SCIL – Symbolic Constraints in Integer Linear Programming *

Ernst Althaus, Alexander Bockmayr, Matthias Elf
Thomas Kasper, Michael Jünger, Kurt Mehlhorn

April 15, 2002

Abstract

We describe SCIL. SCIL introduces symbolic constraints into branch-and-cut-and-price algorithms for integer linear programs. Symbolic constraints are known from constraint programming and contribute significantly to the expressive power, ease of use, and efficiency of constraint programs.

1 Introduction

Many combinatorial optimization problems are naturally formulated through constraints. Consider the *traveling salesman problem* (TSP). It asks for the minimum cost Hamiltonian cycle[1] in an undirected graph $G = (V, E)$ with edge weights $(w_e)_{e \in E}$. Formulated as an optimization task:

Find a subset T of the edges of G such that "T is a Hamiltonian cycle" and $\sum_{e \in T} w_e$ is minimum.

Our vision is that the sentence above (written in some suitable language) suffices to obtain an efficient algorithm for the TSP. Efficiency is meant in a double sense. We want short development time (= efficient use of human resources) and runtime efficiency (= efficient use of computer resources). The software system SCIL is our first step towards realizing this vision. Section 3 shows a SCIL program for the traveling salesman problem.

We propose a programming system for *integer linear programming* (ILP) and *mixed integer linear programming* (MILP) based on a *branch-and-cut-and-price* framework (BCP) that features *symbolic constraints*. Integer linear programming and more specifically the branch-and-cut-and-price paradigm for solving ILPs is one of the most effective approaches to find exact or provably good solutions of hard combinatorial optimization problems [NW88, EGJR01]. It has been used for a wide range of problems including the TSP [ABCC99, Nad02], maximum-cut-problems [SDJ$^+$96], cutting-stock-problems [VBJN94], or crew-scheduling [BJN$^+$98]. For more applications we refer to the overview [CF97].

The implementation of a BCP algorithm requires significant expert knowledge, a fact that has hindered the spread of the method outside the scientific world. It consists of various involved components, each with wide influence on the algorithm's overall performance. Almost all parts of a BCP algorithm considerably rely on *linear programming* (LP) methods or properties of the linear programming relaxation of the underlying ILP. Many components are problem independent and can be provided by existing software packages (see [JT00]). But there is still a major problem dependent part: an ILP formulation has to be found, appropriate linear programs have to be derived, the various methods for exploiting LP solutions to find feasible solutions or speedup the computation have to be designed and implemented. To our knowledge there is no BCP system for combinatorial optimization that covers the problem dependent part in an appropriate way.

SCIL closes this gap. It simplifies the implementation of BCP-algorithms by supporting high-level specifications of combinatorial optimization problems with linear objective functions. It provides a library of symbolic constraints and can convert high-level specifications into efficient BCP-algorithms. A user may extend SCIL by defining new symbolic constraints or by changing the standard behavior of the algorithms.

We have used SCIL already in several projects like curve reconstruction [AM01] and computing protein dockings [AKLM00]. A documentation of SCIL, a list of available symbolic constraints and more examples can be found on the SCIL home page (`http://www.mpi-sb.mpg.de/SCIL`).

The rest of the paper is organized as follows. We give a short introduction on BCP algorithms for combinatorial optimization in Section 1.1, we relate SCIL to extant work in Section 2, give a first SCIL-program in Section 3, formalize the BCP-paradigm and define the role of symbolic constraints in Section 4,

*Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Die Arbeit wurde mit der Unterstützung eines Stipendiums im Rahmen des Hochschulsonderprogramms III von Bund und Ländern über den DAAD ermöglicht.

[1] A Hamiltonian cycle ("tour") in a graph is a cycle passing exactly once through every node of the graph.

discuss the implementation of symbolic constraints in Section 5, the application of Lagrangean relaxation technique in Section 6, and numerical problems in Section 7.

1.1 ILP Formulations and BCP Algorithms

Again we use the TSP to explain how combinatorial optimization problems are modeled as ILPs and solved by BCP. First, one has to find an ILP formulation. For the TSP (as for many other problems) one starts from a first high-level description, e.g., the description we gave in the introduction. Then one tries to characterize the problem in more detail by identifying problem defining structures. This may yield another high-level description like (1b) which is suitable to be formulated as an ILP. The resulting well-known ILP formulation (*subtour elimination formulation*) is given in (1a).

$$
(P)\left|\begin{array}{llll}
\min & \sum_{e\in E} w_e x_e & & \\
\text{s.t.} & \sum_{j\in V} x_{ij} & = 2 & \text{for all } i \in V \\
 & \sum_{i,j\in S} x_{ij} & \leq |S|-1 & \text{for all } S \subsetneq V,\ 3 \leq |S| \leq \lfloor \frac{|V|}{2} \rfloor \\
 & 0 \leq x_e \leq 1 & & \text{for all } e \in E \\
 & x_e \text{ integral} & & \text{for all } e \in E
\end{array}\right|
\begin{array}{ll}
\min_{T\subseteq E} & \sum_{e\in T} w_e \\
\text{s.t.} & \text{each node in T has degree 2} \\
 & T \text{ contains no subtour}
\end{array}
\qquad (1)
$$

a) ILP formulation b) high-level description

In this formulation there is a connection between edges e and binary variables x_e. The value of x_e is 1 if $e \in T$ and 0 otherwise. Furthermore, nodes $i \in V$ correspond to *degree equations* and subsets S correspond to *subtour elimination constraints*. Such correspondences between combinatorial objects and components of the ILP formulation are inevitable and turn out to be very useful in the modeling environment SCIL. Beside the two classes of constraints used in (1a) there are many other classes of inequalities used in state of the art TSP solvers [ABCC99, Nad02].

In the following we sketch a BCP algorithm for the formulation (1a) which also applies to every other ILP problem. A BCP-algorithm for the minimization problem P is basically a branch-and-bound algorithm in which the bounds are solutions of LP-relaxations. These relaxations are naturally derived from the linear description of our problem by dropping the integrality constraints. However, our formulation contains an exponential number of constraints for an instance with n nodes (there are $\Theta(2^n)$ subsets). This leads to the idea to solve the relaxation with a subset of all constraints, check whether the optimum solution violates any other of the constraints, and if so, append them to the relaxation. Identifying violated constraints is called *separation* and added constraints are called *cutting planes*. The procedure is a *cutting plane algorithm*.

Every subproblem in the enumeration of the branch-and-bound algorithm has its own LP relaxation which is iteratively strengthened by the cutting plane procedure. This results in increasing *local lower bounds* on the optimum solution for a subproblem. Subproblems can be discarded if the local lower bound is not smaller than the objective function value g of the best feasible solution found so far. The value g is a *global upper bound* on the optimum solution. Branching, i.e., splitting a subproblem into two or more, is performed by introducing appropriate linear constraints to the resulting subproblems. The LP relaxation of a successor node in the enumeration-tree is created by adding appropriate "branching constraints" to the relaxation of the predecessor.

In some applications where the LP relaxations of the subproblems contain too many variables in order to solve the LPs efficiently the LP relaxations are solved by *column generation*. This method solves the LP on a small subset of the variables. All missing variables are assumed to have value 0. In order to prove that the solution of the restricted LP is optimal for the entire set of variables one has to check the reduced costs of the ignored variables. This step is called *pricing*.

2 Comparison to Extant Work

The cutting plane procedure was invented in 1954 by Dantzig et. al. [DFJ54] for the traveling salesman problem. The branch-and-cut-paradigm was first formulated for and successfully applied to the linear ordering problem by Grötschel et. al. in 1984 [GJR84]. The first state-of-the-art BCP algorithm was developed in 1991 by Padberg and Rinaldi [PR91]. Since then, the method was applied to a huge number of problems.

The LP-solver was treated as an independent black-box almost from the beginning. The software system ABACUS [JT00] is an object oriented software framework for BCP-algorithms. It provides the problem independent part, e.g., the enumeration tree, the interface with the LP-solver and various standard strategies, e.g. for branching or exploiting LP properties. It has concepts like subproblem, cut-generation, and pricing. It has no modeling language and does not provide a library of separation and pricing routines.

Another direction are general ILP-solvers using the BC-paradigm. These solvers are mostly extensions of existing linear programming solvers, e.g. CPLEX [CPL] or XPRESS [Das00]. They solve arbitrary ILPs using general cuts, e.g., Gomory cuts. Modeling languages, such as AMPL [FGK92] or GAMS [Cor02], can be used to specify the ILP. However, modeling is on the level of ILPs, all constraints must be given explicitly, and the support of problem specific cuts and variable generation is rudimentary. Modeling systems for general ILP solvers translate a high-level problem description into a matrix. During this process most of the structural knowledge contained in the model gets lost. The solver only sees a matrix.

Symbolic constraints are one of the main achievements of *constraint programming* (CP). A symbolic constraint in CP can be every subset of all assignments of values to variables. They were first introduced in the CP-system CHIP [BC94, ADH+87] and later used in further CP-systems like ILOG OPL Studio [ILO, vHPP00] and OZ [Smo96]. These systems have highly-developed modeling systems, which support high-level and intuitive problem formulations.

Alexander Bockmayr and Thomas Kasper [BK98, Kas98] suggested to extend ILP by symbolic constraints. SCIL realizes their suggestion. An optimization problem is specified by a set of variables (ranging over the rational numbers) a linear objective function and a set of constraints. A constraint may be a *linear constraint* (a linear inequality or equation) or a *symbolic constraint*. Declaratively, a symbolic constraint can be every subset of the set of all assignments of values to variables. Note that not all subsets of assignments are realizable in our framework. The system of linear constraints can be solved efficiently by an LP-solver whereas the symbolic constraints can make the problem hard.

Symbolic constraints in integer programming allow the modeler to include large families of linear constraints, variables, and branching rules into the model, without writing them down explicitly. The most important symbolic constraint is the integrality constraints. It is the only symbolic constraint which is available in all BCP-systems. In its simplest form the integrality constraint for a variable x includes only the branching rules $x \leq c \vee x \geq c+1$ for all integers c into the system. An other example, is the `tour` constraint mentioned earlier in the paper. It includes the degree and the subtour elimination constraints into the model. Declaratively, this constraint is equivalent to exponentially many linear inequalities. Operationally, however, only some of these inequalities will be added to the model at runtime (as cutting planes). Concerning efficiency, symbolic constraints allow for example to integrate specialized cutting plane algorithms based on polyhedral combinatorics into a general branch-and-cut solver. Symbolic constraints give the modeler the possibility to identify some specific structure in the problem, which later can be exploited when the model is solved. For example, when we solve a model containing the symbolic constraint `tour`, we can enhance our general branch-and-cut solver by computing specialized cutting planes for the traveling salesman problem instead of using more general cutting planes for arbitrary linear 0-1 programs.

All available systems for ILP have in common that the variables are only an indexed set. But as mentioned above we often have a correspondence between combinatorial objects and variables and constraints. This turns out to be a key to elegant modeling. SCIL takes this into account by introducing variable maps and constraint maps. See section 3 for details.

3 Specification of Optimization Problems

As our implementation is based on C++, it is natural that problems, subproblems, symbolic constraints, linear constraint and variables are C++ objects. Since variables and constraints are C++ objects, they can be easily created or extended by inheritance. They can be added to a specific problem by appropriate member functions.

As mentioned in section 1.1 many ILP formulations for combinatorial problems use structured collections of variables and linear constraints, e.g., we may have a variable for each edge of a graph and a linear constraint for each vertex. In SCIL we provide variable maps (similarly constraint maps) that allow one to establish a correspondence between variables and objects of an arbitrary type. A `var_map<T>` where `T` is an arbitrary C++ class stores a mapping between objects of type `T` and ILP variables. Read- and write-access to variable maps is via the subscript-operator.

The following SCIL-program solves the TSP-problem on a graph G with edge weights w. It returns the tour as a list of edges. SCIL imports combinatorial objects from LEDA [MN99, LED] and hence its syntax is remeniscient of LEDA, e.g., `edge_array<double>` is an array of double values indexed by the edges of a graph. Even for a non-experienced C++ programmer the following program should be clear.

```
void compute_tsp_tour(graph& G, edge_array<double>& w, list<edge>&tour)
{
  // define a minimization problem P
  ILP_problem P(optsense_min);

  // add a 0-1 variable for each edge e of G and make w[e] it coefficient in the
  // objective function. The var_map EV establishes the connection between edges
  // and variables
  var_map<edge> EV(nil);
  forall_edges(e,G) EV[e] = P.add_binary_variable(w[e]);

  // add the tour constraint for the edges of G to P
  P.add_constraint(new TOUR(G, EV));

  // solve P
  P.solve();

  // extract the solution
  forall_edges(e,G) if ( P.get_value(EV[e]) ==  1 ) tour.add(e);
}
```

The tour constraint is a built-in constraint of SCIL and makes the program above particularly short. The concept of variable maps makes the program very readable.

In many applications there are side constraints to a central symbolic constraint. Side constraints for the traveling salesman tour arise for example, if the salesman must respect a partial ordering of the cities, e.g., if he picks additional items from some cities for other cities.

In contrast to the previous problem, we use the directed version of the tour constraint to model the Hamiltonian cycle. The partial ordering can be modeled as follows. Let s be the start node of the tour and u and v be two nodes so that u must be visited before v. Even if one ignores all edges adjacent to s there must be a path between u and v. To model this we can use path constraints. The *path constraint* `path(G,u,v,EV)` enforces that the set of edges, whose corresponding variables have value one, describe a superset of a path between u and v. From polyhedral theory it is known that the path constraint is completely described by the following (exponential) class of inequalities:

$$\sum_{e \in \delta^+(S)} x_e \geq 1 \quad \forall S, \{u\} \subseteq S \subseteq V \setminus \{v\}.$$

Note that we have to use two different variable maps for the edges, since we have to exclude the edges adjacent to s for the path constraints.

The polyhedral structure of the traveling salesman problem with precedence constraints was investigated in [BFP95] and a specialized branch-and-cut algorithm was presented in [AJR00].

4 The BCP-framework with Symbolic Constraints

We define the BCP-paradigm by a set of transition rules and then explain the role played by symbolic constraints. The description of an algorithm by transition rules is typical for CP-systems, but uncommon for BCP-systems. Nevertheless, we found this description useful to explore the exact requirements for the symbolic constraints to guarantee the correctness and termination of the algorithm. We define those requirements for symbolic constraints and prove that the BCP-method terminates when the requirements are met.

4.1 Transition Rules for BCP

We consider a minimization problem M formulated as an MILP over a set V of variables and use $\mathcal{F}$ to denote its set of feasible solutions. For a set $\mathcal{F}$ of solutions we use $C(\mathcal{F})$ to denote the set of all linear constraints that are valid for all assignments in $\mathcal{F}$. For a finite set P of linear constraints on the variables in V and a subset $A \subseteq V$ of variables, $LP(P, A)$ denotes the optimal objective function value of the linear program (P, A) with M's objective function where all variables in $V \setminus A$ are set to 0. If (P, A) is infeasible, $LP(P, A) = \infty$. For a set P of linear constrains, $\mathcal{F}_P = \{f \in \mathcal{F}; f \text{ satisfies all } c \in P\}$ denotes the feasible solutions of M selected by the constraints in P. For any set P of linear constraints feasible for M we have $LP(P, V) \leq LP(C(\mathcal{F}_P), V)$ and the value on the right side is the optimal objective value of any feasible solution in $\mathcal{F}_P$ since our objective function is linear. We also maintain a global upper bound g, the minimal objective value of any feasible solution encountered so far. Initially, $g = \infty$ and P and A are arbitrary subsets of $C(\mathcal{F})$ and V respectively.

The BCP-paradigm uses the following transition rules to transform problems (P, A, g). In each case, the old problem is given above the horizontal bar, the new problem is given below, and the condition for applicability is given to the right of the bar. The name of the rule is indicated on the left.

(Cut) $\quad \dfrac{P \quad A \quad g}{P \cup \{c\} \quad A \quad g}$ $\quad c$ is linear and valid for $\mathcal{F}$

(Price) $\quad \dfrac{P \quad A \quad g}{P \quad A \cup \{v\} \quad g}$ $\quad v \in V \setminus A$

(Branch) $\quad \dfrac{P \quad A \quad g}{P \cup \{c_1\} \quad A \quad g \quad | \quad P \cup \{c_2\} \quad A \quad g}$ $\quad c_1$ and c_2 are linear and $c_1 \vee c_2$ is valid for $\mathcal{F}$.

(Improved Bound) $\quad \dfrac{P \quad A \quad g}{P \quad A \quad g'}$ $\quad g'$ is the objective value of a feasible solution and $g' < g$

(Fathom) $\quad \dfrac{P \quad A \quad g}{\text{—}}$ $\quad LP(P, V) \geq g$

If the objective function is known to be integral, the condition of the fathom rule can be relaxed.

(Integral Fathom) $\quad \dfrac{P \quad A \quad g}{\text{—}}$ $\quad LP(P, V) > g - 1$

Notice that the fathom rule requires that the LP value for the whole set of variables has to be greater or equal to g. Otherwise the LP value might not be a lower bound on the objective value of the best feasible solution.

In the cut- and the branch-rule, one may alternatively require that c, respectively $c_1 \vee c_2$, are valid for $\mathcal{F}_P$, i.e., one works with locally valid cuts and branching disjunctions instead of globally valid ones.

In practice, many constraints (rather than just one) are usually added in a cut operation, and this is modelled above via the repeated application of the cut-rule. Similarly, multi-way (instead of 2-way) branching is modelled via the repeated application of the branch-rule.

4.2 The Role of Symbolic Constraints

We explain the role of symbolic constraints in rule application[2]. We are working on a subproblem (P, A, g) with LP-objective value $g' = LP(P, A)$. If $g' < \infty$, we also have an LP-solution s; s is an assignment to the variables in A. We can view it as an assignment to all variables by setting $s(v) = 0$ for $v \in V \setminus A$. We ask each symbolic constraint SC for its opinion about s; whether s is a feasible solution for SC or not and in the latter case, whether the constraint suggests a variable in $V \setminus A$ for the price rule, a cut c for the cut rule, or a disjunction $c_1 \vee c_2$ for the branching rule.

If s is feasible for all symbolic constraints we have found a new feasible solution to our minimization problem. If $g' < g$, we can improve the upper bound. The user may supply a heuristic that attempts to construct a feasible solution from an infeasible s. If s is feasible, we also try to apply the fathom rule or the price rule.

A variable v is *local* to a symbolic constraint SC if the variable is used in no constraint generated outside SC. This includes especially branching constraints. We also say that SC owns v. The nice feature of local

[2]We will later discuss requirements on symbolic constraints and rule application which guarantee termination.

variables is that the symbolic constraint owning a variable can compute its reduced cost. Non-local variables are called *shared*. We may apply the fathom rule if A contains all shared variables and the local variables of all symbolic constraints have non-negative reduced cost. We initialize A with the set of shared variables and hence it suffices to ensure that the local variables have non-negative reduced cost. Therefore, the symbolic constraints are asked to suggest variables for the price rule. In symbolic constraints that contain too many variables to compute the reduced costs directly the pricing problem is solved in an algorithmic way.

We require:

Requirement 1: If a symbolic constraint has local variables of negative reduced cost, it suggests one; if $g' = \infty$, all local variables are considered to have negative reduced cost.

So assume that $g' < g$ and s is infeasible for M. We ask the symbolic constraints to supply cuts and/or disjunctions for the branching rule. A symbolic constraint may supply any cut or disjunction that is feasible for its feasible solutions (or for its feasible solutions selected by P). The built-in integrality constraints always suggest the obvious branching rules for fractional variables. We require:

Requirement 2: If s satisfies all integrality constraints, but does not satisfy a particular symbolic constraint, the constraint must suggest a cut c or a disjunction $c_1 \vee c_2$ which cuts off s, i.e., in the case of a cut, s does not satisfy $P \cup c$, and in the case of a disjunction, s satisfies neither $P \cup c_1$ nor $P \cup c_2$.

4.3 Termination

Having defined the BCP-framework with symbolic constraints, we now argue termination under the following assumptions:

1. The minimization problem is either infeasible or has a finite optimum and we start with a set of constraints that guarantees that none of the LPs generated is unbounded[3].
2. The integral variables are bounded, i.e., for each integral variable v there are constants l_v and u_v and constraints $l_v \leq v \leq u_v$.
3. Every symbolic constraint satisfies requirements 1 and 2.
4. **Requirement 3:** Every symbolic constraint SC has the strong termination property, i.e., the method terminates if it is run just with constraint SC and if an adversary is allowed to modify subproblems (P, A, g) by adding an arbitrarily chosen set P' of additional constraints; the LP-solver is then required to return an optimal solution for the problem $(P \cup P', A, g)$.

Lemma 1 *In every step of the algorithm, at least one rule is applicable.*

Proof. Assume otherwise. Since no symbolic constraint provides a cut or a branching rule, all symbolic constraints accept the solution of the LP solver. If the value of the LP solution is smaller than g, we can apply the rule (Improved Bound), otherwise we can apply the rule (Fathom). Thus, we can apply a rule in any case, a contradiction. □

Theorem 1 *If assumptions (1) to (4) are satisfied, the method terminates.*

Proof. Assume to the contrary that the method will not terminate. Thus there will be an infinite path in the computation tree. Let (P_i, A_i) be the LP-problems along this path, let g_i be its objective value, and, if $g_i < \infty$, let s_i be an optimal LP-solution.

Whenever $g_i \geq g$ (= the best objective value known) or s_i is feasible for all symbolic constraints, a variable is added to the problem (since we cannot apply the fathom rule). Since the number of variables is finite, there is an i_0 such that for each $i \geq i_0$ we have $g_i < g$ and s_i is infeasible for at least one constraint.

Consider $i \geq i_0$. Then s_i is infeasible for at least one constraint. Thus at least one of the violated symbolic constraints suggests a cut or a disjunction. Thus, we can apply the cut or the branch rule. Since our symbolic constraints have the strong termination property, each symbolic constraint proposes only a finite number of cuts or disjunctions.

We conclude that the method terminates. □

[3] We make this assumption because an optimal solution to an unbounded LP contains a ray and, as stated, symbolic constraints only know how to exploit finite solutions.

Lemma 2 *The integrality constraint satisfies Requirements 1, 2 and 3.*

Proof. Look at the integrality constraint for the variable x and let c be the current LP-value of x. If c is integral, the integrality constraint accepts the solution otherwise it suggests the branching rule $x \leq \lfloor c \rfloor \vee x \geq \lfloor c \rfloor + 1$ thus requirement 2 is satisfied. Since every integral variable is bounded, the integrality constraint proposes only a finite number of branching rules and hence requirement 3 is satisfied. Clearly requirement 2 is satisfied, since the constraint has no local variable. □

Which of our assumptions are necessary? Clearly, every constraint should terminate by itself. Thus requirements 1 and 2 are needed. It should also terminate when cooperating with other constraints; this suggests a requirement along the lines of the strong termination property. Boundedness of the optimal solution is a technical requirement. One can probably overcome it, by properly defining how symbolic constraints must react to solutions of unbounded LPs. We would like to remove the requirement that integral variables have to be bounded. Note that previous MILP-sovlers make the same assumption. The following example illustrates the difficulty.

Consider a mixed minimization problem with integral variables x and y, objective function $2x - 2y$ and constraints $x - y \geq -1/2$, $x \geq 0$, $y \geq 0$. Only the integrality constraints generate branching disjunctions. No matter which branching disjunctions are generated there will always be a number M and a subproblem whose associated LP allows the solutions $x = m + 1/2$, $y = m$, for all $m \geq M$. Its objective function is -1, whereas the objective function value of the optimal solution is 0. Thus we cannot apply the stopping rule and we get a infinite path in the computation tree.

4.4 Order of Rule Application

The efficiency of the BCP-method depends on the order of rule application. In most applications preferring cutting to branching turns out to be advantageous. Pricing should be performed before cutting.

For each of the rules there are many strategies to deal with the situation that we can choose from several possibilities. Take for example the situation that more than one violated constraint is detected. Which of them should be added to the system? A popular strategy is to add constraints according to their violation and disregard those with small violation. Another strategy is to rank cutting planes with respect to the Euclidean distance of the previously found LP solution to the hyperplanes defined by the violated constraints. ABACUS implements both strategies and makes them available to SCIL. The user of SCIL may implement his or her own strategy.

Another example is the choice between several branching rules. One strategy (*strong branching*) tentatively tries each of the suggested rules by solving the two resulting LPs. The rank of the disjunction is the smaller of the LP-values. We select the branching disjunction of largest rank. This is computationally expensive and might not pay. But in most cases it decreases the enumeration tree significantly. SCIL supports strong branching as well as other branching rules for branching on variables. The user may implement his or her own rules.

5 Implementation of Symbolic Constraints

We first discuss how LP matrices are specified, then we discuss the implementation of the symbolic constraint for Hamiltonian cycles.

5.1 The LP-Matrix

Every pair of variable and linear constraint has an entry in the LP-matrix. The default value is 0. There are several ways to create coefficients different from 0. These methods are very similar to the methods provided by typical LP solvers. Unless specified otherwise, non-zero values are not stored explicitly.

The basic method defines a single entry of the matrix, e.g., the call

```
P.set_coefficient(c,v,5);
```

sets the coefficient corresponding to constraint c and variable v to 5. `P` is either the main problem or a subproblem.

Frequently, we want to generate rows or columns in the LP-matrix which follow a common pattern. Consider, for example, the degree equations for the TSP. There are as many degree equations as nodes in the graph and all of them generate a 1 in the LP-matrix for all edges which are incident to the node. SCIL provides *constraint-schemas* and *variable-schemas* to implement patterns. In order to implement a constraint-schema, one derives a class from the system class `cons_obj`. The non-zero entries for the constraint are created in a specific member function. We use the degree equation as an example.

```
class degree_equation : public cons_obj {
  <<private part>>

  degree_equation(const graph& G_, node v_, var_map<edge>& VM_, double  rhs_);
  // the constructor gets a graph, a node, a mapping from edges to variables,
  // and a right hand side.

  virtual void non_zero_entries(row& r)
  // generate the row for v
  { edge e;
    // generate a 1 for each edge e incident to v
    forall_adj_edges(e, v) { r += 1*VM[e]; }
  };
};
```

We will use the degree constraint in the definition of the tour constraint below.

5.2 The Implementation of the Tour Constraint

We sketch the implementation of the tour constraint. The tour constraint has no local variables and hence it must provide only three functions beside the constructor, a function `init` that adds an initial set of constraints, a function `is_feasible` that determines whether the current LP-solution satisfies the constraint, and a function `separate` that generates cuts. We give more details.

```
class TOUR : public sym_constraint {

  <<private data>>

  TOUR(const graph& g_, const var_map<edge>& VM_);
  // we save  references to g_ and VM_ in local variables g and VM and
  // certify that the variables associated with the edges are binary.

  void init(subproblem& S) // add the degree constraints for all nodes
  { node v;
    forall_nodes(v,g) S.add_constraint(new degree_equation(g,v,VM,2));
  }

  status feasible(subproblem& S);
  // return feasible if all variables are integral and if the subgraph
  // induced by the variables with value 1 is connected.

  status separate(subproblem& S);
  // try to add a violated subtour elimination inequality or comb inequality
  // and return whether an inequality was added
};
```

In the constructor the graph and the variables that correspond to the edges are given. The precondition is that all given variables are binary. We check the precondition and save references to the graph and to the variable-map.

The init function simply adds degree equations for all nodes of g. It iterates over all nodes and instantiates an instance of the inequality schema `degree_equation` sketched above. We add the degree equation permanently to the LP, i.e., we do not allow the system to remove them.

An LP-solution is feasible if it is integral and if the subgraph induced by the variables with value 1 forms a tour. Since we know that the degree equations are added permanently to the LPs, they are always satisfied. Hence, an integer solution is a collection of cycles and a tour if and only if it contains no subtour, i.e., none of the subtour constraints may be violated. We check this using the separation routine for subtour constraints described next.

The member function that needs most work is the `separate` function. In a first implementation we might look for violated subtour elimination constraints. In the presence of degree constraints, the subtour elimination constraints are equivalent to the fact that for each subset $S \subset V$ of the nodes with $\emptyset \neq S \neq V$ the induced cut has value at least two. The value of a cut is the sum of the values of the LP-variables associated with edges cross the cut. A cut of value less than two corresponts to a violated subtour elimination cosntraint. It can be found efficiently by a minimum capacity cut computation.

We need to argue that our implementation of the tour constraint satisfies requirements 1 and 2 and has the strong termination property. It has property 1 since it has no local variables, it has property 2 since any integral solution of the degree constraints consists of a set of cycles and hence is either a tour or violates a subtour elimination constraint. The strong termination property holds since there are only finitely many subtour elimination constraints and since all variables are binary.

There are many methods known to improve the performance of the TSP solvers implemented above. All of them are easily added to the implementation of our symbolic constraint. A possible modification may be a branching rule different from the standard rule of branching on fractional integer variables. Some TSP codes reduce the size of the enumeration tree by branching on a hyperplane.

5.3 A Hierarchy of Constraints

The tour constraint uses integrality constraints and packages a family of linear constraints. More generally, any previously defined constraint can be used to define a new constraint. At the bottom of the hierarchy, we have linear constraints. The LP-solver knows how to handle them. A new symbolic constraint is formed in analogy to the tour constraint of the preceding section. The `init`-function specifies the initial set of constraints (for example, by calling the `init`-functions of the constituent symbolic constraints and by directly generating some linear constraints), the `is_feasible`-function decides feasibility (for example, by calling the `is_feasible`-function of the constituent constraints and by performing additional tests), ...

6 Utilizing Structure with Lagrangean Relaxation

A SCIL program can be seen as a collection of constraints together with an objective function. Symbolic constraints describe combinatorial optimization problems. Some of them are computationally hard but some can be solved in polynomial time. In many applications the core of the problem is particularly easy but side constraints make it hard to solve. An illustrative example is the *resource constraint assignment problem* (RCAP). We are given n jobs, each to be assigned to exactly one of n machines. Performing job i on machine j gives rise to a cost of w_{ij} and needs a_{ij}^k units of a resource k ($k = 1, \dots, m$). The resources are limited by d^k. The RCAP is to find an assignment of jobs to machines with minimum cost which requires at most d^k units of each resource k.

The problem can be modeled as follows. We introduce a complete bipartite graph $G = (V, E)$ with bipartition $V = J \cup M$ and $|J| = |M| = n$ representing jobs and machines. For each edge $e \in E$ we define a binary variable x_e which indicates if the edge $e = (i, j)$ is selected, i.e., if job i is submitted to machine j . The program below represents the RCAP.

$$\begin{array}{lll} (RCAP) & \min & \sum_{e \in E} w_e x_e \\ & \text{s.t.} & \sum_{e \in E} a_e^k x_e \leq d^k \text{ for all } k \in \{1, \dots, m\} \\ & & x \text{ defines an assignment in } G \end{array}$$

It is well known that this problem is $\mathcal{NP}$-hard already for one resource constraint. But the core of the problem is an easy assignment problem. Since `assignment` is one of SCIL's predefined symbolic constraints, we could submit the problem to SCIL and solve it with BCP. But we possibly can do better using Lagrangean relaxation. This is particularly true if we want to emphasize on finding "good" solutions fast without claiming optimality.

Lagrangean relaxation is a way to compute the optimum solution of an LP by restricting to a subset of the constraints and relaxing the others. For the RCAP we relax the resource constraints and receive an easy to solve assignment problem. The relaxation is done in the following way. For each relaxed constraint we introduce a *Lagrangean multiplier* $\lambda_k \in \mathbb{R}_{\geq 0}$ $(1 \leq k \leq m)$. For any vector of non-negative Lagrangean multipliers $\lambda \in \mathbb{R}^m_{\geq 0}$ the value of the optimum solution of the following *Lagrangean subproblem* D_λ yields a lower bound on the optimum value of the original problem.

$$(D_\lambda) \qquad L(\lambda) = \quad \min \quad \sum_{e \in E} w_e x_e + \sum_{k=1}^{m} \lambda_k (d^k - \sum_{e \in E} a_e^k x_e)$$

$$\text{s.t.} \quad x \text{ defines an assignment in } G$$

This is easy to see since all feasible solutions of RCAP are also feasible for D_λ and the Lagrangean multipliers are non-negative. The best lower bound is given by the *Lagrangean dual*

$$l^* := \max_{\lambda \in \mathbb{R}^m_{\geq 0}} L(\lambda)$$

The function $L(\lambda)$ is piecewise linear and its optimum can be computed in different ways. The most prominent method is subgradient optimization with all its variants like the *volume algorithm* [BA00]. During subgradient optimization a series of Lagrangean subproblems is solved for a series $(\lambda^{(t)})$ of Lagrangean multipliers until $(L(\lambda^{(t)}))$ converges to l^*. See [Lem01] for details on Lagrangean relaxation.

Approximating the LP bound of the original problem by Lagrangean relaxation can be advantageous for several reasons. Most important is the speedup of computation by solving easier subproblems. Another reason may be the computation of feasible solutions. During subgradient optimization one gets a series of "solutions" which may violate relaxed constraints. Since violated constraints tend to get larger Lagrangean multipliers during subgradient optimization they are likely to become unviolated in later iterations of the algorithm. This often results in good feasible solutions of the original problem. In order to ensure that the optimum solution is found the method must be embedded in a branch and bound framework. SCIL provides this.

Modeling Lagrangean relaxation in SCIL is as easy as modeling ILPs. The SCIL program below demonstrates the principles for a RCAP with one additional resource constraint.

```
void compute_RCAP(graph& G, edge_array<double>& w, edge_array<double>& a,
                  double d, list<edge>& assignment)
{
  // define a minimization problem P
  ILP_problem  P(optsense_min);

  // add a 0-1 variable for each edge e of G and make w[e] it coefficient in the
  // objective function. The var_map EV establishes the connection between edges
  // and variables
  var_map<edge> EV(nil);
  forall_edges(e,G) EV[e] = P.add_binary_variable(w[e]);

  // add the assignment constraint for G
  P.add_constraint(new ASSIGNMENT(G, EV));

  // add a relaxed <= inequality for the
  row r;
  forall_edges(e,G) r+=a[e]*EV[e];
  P.add_relaxed_constraint(r<=d);

  // solve P
  P.solve_relaxed();

  // extract the solution
  forall_edges(e,G) if ( P.get_value(EV[e]) ==  1 ) assignment.add(e);
}
```

Lagrangean relaxation cannot be used for any ILP Problem in SCIL. Some requirements must be met. Lagrangean relaxation in SCIL only works for symbolic constraints without local variables. Each non-relaxed constraint must provide a function `solve`, which solves the subproblem defined by a symbolic constraint for a given objective function. Non-relaxed constraints must be independent, i.e., they must not share variables.

7 Numerical Problems

SCIL aims at making the BCP-paradigm more easily accessible. We hope to attract users who are not aware of the inexactness of present LP-solvers. Current LP-solvers use floating point arithmetic and are not guaranteed to solve linear programs optimally: an infeasible LP may be declared feasible or vice versa. The solution computed for a feasible LP may be suboptimal or even infeasible. A non-integral solution may be computed for an LP whose optimal solution is integral.

The presented implementation of SCIL presupposes an exact LP-solver. Since we run it with an inexact LP-solver, SCIL is not guaranteed to terminate or to find optimal solutions. We take some precautions, e.g., testing linear constraints for ϵ-violation instead of strict violation, but these safe guards are known to fail sometimes. *We consider it a major challenge to develop exact LP-solvers and/or BCP-solvers with only a moderate, say a factor of less than 100, increase in running time.*

We will describe a solver which is guaranteed to terminate in a forthcoming paper. We also describe a solver which is guaranteed to compute correct solution for pure integer programs.

8 Conclusion

We gave the linear framework how to enrich integer programming solvers by symbolic constraints. The success of the system strongly depends on the quality of the symbolic constraints in the library. On the one hand they must be general enough to have a broad area of applications, on the other hand, they must be simple enough to have an efficient implementation.

References

[ABCC99] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Tsp-solver “concorde”. `http://www.keck.caam.rice.edu/concorde.html`, 1999.

[ADH+87] Abderrahmane Aggoun, M. Dincbas, A. Herold, H. Simonis, and P. Van Hentenryck. The CHIP System. Technical Report TR-LP-24, ECRC, Munich, Germany, June 1987.

[AJR00] N. Ascheuer, M. Jünger, and G. Reinelt. A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17(1):61–84, 2000.

[AKLM00] E. Althaus, O. Kohlbacher, H.-P. Lenhof, and P. Müller. A combinatorial approach to protein docking with flexible side-chains. In Ron Shamir, Satoru Miyano, Sorin Istrail, Pavel Pevzner, and Michael Waterman, editors, *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB-00)*, pages 15–24. ACM Press, 2000.

[AM01] Ernst Althaus and Kurt Mehlhorn. Traveling salesman-based curve reconstruction in polynomial ti me. *SIAM Journal on Computing*, 31(1):27–66, 2001.

[BA00] F. Barahona and R. Anbil. The volume algorithm: Producing primal solutions with a subgradient method. *Mathematical Programming A*, 87, 2000.

[BC94] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97 – 123, 1994.

[BFP95] E. Balas, M. Fischetti, and W. R. Pulleyblank. The precedence-constrained asymmetric traveling salesman polytope. *Mathematical Programming*, 68:241–265, 1995.

[BJN+98] C. Barnhart, E. Johnson, G. Nemhauser, M. Savelsbergh, and P. Vance. Branch-and-price: column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[BK98] A. Bockmayr and T. Kasper. Branch and infer: a unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10:287–300, 1998.

[CF97] A. Caprara and M. Fischetti. *Annotated bibliographies in combinatorial optimization*, chapter Branch-and-cut algorithms, pages 45–64. Wiley, 1997.

[Cor02] GAMS Development Corporation. Gams: General algebraic modeling system, 2002.

[CPL] CPLEX. `www.cplex.com`.

[Das00] Dash Associates. *XPRESS 12 Reference Manual: XPRESS-MP Optimizer Subroutine Library XOSL*, 2000.

[DFJ54] G.B. Dantzig, D.R. Fulkerson, and S.M. Johnson. Solution of a large scale traveling salesman problem. *Operations Research*, 2:393–410, 1954.

[EGJR01] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. *Computational Combinatorial Optimization*, volume 2241 of *Lecture Notes in Computer Science*, chapter Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS, pages 157–222. Springer, 2001.

[FGK92] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A modeling language for Mathematical Programming.* Duxbury Press/Wadsworth Publishing, 1992.

[GJR84] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[ILO] Ilog. `www.ilog.com`.

[JT00] M. Jünger and S. Thienel. The abacus system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience*, 30:1325–1352, 2000.

[Kas98] Thomas Kasper. *A Unifying Logical Framework for Integer Linear Programming and Finite Domain Constraint Programming.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1998.

[LED] LEDA (Library of Efficient Data Types and Algorithms). `www.algorithmic-solutions.com`.

[Lem01] C. Lemaréchal. *Computational Combinatorial Optimization*, volume 2241 of *Lecture Notes in Computer Science*, chapter Lagrangean relaxation, pages 112–156. Springer, 2001.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999. 1018 pages.

[Nad02] D. Naddef. *The Traveling Salesman Problem and its Variations*, chapter Polyhedral theory, branch and cut algorithms for the symmetric traveling salesman problem. Kluwer Academic Publishing, 2002.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[PR91] M. Padberg and G. Rinaldi. A branch and cut algorithm for the resolution of large scale symmetric traveling salesman problems. *SIAM Review*, 33(60–10), 1991.

[SDJ+96] C. Simone, M. Diehl, M. Jünger, P. Mutzel, and G. Rinaldi. Exact ground states of two-dimensional $\pm J$ ising spin glasses. *Journal of Statistical Physics*, 84:1363–1371, 1996.

[Smo96] Gert Smolka. Constraints in OZ. *ACM Computing Surveys*, 28(4es):75, December 1996.

[VBJN94] P.H. Vance, C. Barnhart, E.L. Johnson, and G. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3:111—130, 1994.

[vHPP00] Pascal van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, 2000.