe graph structures of the core Choco solver. These graphs are often used as internal data structures for propagators. In our case, it will serve to define the domain of graph variables.

2.1 Graphs

There are basically two kind of graphs in Choco : directed (`DirectedGraph.java`) and undirected (`UndirectedGraph.java`) graphs. Directed and undirected graphs have similar methods. The constructor of an undirected graph is the following:

```
public UndirectedGraph(int n, SetType type, boolean allNodes)
```

- The integer `n` denotes the maximum number of nodes. This is necessary for memory allocation. The set of nodes is then a subset of $\{0, 1, ..., n-1\}$. Once the graph has been created, it is not possible to modify that value.
- `SetType type` indicates which kind of data structure to use. If the graph is very sparse, a linked list (`SetType.LINKED_LIST`) implementation would reduce the memory consumption. Otherwise, `SetType.BIPARTITESET` provides an optimal time complexity for every request, but has some hidden constants and a higher memory consumption. `SetType.BITSET` is a good default choice.
- The boolean `allNodes` indicates whether or not the node set is fixed. This parameter is very important. Whenever set to true, it means that the vertex set is $[0, n-1]$ and will not change during search. It is not necessary to add vertices explicitly (all of them are present). If set to false, it means that the vertex set must be a subset of $[0, n-1]$, and is initially EMPTY. Therefore, the user may have to add them explicitly by using the `addNode(int i)` method.

Graph manipulations (iterations over nodes, edges, neighbors of a particular vertex...) rely on the choco `ISet` interface.

```
ISet nodes = graph.getNodes();
for(int i=nodes.getFirstElement(); i>=0; i = nodes.getNextElement())
```

Note that for efficiency reason iteration of one `ISet` is not context safe, i.e. you cannot encapsulate two iteration loops of the same set (copy the set in an int[] for doing that).

2.2 Backtrackable graphs

As we wish to use graphs to represent the domain of a variable, such graphs must be backtrackable, i.e. the graph must restore its previous value upon backtracking. To do so, one simply use a different constructor, having the solver in argument (to catch backtrack events):

```
public UndirectedGraph(Solver solver, int n, SetType type, boolean allNodes)
```

2.3 Graph variable

Graph variables can be created through `GraphVarFactory.java`. The domain of such a variable is defined by two graphs : the lower bound graph gives nodes and arcs that belong to every solution, whereas the upper bound graph gives nodes and arcs that may belong to a solution.

```
/**
 * Create an undirected graph variable named NAME
 * and whose domain is the graph interval [LB,UB]
 * BEWARE: LB and UB graphs must be backtrackable
 * (use the solver as an argument in their constructor)!
 *
 * @param NAME      Name of the variable
 * @param LB        Undirected graph representing mandatory nodes and edges
 * @param UB        Undirected graph representing possible nodes and edges
 * @param SOLVER    Solver of the variable
 * @return  An undirected graph variable
 */
public static IUndirectedGraphVar undirected_graph_var(String NAME,
                                        UndirectedGraph LB,
                                        UndirectedGraph UB,
                                        Solver SOLVER) {
    return new UndirectedGraphVar(NAME, SOLVER, LB, UB);
}

/**
 * Create a directed graph variable named NAME
 * and whose domain is the graph interval [LB,UB]
 * BEWARE: LB and UB graphs must be backtrackable
 * (use the solver as an argument in their constructor)!
 *
 * @param NAME      Name of the variable
 * @param LB        Directed graph representing mandatory nodes and edges
 * @param UB        Directed graph representing possible nodes and edges
 * @param SOLVER    Solver of the variable
 * @return  An undirected graph variable
 */
public static IDirectedGraphVar directed_graph_var(String NAME,
                                    DirectedGraph LB,
                                    DirectedGraph UB,
                                    Solver SOLVER) {
    return new DirectedGraphVar(NAME, SOLVER, LB, UB);
}
```

Note the the bound graphs must be able to restore their value upon backtracking. Therefore, you should use the following signatures:

```
new UndirectedGraph(Solver solver, int n, SetType type, boolean allNodes)
new DirectedGraph(Solver solver, int n, SetType type, boolean allNodes)
```

The input maximum number of nodes should be the same for both the lower and the upper bound graphs. Here is an example involving an undirected graph variable:

```
// graph variable domain
UndirectedGraph GLB = new UndirectedGraph(
                                        solver,          // Restore value on backtrack
                                        n,               // Maximal number of nodes
                                        SetType.BITSET,  // data structure type
                                        false            // fixed node set?
                                    );
UndirectedGraph GUB = new UndirectedGraph(
                                        solver,          // Restore value on backtrack
```

```
                                        n,                  // Maximal number of nodes
                                        SetType.BITSET,     // data structure type
                                        false               // fixed node set?
                                    );
for (int i = 0; i < n; i++) {
    GUB.addNode(i);             // potential node
    for (int j = i; j < n; j++) {
        if (link[i][j]) {       // some input data providing potential edges
            GUB.addEdge(i, j);      // potential edge
        }
    }
}
GLB.addNode(1);             // 1 and 2 must belong to the solution
GLB.addNode(2);
GLB.addEdge(1,2);                   // 1 and 2 must belong to the same clique
// graph variable
graphvar = GraphVarFactory.undirected_graph_var("G", GLB, GUB, solver);
```

In this example, we see that vertices 1 and 2 must belong to every solution, as well as the edge $(1, 2)$. Other potential vertices and edges are given by `GUB`.

For simplicity reasons, one may prefer to use the following method, which creates an empty lower bound graph and a complete upper bound graph with 42 vertices:

```
GraphVarFactory.undirected_graph_var("G",42,solver);
```

3 Constraining a graph variable

A collection of constraints over a graph variable can be found in `GraphConstraintFactory.java`. The name is explicit but a bit long. You can either import it statically or use `GCF.java` as a shortcut.

3.1 Usual graph constraints

3.1.1 Node and edge counts

The factory contains several basic constraints, such as `nb_nodes`, which enables to constrain the number of nodes to be equal to a given integer variable. To make things simpler, you can call the `GraphVarFactory.nb_nodes(g)` function which will create and return an integer variable that is equal to the number of nodes (i.e. it posts the `nb_nodes` constraint). In the same way, one can count the number of edge (resp. arc) of an undirected (resp. directed) graph variable as follows:

```
IntVar nbArcs = GraphVarFactory.nb_arcs(g);
```

3.1.2 Loops

Graph variables may contain loops, i.e. arcs of the from (i, i). If you want the graph to contain no loops, then you should simply make sure the graph upper bound has initially no loop. Instead, if you wish some vertices to have a loop, then you can use the `loop_set(g,l)` constraint which ensures that the set variable l represents the nodes of g that have a loop. You can also directly create that set variable using:

```
SetVar loops = GraphVarFactory.loop_set(graphvar);
```

Finally, you can control the number of loops the graph variable has with an integer variable with the following :

```
IntVar nbLoops = GraphVarFactory.nb_loops(g);
```

3.1.3 Degrees

It is possible to constrain the minimum and the maximum degree each node of an undirected graph variable, by using respectively `min_degrees` and `max_degrees` constraints. Such constraints only hold on vertices that belong to the solution. For instance, if vertex a is constrained to have a degree greater than 5 but has only 4 potential neighbors, then vertex a should be removed from the potential vertex set. Unless a was a mandatory vertex, this does not trigger any failure. Here is an example imposing every vertex to have at most 5 neighbors:

```
solver.post(GraphConstraintFactory.max_degrees(graph,5));
```

It is also possible to constrain the exact degree of every node with an integer variable, thanks to the `degrees` constraint. Instead of the above, this constraint holds on every vertex. Therefore, a vertex which does not belong to the potential vertex set

should have its degree variable equal to 0. You can create these degree variables simply as follows:

```
IntVar[] degrees = GraphVarFactory.degrees(g);
```

In the same way, one can restrict the in-degree (number of predecessors) and out-degree (number of successors) of each node of a directed graph variable.

3.1.4 Graph inclusion

The `subgraph(g1,g2)` constraint enables to state that $g1$ is a subgraph of $g2$, i.e. every vertex and edge in $g1$ is also in $g2$. It follows that $g1$ cannot be larger than $g2$.

3.1.5 Symmetry

You can force a directed graph variable to be either symmetric or antisymmetric, by respectively using the `GCF.symmetric(g)` or the `GCF.antisymmetric(g)` constraints. For instance, by posting the following constraint you make sure that for any arc $(i, j) \in g$, then $(j, i) \notin g$.

```
solver.post(GCF.antisymmetric(g));
```

3.1.6 Transitivity

Transitivity is a useful property which enables to compute transitive closures and cliques.

```
solver.post(GCF.transitivity(g));
```

3.1.7 Cycles

To constrain an undirected graph variable to form a (Hamiltonian) cycle, then you can simply use the `cycle` (`hamiltonian_cycle`) constraint:

```
solver.post(GraphConstraintFactory.cycle(g));
```

In the same way, a directed graph variable can be forced to form a circuit.

```
solver.post(GraphConstraintFactory.hamiltonian_circuit(g));
```

You can prevent a directed (resp. undirected) graph from containing any circuit (resp. cycle) by posting the `no_circuit` (resp. `no_cycle`) constraint, as follows:

```
solver.post(GraphConstraintFactory.no_circuit(g));
```

3.1.8 Connectivity

It is possible to force an undirected (resp. directed) graph variable to be connected (resp. strongly connected) or even to control its number of connected (resp. strongly connected) components with an integer variable. The filtering of such constraint is quite weak but fast.

Here is an example :

```
IntVar nbSCC = VariableFactory.fixed(2,solver);
solver.post(GraphConstraintFactory.nb_strongly_connected_components(g,nbSCC));
```

3.1.9 Tree

You can force an undirected graph variable to form a tree (i.e. a connected acyclic graph) or a forest (i.e. an acyclic but potentially disconnected graph) by posting the respective constraints:

```
solver.post(GCF.tree(graphvar));
solver.post(GCF.forest(graphvar));
```

In the case of directed graph variable, you can also have directed trees or directed forests (also called arborescences).

```
IntVar root = VF.enumerated("rootOfTree",0,n-1,solver);
solver.post(GCF.directed_tree(graphvar,root));
solver.post(GCF.directed_forest(graphvar));
```

Note that the `directed_tree` constraint requires an integer variable denoting the root of the tree, i.e. the vertex which has no predecessor and from which all nodes can be reached.

3.1.10 Cliques

It is possible to partition a graph into cliques by using the transitivity and connectivity constraints. Nevertheless, if the maximum number of cliques is small, a stronger filtering is provided by the `nb_cliques(g,nb)` constraint.

3.1.11 Diameter

You can impose the diameter of a graph variable to be equal to a given integer variable with the `diameter` constraint. This constraint also forces the graph variable to be connected (or strongly connected in case of a directed graph variable). As a recall, the diameter is the length (in number of arcs) of the largest shortest path between any pair of nodes.

3.2 Some optimization constraints

Solving hard optimization problems to optimality often requires to embed cost-based reasonings into global constraints. We have included two minimum spanning tree relaxations : the one-tree Lagrangian relaxation to solve the Traveling Salesman Problem and a minimum spanning tree subject to (dualized) degree constraints, to solve the

more general Degree Constrained Minimum Spanning Tree Problem. Such constraints introduce a significant overhead but they provide a very powerful filtering as well. Presumably, they should only be used once a good upper bound has been found (in case of a minimization problem), because the filtering depends on that value.

3.2.1 The TSP constraint

The TSP constraint enables to find a Hamiltonian cycle of minimum cost. It is built as follows:

```
// constraints (TSP basic model + Lagrangian relaxation)
solver.post(GraphConstraintFactory.tsp(graph, totalCost, costMatrix, 1));
```

The arguments of this methods are respectively : the undirected graph variable representing the cycle, the integer variable representing the cost of the cycle, the integer (symmetric) cost matrix and the Lagrangian mode. Three values are possible for that parameter : 0 means the Lagrangian relaxation is not used; 1 means that the Lagrangian relaxation is turned on after a first solution has been found; 2 means the Lagrangian relaxation is used since root node.

3.3 Channeling constraints

A wide range of channeling constraints are provided to allow links between boolean, integer or set variables and a graph variables. This enables to post some usual constraints over some vertex (sub)sets of some edge (sub)sets.

Note that you do not have to create such channeling variables yourself : `GraphVarFactory.java` does it for you! See for instance the static method `nodes_set` which creates a set variables associates to the nodes of the graph variable given in parameter. Here is an example showing how to constrain the number of vertices of a graph variable g:

```
SetVar vertices = GraphVarFactory.nodes_set(g);
IntVar card = VF.fixed(3,solver);
solver.post(SCF.cardinality(vertices, card));
```

In the same way, one can want to constraint outgoing (resp. ingoing) arcs of a vertex, by extracting such arcs in a set variable.

3.3.1 Set channeling

A set variable can be associated with:

- Nodes of a graph variable
- Neighbors of one node of an undirected graph variable
- Successors of one node of a directed graph variable
- Predecessors of one node of a directed graph variable

An array of set variables can be associated with:

- Neighbors of every node of an undirected graph variable
- Successors of every node of a directed graph variable
- Predecessors of every node of a directed graph variable

3.3.2 Boolean channeling

A boolean variable can be associated with:

- a node of a graph variable
- An edge of an undirected graph variable
- An arc of one node of a directed graph variable

An array of boolean variables can be associated with:

- Nodes of a graph variable
- Neighbors of a node of an undirected graph variable
- Successors of a node of a directed graph variable
- Predecessors of a node of a directed graph variable

A matrix of boolean variables can be associated with:

- The adjacency matrix of a graph variables

3.3.3 Integer channeling

An array of integer variables can be associated with:

- Successors of a directed graph variable for which each node belongs to the solution and has exactly one successor

3.4 Implementing your own constraint

In Choco-3, a constraint is nothing else but a String name and a set of propagators (filtering algorithm objects). Therefore, to implement your own constraint, you need to create your own propagators. Let see an example with a simple constraint enforcing that a given directed graph should be antisymmetric, i.e. if an arc (i, i) belong to the solution, then the arc (j, i) is forbiden. This constraint can be created with the following line of code, where `PropAntiSymmetric` is a propagator:

```
return new Constraint("antisymmetric", new PropAntiSymmetric(g));
```

Let us now investigate how to implement such a propagator. There are basically two ways : either use a non-incremental or an incremental propagator.

3.4.1 A simple and non-incremental propagators

The simplest is the non-incremental approach, but it is also the slower. As we can see, every time the constraint is propagated, we perform an iteration over every mandatory arc (to remove its opposite if it has not been already done).

```
public class PropAntiSymmetric_coarse extends Propagator<IDirectedGraphVar> {

    //***********************************************************************************
    // VARIABLES
    //***********************************************************************************

    IDirectedGraphVar g;
    int n;

    //***********************************************************************************
    // CONSTRUCTORS
    //***********************************************************************************

    public PropAntiSymmetric_coarse(IDirectedGraphVar graph) {
        super(graph);
        g = graph;
        n = g.getNbMaxNodes();
    }

    //***********************************************************************************
    // METHODS
    //***********************************************************************************

    @Override
    public void propagate(int evtmask) throws ContradictionException {
        ISet ker = g.getMandatoryNodes();
        ISet succ;
        // iterates over mandatory nodes
        for (int i = ker.getFirstElement(); i >= 0; i = ker.getNextElement()) {
            succ = g.getMandSuccOf(i);
            // iterates over mandatory arcs
            for (int j = succ.getFirstElement(); j >= 0; j = succ.getNextElement()) {
                g.removeArc(j, i, aCause); // removes symmetric arcs
            }
        }
    }

    @Override // checker of partial instantiations, useful for reification
    public ESat isEntailed() {
        ISet ker = g.getMandatoryNodes();
        ISet succ;
        for (int i = ker.getFirstElement(); i >= 0; i = ker.getNextElement()) {
            succ = g.getMandSuccOf(i);
            for (int j = succ.getFirstElement(); j >= 0; j = succ.getNextElement()) {
                if (g.getMandSuccOf(j).contain(i)) {
                    return ESat.FALSE; // the constraint is violated
                }
            }
        }
        if (g.isInstantiated()) {
            return ESat.TRUE; // the constraint is satisfied for sure
        }
        return ESat.UNDEFINED; // satisfiability is undefined
    }
}
```

In order to improve performances, you can inform the propagation engine that the propagator should be called only after one or many arc enforcing. It is useless to propagate it after a set of arc removals or node modifications. To do so, you can override

the `getPropagationConditions` method as follows:

```
@Override
public int getPropagationConditions(int vIdx) {
    // propagation condition (facultative) : only propagate arc enforcing events
    return GraphEventType.ADD_ARC.getMask();
}
```

3.4.2 Incremental propagators

In an incremental approach, then we can run in constant time per newly enforce arc, with the following implementation:

```
public class PropAntiSymmetric extends Propagator<IDirectedGraphVar> {

    //***********************************************************************************
    // VARIABLES
    //***********************************************************************************

    IDirectedGraphVar g;
    IGraphDeltaMonitor gdm; // object enabling to iterate over enforced/removed nodes/arcs
    EnfProc enf; // procedure to apply to every enforced arc
    int n;

    //***********************************************************************************
    // CONSTRUCTORS
    //***********************************************************************************

    public PropAntiSymmetric(IDirectedGraphVar graph) {
        super(new IDirectedGraphVar[]{graph}, PropagatorPriority.UNARY, true);
        g = graph;
        gdm = g.monitorDelta(this);
        enf = new EnfProc();
        n = g.getNbMaxNodes();
    }

    //***********************************************************************************
    // METHODS
    //***********************************************************************************

    @Override
    public void propagate(int evtmask) throws ContradictionException {
    // First propagation (not incremental)
        ISet ker = g.getMandatoryNodes();
        ISet succ;
        for (int i = ker.getFirstElement(); i >= 0; i = ker.getNextElement()) {
            succ = g.getMandSuccOf(i);
            for (int j = succ.getFirstElement(); j >= 0; j = succ.getNextElement()) {
                g.removeArc(j, i, aCause);
            }
        }
        gdm.unfreeze(); // necessary call to setup incremental data-structures
    }

    @Override
    public void propagate(int idxVarInProp, int mask) throws ContradictionException {
    // incremental propagation over every enforced arc since the last call
        gdm.freeze();
        gdm.forEachArc(enf, GraphEventType.ADD_ARC);
        gdm.unfreeze();
    }

    @Override
    public int getPropagationConditions(int vIdx) {
        return GraphEventType.ADD_ARC.getMask();
```

```
    }

    @Override
    public ESat isEntailed() {
        ISet ker = g.getMandatoryNodes();
        ISet succ;
        for (int i = ker.getFirstElement(); i >= 0; i = ker.getNextElement()) {
            succ = g.getMandSuccOf(i);
            for (int j = succ.getFirstElement(); j >= 0; j = succ.getNextElement()) {
                if (g.getMandSuccOf(j).contain(i)) {
                    return ESat.FALSE;
                }
            }
        }
        if (g.isInstantiated()) {
            return ESat.TRUE;
        }
        return ESat.UNDEFINED;
    }

    //***********************************************************************************
    // PROCEDURES
    //***********************************************************************************

    /**
     * Enable to remove the opposite arc
     */
    private class EnfProc implements PairProcedure {
        @Override
        public void execute(int from, int to) throws ContradictionException {
            if (from != to) {
                g.removeArc(to, from, aCause); // whenever (from,to) is enforced, (to,from) is removed
            }
        }
    }
}
```

Note that the super constructor is no longer the same : `super(new IDirectedGraphVar[]graph, PropagatorPriority.UNARY, true);`. The first argument is the array of variables this propagator involves. The second one is an indicator of the runtime of the algorithm (here constant time). Finally, the last boolean argument states whether of not this propagator should be incremental or not. Therefore, it should be set to `true`.

4 Search

4.1 Variable selection

Search procedures are necessary to explore a search space when (and it is usual case) propagation is not sufficient to find a solution. Therefore, at each node of a search tree, whenever propagation has terminated, a search procedure must compute a new decision (refutable hypothesis), which creates a new search node, in order to continue the solving process. A decision consists of selecting a variable and restricting its domain (e.g. $X = 3$ or $X < 3$).

In case the model includes one or many graph variables, then a search process must select one variable and change its domain. For that, the user need to create a search procedure for each variable type and then create a composite search procedure which will decide, at each node, which one to apply (i.e. decide which variable type the solver should branch on). Note that in case your model has many graph variables, you should create one search strategy per such variable (or make your own search strategy), because built-in strategies only consider one graph variable. Here is a simple example which consists in applying successively `intSearch`, then `setSearch` and finally `graphSearch`:

```
final AbstractStrategy<IntVar> intSearch = IntStrategyFactory.minDom_LB(card);
final AbstractStrategy<SetVar> setSearch = SetStrategyFactory.force_first(vertices);
final AbstractStrategy<IUndirectedGraphVar> graphSearch = GraphStrategyFactory.graphLexico(graphvar);
solver.set(intSearch,setSearch,graphSearch); // this implicitly use a sequencer composite strategy
```

If you want to decide yourself which strategy to apply, you can build your own composite strategy as in the following example which performs a random selection:

```
AbstractStrategy<Variable> randomSelector = new AbstractStrategy(new Variable[]{vertices,card,graphvar}) {
    Random rd;
    AbstractStrategy[] strats;
    ArrayList<Decision> choices;
    @Override
    public void init() throws ContradictionException {
        rd = new Random();
        strats = new AbstractStrategy[]{intSearch,setSearch,graphSearch};
        choices = new ArrayList<>();
        for(AbstractStrategy s:strats){
            s.init();
        }
    }
    @Override
    public Decision getDecision() {
        choices.clear();
        for(AbstractStrategy s:strats){
            Decision d = s.getDecision();
            if (d!=null){
                choices.add(d);
            }
        }
        if(choices.isEmpty()){
            return null; // all variables are instantiated
        }else{
            return choices.get(rd.nextInt(choices.size()));
        }
    }
};
solver.set(randomSelector);
```

4.2 Branching on a graph variable

Let us now investigate how to modify the domain of a graph variable in a search decision. The `getDecision()` method of `AbstractStrategy<IGraphVar>` should return a `GraphDecision` object. Let call dec this decision object. There are basically four options:

- Make a potential (but not mandatory) vertex $node$ become mandatory
  (e.g. `dec.setNode(g, node, GraphAssignment.graph_enforcer);`)
- Remove a potential (but not mandatory) vertex $node$
  (e.g. `dec.setNode(g, node, GraphAssignment.graph_remover);`)
- Make a potential (but not mandatory) edge/arc $(from, to)$ become mandatory
  (e.g. `dec.setArc(g, from, to, GraphAssignment.graph_enforcer);`)
- Remove a potential (but not mandatory) edge/arc $(from, to)$
  (e.g. `dec.setArc(g, from, to, GraphAssignment.graph_remover);`)

You can implement your own `AbstractStrategy<IGraphVar>` or use build-in strategies that you can find in `GraphStrategyFactory`. Note that only `GraphAssignment.graph_enforce` is used by default.

- `GraphStrategyFactory.lexico` Selects nodes then edges according to their lexicographic ordering.
- `GraphStrategyFactory.random` Selects nodes randomly and then edges randomly.

You can also use the generic method:

```
/**
 * Dedicated graph branching strategy.
 *
 * @param GRAPHVAR   a graph variable to branch on
 * @param NODE_STRAT strategy over nodes
 * @param ARC_STRAT  strategy over arcs/edges
 * @param PRIORITY   enables to mention if it should first branch on nodes
 * @param <G>        either directed or undirected graph variable
 * @return a dedicated strategy to instantiate GRAPHVAR
 */
public static <G extends IGraphVar> AbstractStrategy graphStrategy(G GRAPHVAR,
                                                                   NodeStrategy NODE_STRAT,
                                                                   ArcStrategy ARC_STRAT,
                                                                   GraphStrategy.NodeArcPriority PRIORITY) {
    return new GraphStrategy(GRAPHVAR, NODE_STRAT, ARC_STRAT, PRIORITY);
}
```

You can then implement your own `NodeStrategy`, which should select the next node to branch on, as the following which returns the first unfixed vertex or -1 if none exists:

```
public class LexNode extends NodeStrategy<IGraphVar> {

    public LexNode(IGraphVar g) {
        super(g);
    }
```

```
    @Override
    public int nextNode() {
        for (int i = envNodes.getFirstElement(); i >= 0; i = envNodes.getNextElement()) {
            if (!kerNodes.contain(i)) {
                return i;
            }
        }
        return -1;
    }
}
```

You can also implement your own `ArcStrategy`, which should select the next arc to branch on, as the following which selects the first unfixed arc and returns true or false depending of whether such an arc exists or not. Note that this time the arc is defined through instance variables called $from$ and to.

```
public class LexArc extends ArcStrategy<IGraphVar> {

    public LexArc(IGraphVar g) {
        super(g);
    }

    @Override
    public boolean computeNextArc() {
        ISet envSuc, kerSuc;
        for (int i = envNodes.getFirstElement(); i >= 0; i = envNodes.getNextElement()) {
            envSuc = g.getPotSuccOrNeighOf(i);
            kerSuc = g.getMandSuccOrNeighOf(i);
            if (envSuc.getSize() != kerSuc.getSize()) {
                for (int j = envSuc.getFirstElement(); j >= 0; j = envSuc.getNextElement()) {
                    if (!kerSuc.contain(j)) {
                        this.from = i;
                        this.to = j;
                        return true;
                    }
                }
            }
        }
        this.from = this.to = -1;
        return false;
    }
}
```

There are two options for `NodeArcPriority`:

- `NodeArcPriority.NODES_THEN_ARCS`: First fixes every node and then fixes every arc
- `NodeArcPriority.ARCS`: Fixes every arc (forcing an arc automatically forces its incident nodes). Note that potential nodes with no incident arcs may remain unfixed.

4.3 Large Neighborhood Search

Large Neighborhood Search (LNS) is most powerful technique to solve large scale optimization problems. It may not be able to prove optimality but is designed to provide very good solutions in a reasonable runtime. Another interesting motivation for setting up an LNS is that the output solution may not be easy to improve by hand.

The sample `TSP_lns.java` provides an illustration of how to implement a large neighborhood search to solve a tsp. Setting up an LNS requires can be done as follows:

```
// object describing which edges to freeze
SubpathLNS LNS = new SubpathLNS(graph.getNbMaxNodes(),solver);
// restarts every 30 fails (facultative)
LNS.fastRestart(new FailCounter(30));
// set up the lns (the last argument indicates whether to restart on every solution)
solver.plugMonitor(new LargeNeighborhoodSearch(solver,LNS,false))
```

The only smart part relies in the implementation of `INeighbor`. We provide the following implementation

```
/**
 * Object describing which edges to freeze and which others to relax in the LNS
 * Relaxes a (sub)path of the previous solution (freezes the rest)
 */
private class SubpathLNS extends ANeighbor{

    Random rd = new Random(0);
    int n, nbRL; // number of nodes, counter
    UndirectedGraph solution; // object to  store the current best solution
    int nbFreeEdges = 15; // number of edges which should not be frozen

    protected SubpathLNS(int n, Solver mSolver) {
        super(mSolver);
        this.n = n;
        this.solution = new UndirectedGraph(n,SetType.LINKED_LIST,true);
    }

    @Override
    public void recordSolution() {
        // stores a solution in a graph object
        for(int i=0;i<n;i++)solution.getNeighOf(i).clear();
        for(int i=0;i<n;i++){
            ISet nei = graph.getMandNeighOf(i);
            for(int j=nei.getFirstElement();j>=0;j=nei.getNextElement()){
                solution.addEdge(i,j);
            }
        }
    }

    @Override
    public void fixSomeVariables(ICause cause) throws ContradictionException {
        // relaxes a sub-path (a set of consecutive edges in a solution)
        int i1 = rd.nextInt(n);
        ISet nei = solution.getNeighOf(i1);
        int i2 = nei.getFirstElement();
        if(rd.nextBoolean()){
            i2 = nei.getNextElement();
        }
        for(int k=0;k<n-nbFreeEdges;k++){
            graph.enforceArc(i1,i2,cause);
            int i3 = solution.getNeighOf(i2).getFirstElement();
            if(i3==i1){
                i3 = solution.getNeighOf(i2).getNextElement();
            }
            i1 = i2;
            i2 = i3;
        }
    }

    @Override
    public void restrictLess() {
        nbRL++;
        // Eventually increases the size of the relaxes fragment (not necessary)
        if(nbRL>nbFreeEdges){
```

```
            nbRL = 0;
            nbFreeEdges += (nbFreeEdges*3)/2;
        }
    }

    @Override
    public boolean isSearchComplete() {
        return nbFreeEdges>=n;
    }
}
```

The main methods to implement are `recordSolution()` which requires to copy some part of the current solution in internal data structure in order to be able to freeze some edges later. The method `fixSomeVariables` is called after a restart in order to freeze some variables (here some edges). The method `restrictLess` does not need to be implemented. It is called some times to suggest considering a larger neighborhood (variable neighborhood search). Finally, `isSearchComplete` asks whether the search is complete or not. The answer should be `false` in the general case, unless no variable has been frozen (which may occur in variable large neighborhood search).

5 Practical examples

5.1 Large scale Hamiltonian cycle : The Knight's Tour Problem

The Knight's Tour Problem (KTP) is defined over a chessboard and consists of making a chess knight visit every cell exactly once and reach back its original position, where possible moves are given by classical chess rules for the knight. This problem can be seen as a graph problem. Let us introduce an undirected graph for which every vertex is associated with a cell of the chessboard and there is an edge between two vertices if and only if the chess rules allow to travel between the two cells associated with the edge endpoints. The problem then consists of finding a Hamiltonian cycle in this graph. This can be addressed using `Choco-Graph`. Here is the graph-based CP model of this KTP (with a board length of 200, which involves a $40,000$-vertex graph) :

```
public class KnightTourProblem extends AbstractProblem {

    //***********************************************************************************
    // VARIABLES
    //***********************************************************************************

    @Option(name = "-tl", usage = "time limit.", required = false)
    private long limit = 20000; // 20 sec time limit
    @Option(name = "-bl", usage = "Board length.", required = false)
    private int boardLength = 200; // length = 200 => 40000 vertices

    private IUndirectedGraphVar graph;

    //***********************************************************************************
    // METHODS
    //***********************************************************************************

    public static void main(String[] args) {
        new KnightTourProblem().execute(args);
    }

    @Override
    public void createSolver() {
        level = Level.SILENT; // do not print the solution value (too big!)
        solver = new Solver("solving the knight's tour problem with graph variables");
    }

    @Override
    public void buildModel() {
        // This generates the boolean incidence matrix of the chessboard graph
        // It is responsible of the high memory consumption of this example
        // and could be replaced by lighter data structure (but it is simpler as it is)
        boolean[][] matrix = HCP_Utils.generateKingTourInstance(boardLength);
        // variables
        SetFactory.RECYCLE = false; // (optimization for large instances involving few backtracks, not very import
        int n = matrix.length;
        // graph representing mandatory nodes and edges
        // (linked list data structure as the expected solution is expected to be sparse,
        // every vertex in [0,n-1] is mandatory)
        UndirectedGraph GLB = new UndirectedGraph(solver,n,SetType.LINKED_LIST,true);
        // graph representing potential nodes and edges
        // (linked list data structure as its initial value is sparse,
        // every vertex in [0,n-1] belongs to the potential)
        UndirectedGraph GUB = new UndirectedGraph(solver,n,SetType.LINKED_LIST,true);
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (matrix[i][j]) {
                    GUB.addEdge(i, j); // add possible edges to the domain
                }
```

```
            }
        }
        // build the graph variable
        graph = GraphVarFactory.undirected_graph_var("G", GLB, GUB, solver);
        // constraints (hamiltonian cycle)
        solver.post(GraphConstraintFactory.hamiltonianCycle(graph));
    }

    @Override
    public void configureSearch() {
        // basically branch on sparse areas of the graph
        solver.set(GraphStrategyFactory.graphStrategy(graph,     // variable to branch on
                                            null,       // no need node selection heuristic (all are mandatory)
                                            new MinNeigh(graph), // arc selection heuristic
                                            GraphStrategy.NodeArcPriority.ARCS) // branch on arcs only
                                            );
        SearchMonitorFactory.limitTime(solver, limit);
        SearchMonitorFactory.log(solver,false,false);
    }

    @Override
    public void solve() {
        solver.findSolution();
    }

    @Override
    public void prettyOut() {}

    //***************************************************************************
    // HEURISTICS
    //***************************************************************************

    private static class MinNeigh extends ArcStrategy<IUndirectedGraphVar> {
        int n;

        public MinNeigh(IUndirectedGraphVar graphVar) {
            super(graphVar);
            n = graphVar.getNbMaxNodes();
        }

        @Override
        public boolean computeNextArc() {
            ISet suc;
            int size = n + 1;
            int sizi;
            from = -1;
            // find the lowest remaining degree vertex
            for (int i = 0; i < n; i++) {
                sizi = g.getPotNeighOf(i).getSize() - g.getMandNeighOf(i).getSize();
                if (sizi < size && sizi > 0) {
                    from = i;
                    size = sizi;
                }
            }
            if (from == -1) {
                return false;
            }
            // find its lowest remaining degree neighbor
            suc = g.getPotNeighOf(from);
            for (int j = suc.getFirstElement(); j >= 0; j = suc.getNextElement()) {
                if (!g.getMandNeighOf(from).contain(j)) {
                    to = j;
                    return true;
                }
            }
            throw new UnsupportedOperationException("this should not happen!");
        }
    }
```

```
}
```

The Hamiltonian cycle constraint involves basic filtering but which run incrementally in constant time for each edge removal/enforcing. Note that having a undirected model is a key (a directed representation would bring symmetries and increase the search space). This model provides the following output (obtained on a usual laptop):

```
** Choco 3.2.1-SNAPSHOT (2014-05) : Constraint Programming Solver, Copyleft (c) 2010-2014
** Solve : solving the knight's tour problem with a graph variable
- Search statistics
    Solutions: 1
    Building time : 2,796s      // time to build the graph variable domain and propagators
    Initialisation : 0,007s
    Initial propagation : 0,039s    // initial propagation runtime
    Resolution : 7,918s         // total solving time
    Nodes: 39 507               // number of branching node is almost the number of nodes in the graph (40,000)
    Backtracks: 1               // good filtering and good search! This model is good on the Hamiltonian cycle prob
    Fails: 1
    Restarts: 0
    Max depth: 39 506
    Propagations: 103 963 + 0   // number of incremental propagations + number of non-incremental propagations
    Memory: -20mb           // memory usage of the model (while the input matrix takes >1gb)
    Variables: 3                // graph + default solver constants (ZERO and ONE)
    Constraints: 1              // Hamiltonian cycle constraint (which has 3 incremental propagators)
```

If we take a small instance, with a board length of 8, whence 64 vertices. Here is the print of the graph variable initial domain :

```
graph_var G
upper bound: // (potential nodes and edges)
nodes :
[0,63]
neighbors :
0 -> {17 10 }
1 -> {18 16 11 }
2 -> {19 17 12 8 }
3 -> {20 18 13 9 }
4 -> {21 19 14 10 }
5 -> {22 20 15 11 }
6 -> {23 21 12 }
7 -> {22 13 }
8 -> {25 18 2 }
9 -> {26 24 19 3 }
10 -> {27 25 20 16 4 0 }
11 -> {28 26 21 17 5 1 }
12 -> {29 27 22 18 6 2 }
13 -> {30 28 23 19 7 3 }
14 -> {31 29 20 4 }
15 -> {30 21 5 }
16 -> {33 26 10 1 }
17 -> {34 32 27 11 2 0 }
18 -> {35 33 28 24 12 8 3 1 }
19 -> {36 34 29 25 13 9 4 2 }
20 -> {37 35 30 26 14 10 5 3 }
21 -> {38 36 31 27 15 11 6 4 }
22 -> {39 37 28 12 7 5 }
23 -> {38 29 13 6 }
24 -> {41 34 18 9 }
25 -> {42 40 35 19 10 8 }
26 -> {43 41 36 32 20 16 11 9 }
27 -> {44 42 37 33 21 17 12 10 }
28 -> {45 43 38 34 22 18 13 11 }
29 -> {46 44 39 35 23 19 14 12 }
30 -> {47 45 36 20 15 13 }
```

```
31 -> {46 37 21 14 }
32 -> {49 42 26 17 }
33 -> {50 48 43 27 18 16 }
34 -> {51 49 44 40 28 24 19 17 }
35 -> {52 50 45 41 29 25 20 18 }
36 -> {53 51 46 42 30 26 21 19 }
37 -> {54 52 47 43 31 27 22 20 }
38 -> {55 53 44 28 23 21 }
39 -> {54 45 29 22 }
40 -> {57 50 34 25 }
41 -> {58 56 51 35 26 24 }
42 -> {59 57 52 48 36 32 27 25 }
43 -> {60 58 53 49 37 33 28 26 }
44 -> {61 59 54 50 38 34 29 27 }
45 -> {62 60 55 51 39 35 30 28 }
46 -> {63 61 52 36 31 29 }
47 -> {62 53 37 30 }
48 -> {58 42 33 }
49 -> {59 43 34 32 }
50 -> {60 56 44 40 35 33 }
51 -> {61 57 45 41 36 34 }
52 -> {62 58 46 42 37 35 }
53 -> {63 59 47 43 38 36 }
54 -> {60 44 39 37 }
55 -> {61 45 38 }
56 -> {50 41 }
57 -> {51 42 40 }
58 -> {52 48 43 41 }
59 -> {53 49 44 42 }
60 -> {54 50 45 43 }
61 -> {55 51 46 44 }
62 -> {52 47 45 }
63 -> {53 46 }

lower bound:     // (mandatory nodes and edges)
nodes :
[0,63]       // all vertices are mandatory
neighbors : // no edge is mandatory
0 -> {}
1 -> {}
2 -> {}
3 -> {}
4 -> {}
5 -> {}
6 -> {}
7 -> {}
8 -> {}
9 -> {}
10 -> {}
11 -> {}
12 -> {}
13 -> {}
14 -> {}
15 -> {}
16 -> {}
17 -> {}
18 -> {}
19 -> {}
20 -> {}
21 -> {}
22 -> {}
23 -> {}
24 -> {}
25 -> {}
26 -> {}
27 -> {}
28 -> {}
29 -> {}
```

```
30 -> {}
31 -> {}
32 -> {}
33 -> {}
34 -> {}
35 -> {}
36 -> {}
37 -> {}
38 -> {}
39 -> {}
40 -> {}
41 -> {}
42 -> {}
43 -> {}
44 -> {}
45 -> {}
46 -> {}
47 -> {}
48 -> {}
49 -> {}
50 -> {}
51 -> {}
52 -> {}
53 -> {}
54 -> {}
55 -> {}
56 -> {}
57 -> {}
58 -> {}
59 -> {}
60 -> {}
61 -> {}
62 -> {}
63 -> {}
```

After solving the KTP, printing the (value of) the graph variable gives:

```
graph_var G
value:       // the variable is instantiated
nodes :
[0,63]       // All vertices belong to the solution graph
neighbors :
0 -> {17 10 }    // The neighbors of node 0 are nodes 17 and 10
1 -> {18 16 }    //... edges (1,18) and (1,16) belong to the solution graph
2 -> {19 17 }    // ...
3 -> {20 13 }
4 -> {19 14 }
5 -> {22 11 }
6 -> {23 21 }
7 -> {22 13 }
8 -> {25 18 }
9 -> {26 24 }
10 -> {27 0 }
11 -> {28 5 }
12 -> {29 27 }
13 -> {7 3 }
14 -> {31 4 }
15 -> {30 21 }
16 -> {33 1 }
17 -> {2 0 }
18 -> {8 1 }
19 -> {4 2 }
20 -> {35 3 }
21 -> {15 6 }
22 -> {7 5 }
23 -> {38 6 }
24 -> {41 9 }
25 -> {42 8 }
```

```
26 -> {36 9 }
27 -> {12 10 }
28 -> {34 11 }
29 -> {44 12 }
30 -> {36 15 }
31 -> {46 14 }
32 -> {49 42 }
33 -> {48 16 }
34 -> {51 28 }
35 -> {45 20 }
36 -> {26 30 }
37 -> {43 47 }
38 -> {55 23 }
39 -> {54 45 }
40 -> {57 50 }
41 -> {56 24 }
42 -> {32 25 }
43 -> {37 60 }
44 -> {61 29 }
45 -> {35 39 }
46 -> {63 31 }
47 -> {62 37 }
48 -> {58 33 }
49 -> {59 32 }
50 -> {56 40 }
51 -> {57 34 }
52 -> {62 58 }
53 -> {63 59 }
54 -> {60 39 }
55 -> {61 38 }
56 -> {50 41 }
57 -> {51 40 }
58 -> {52 48 }
59 -> {53 49 }
60 -> {54 43 }
61 -> {55 44 }
62 -> {52 47 }
63 -> {53 46 }
```

5.2 Solving the Traveling Salesman Problem

The sample `TSP_CP_Solver` applies 30 seconds of LNS and 30 seconds of classical DFS of the CP model for solving the TSP. The CP model for the TSP is the following:

```
// variables
totalCost = VariableFactory.bounded("obj", 0, 99999999, solver);
// creates a graph containing n nodes
UndirectedGraph GLB = new UndirectedGraph(solver, n, SetType.LINKED_LIST, true);
UndirectedGraph GUB = new UndirectedGraph(solver, n, SetType.SWAP_ARRAY, true);
// adds potential edges
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        GUB.addEdge(i, j);
    }
}
graph = GraphVarFactory.undirected_graph_var("G", GLB, GUB, solver);

// constraints : TSP basic model + lagrangian relaxation (after a first solution has been found)
solver.post(GraphConstraintFactory.tsp(graph, totalCost, costMatrix, 2));
```

Note that in the exact (DFS) model, the Lagrangian relaxation is triggered since root node (mode = 1). The search procedure varies from one model to the other (see

the sample files). The output on the `bier127` instance, involving a complete graph of 127 vertices, is the following:

```
** Choco 3.2.1-SNAPSHOT (2014-05) : Constraint Programming Solver, Copyleft (c) 2010-2014
** Solve : TSP_lns
solution found : obj = 141339
solution found : obj = 138636
solution found : obj = 137046
solution found : obj = 136912
solution found : obj = 136263
...
solution found : obj = 118528
solution found : obj = 118502
solution found : obj = 118442
solution found : obj = 118386
solution found : obj = 118374
solution found : obj = 118326
solution found : obj = 118282
- Search statistics
    Solutions: 165
    Minimize obj = 118282,
    Building time : 0,123s
    Initialisation : 0,007s
    Initial propagation : 0,333s
    Resolution : 30,014s
    Nodes: 2 348
    Backtracks: 2 960
    Fails: 1 498
    Restarts: 137
    Max depth: 126
    Propagations: 19 981 + 0
    Memory: -6mb
    Variables: 4
    Constraints: 1
Best solution found : 118282 (but no optimality proof)
** Choco 3.2.1-SNAPSHOT (2014-05) : Constraint Programming Solver, Copyleft (c) 2010-2014
** Solve : TSP_exact
solution found : obj = 118282
- Search statistics
    Solutions: 1
    Minimize obj = 118282,
    Building time : 0,003s
    Initialisation : 0,000s
    Initial propagation : 0,039s
    Resolution : 1,077s
    Nodes: 129
    Backtracks: 257
    Fails: 128
    Restarts: 0
    Max depth: 23
    Propagations: 1 535 + 0
    Memory: -1mb
    Variables: 4
    Constraints: 1
Optimality proved with exact CP approach
```

As you can see, the LNS is a powerful tool to find very good (sometimes optimal) solutions, and we can then use a classical DFS approach to perform the optimality proof.

5.3 Finding a Directed Acyclic (sub)Graph

We now consider the problem of finding a DAG comprised between two input graphs (initial domain) and such that the number of arcs is maximal. This problem can be

stated through the following program:

```
public class DAGProblem extends AbstractProblem{

    IDirectedGraphVar dag;
    IntVar nbArcs;

    @Override
    public void createSolver() {
        solver = new Solver("DAG sample");
    }

    @Override
    public void buildModel() {
        // input graph
        int n = 5;

        // VARIABLE COUNTING THE NUMBER OF ARCS
        nbArcs = VF.bounded("arcCount", 0, n * n, solver);
        // GRAPH VARIABLE : initial domain (every node belongs to the solution)
        DirectedGraph GLB = new DirectedGraph(solver, n, SetType.BITSET, true);
        DirectedGraph GUB = new DirectedGraph(solver, n, SetType.BITSET, true);
        GLB.addArc(0,1); // some arbitrary mandatory arcs
        GLB.addArc(1,2);
        GLB.addArc(3,1);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                GUB.addArc(i, j);        // potential edge
            }
        }
        dag = GraphVarFactory.directedGraph("dag", GLB, GUB, solver);

        // CONSTRAINTS
        solver.post(GraphConstraintFactory.no_circuit(dag));
        solver.post(GraphConstraintFactory.nb_arcs(dag,nbArcs));
    }

    @Override
    public void configureSearch() {
        // tries to find the largest graph first
        solver.set(GraphStrategyFactory.lexico(dag));
    }

    @Override
    public void solve() {
        solver.findOptimalSolution(ResolutionPolicy.MAXIMIZE,nbArcs);
    }

    @Override
    public void prettyOut() {}

    public static void main(String[] args){
        new DAGProblem().execute(args);
    }
}
```

This provides the following output:

```
** Choco 3.2.1-SNAPSHOT (2014-05) : Constraint Programming Solver, Copyleft (c) 2010-2014
** Solve : DAG sample
- Solution #1 found. 1 Solutions, Maximize arcCount = 10, Resolution 0,007s, 8 Nodes, 0 Backtracks, 0 Fails, 0 Res
    graph_var dag
value:
nodes :
[0,4]
successors :
0 -> {1 2 3 4 }
1 -> {2 4 }
```

```
2 -> {4 }
3 -> {1 2 4 }
4 -> {}
  .
- Search statistics
    Solutions: 1
    Maximize arcCount = 10,
    Building time : 0,058s
    Initialisation : 0,004s
    Initial propagation : 0,005s
    Resolution : 0,013s
    Nodes: 23
    Backtracks: 45
    Fails: 22
    Restarts: 0
    Max depth: 7
    Propagations: 31 + 0
    Memory: 0mb
    Variables: 4
    Constraints: 2
```

6 How to contribute?

You can contribute to this project in vert different ways. The first simple thing you can do is to just use it and send us some feedback about your experience. We are interested in understanding what can be improved in the module and the kind of applications you are doing with it. Feel also free to advertise around you.

You can also contribute to choco-graph with your work by sending us some code (models, tests, constraints...) in order to improve the library. We have a GitHub repository at `https://github.com/chocoteam`.

You can also fund us directly so that we develop some features you need for your application or research activities.

Last but not least, you can contact us for partnerships on industrial or academic projects.