Implementation with SCIP

Ambros M. Gleixner

slides by Timo Berthold, Stefan Heinz, and Kati Wolter

1 December 2011
Given

- Complete graph \( G = (V, E) \)
- for each \( e \in E \) a distance \( d_e > 0 \)

Binary variables

- \( x_e = 1 \) if edge \( e \) is used
TSP – Integer Programming Formulation

Given

▷ Complete graph $G = (V, E)$
▷ for each $e \in E$ a distance $d_e > 0$

Binary variables

▷ $x_e = 1$ if edge $e$ is used

$$\begin{align*}
\text{min} & \quad \sum_{e \in E} d_e x_e \\
\text{subject to} & \quad \sum_{e \in \delta(v)} x_e = 2 & \forall v \in V \\
& \quad \sum_{e \in \delta(S)} x_e \geq 2 & \forall S \subset V, S \neq \emptyset \\
& \quad x_e \in \{0, 1\} & \forall e \in E
\end{align*}$$
Given

- Complete graph $G = (V, E)$
- For each $e \in E$ a distance $d_e > 0$

Binary variables

- $x_e = 1$ if edge $e$ is used

$$\min \sum_{e \in E} d_e x_e$$

subject to

- $\sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V$
- $\sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, S \neq \emptyset$
- $x_e \in \{0, 1\} \quad \forall e \in E$
TSP – Integer Programming Formulation

Given

- Complete graph $G = (V, E)$
- For each $e \in E$ a distance $d_e > 0$

Binary variables

- $x_e = 1$ if edge $e$ is used

\[
\begin{align*}
\text{min} & \quad \sum_{e \in E} d_e x_e \\
\text{subject to} & \quad \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \\
& \quad \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, S \neq \emptyset \\
& \quad x_e \in \{0, 1\} \quad \forall e \in E
\end{align*}
\]
Given

- Complete graph $G = (V, E)$
- For each $e \in E$ a distance $d_e > 0$

Binary variables

- $x_e = 1$ if edge $e$ is used

\[
\begin{align*}
\text{min} & \quad \sum_{e \in E} d_e x_e \\
\text{subject to} & \quad \sum_{e \in \delta(v)} x_e = 2 & \forall v \in V \\
& \quad \sum_{e \in \delta(S)} x_e \geq 2 & \forall S \subset V, S \neq \emptyset \\
& \quad x_e \in \{0, 1\} & \forall e \in E
\end{align*}
\]
Given

- Complete graph \( G = (V, E) \)
- for each \( e \in E \) a distance \( d_e > 0 \)

Integer variables

- \( x_v \) position of \( v \in V \) in tour
TSP – Constraint Programming Formulation

Given

- Complete graph $G = (V, E)$
- For each $e \in E$ a distance $d_e > 0$

Integer variables

- $x_v$ position of $v \in V$ in tour

\[
\begin{align*}
\text{min} & \quad \text{length}(x_1, \ldots, x_n) \\
\text{subject to} & \quad \text{alldifferent}(x_1, \ldots, x_n) \\
& \quad x_v \in \{1, \ldots, n\} \\
& \quad \forall v \in V
\end{align*}
\]
The Heart of the CIP Concept

<table>
<thead>
<tr>
<th>Mixed Integer Program</th>
<th>Constraint Integer Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Linear objective function</td>
<td>▶ Linear objective function</td>
</tr>
<tr>
<td>▶ <strong>Linear constraints</strong></td>
<td>▶ <strong>Arbitrary constraints</strong></td>
</tr>
<tr>
<td>▶ Real and integer variables</td>
<td>▶ Real and integer variables</td>
</tr>
<tr>
<td>▶ After fixing all integer variables: CIP becomes an LP</td>
<td></td>
</tr>
</tbody>
</table>

Remark:
▶ Arbitrary objective or variables modeled by constraints

→ For each type of constraint, one constraint handler is responsible.
## Characteristics

### Objective function:
- **linear function**

### Feasible region:
- described by arbitrary constraints
- after fixing all integer vars: CIP becomes an linear program (LP)

### Variable domains:
- real or integer values

### Mathematical Formulation

\[
\begin{align*}
\text{min} & \quad \sum_{e \in E} d_e x_e \\
\text{s.t.} & \quad \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \\
& \quad \text{nosubtour}(x) \\
& \quad x_e \in \{0, 1\} \quad \forall e \in E
\end{align*}
\]

(CIP formulation of TSP)

Single nosubtour constraint rules out subtours (e.g. by domain propagation). It may also separate subtour elimination inequalities.
Constraint Integer Programming

- Mixed Integer Programs

MIP
Constraint Integer Programming

- Mixed Integer Programs
- Satisfiability
- **Mixed Integer Programs**
- **Satisfiability**
- **Pseudo-Boolean**
Constraint Integer Programming

- Mixed Integer Programs
- Satisfiability
- Pseudo-Boolean
- Finite Domain
Constraint Integer Programming

- Mixed Integer Programs
- Satisfiability
- Pseudo-Boolean
- Finite Domain
- Constraint Programming
Constraint Integer Programming

- Mixed Integer Programs
- Satisfiability
- Pseudo-Boolean
- Finite Domain
- Constraint Programming
- Constraint Integer Programming
Combination of ...

- LP relaxation
- branch-and-bound
- cutting planes
Combination of ...

- LP relaxation
- branch-and-bound
- cutting planes
Combination of...

- LP relaxation
- branch-and-bound
- cutting planes
Combination of ...

- branch-and-bound
- domain propagation
Combination of ...

- branch-and-bound
- domain propagation

Alldifferent constraint:

\[ \{x_1, x_2, x_3, x_4\} \in \{1, 2, 3, 4\} \]

pairwise different.

\[ x_1 \neq x_2 \neq x_3 \neq x_4 \]
SCIP Facts

- Approx. 275,640 lines of C code
  - 18% documentation
  - 20% assertions
- 7 examples illustrating the use of SCIP
- HowTos: each plugin type, debugging, automatic testing, ...
- C++ wrapper classes
- 7 interfaces to external linear programming solvers
  - CLP, CPLEX, Gurobi, Mosek, QSopt, SoPlex, XPRESS
- 6 different input formats
  - cnf, flatzinc, lp, mps, opb (pseudo-boolean), zimpl
- 876 parameters to play with
- Part of the ZIB Optimization Suite http://zibopt.zib.de
Performance of SCIP

-One of the fastest noncommercial MIP solver

![Comparison of MIP solvers](image)

- Results taken from H. Mittelmann (09/17/2009)
Performance of SCIP

- One of the fastest noncommercial MIP solver

![Graph showing performance comparison between SCIP and other solvers]

- Performance development

![Graph showing performance development over time]

results taken from H. Mittelmann (09/17/2009)
Performance of SCIP

▷ One of the fastest noncommercial MIP solver

![Graph showing performance comparison between non-commercial and commercial solvers. The graph indicates SCIP 1.2 as the winner in terms of performance, with a speedup of 15.2× compared to other solvers.](image)

▷ Performance development

![Graph showing performance development over time. The graph indicates a significant improvement in performance from SCIP 0.7 to SCIP 1.1, with a speedup of 7.48×.](image)

▷ Winner of the “Pseudo-Boolean Evaluation 2009”

http://www.cril.univ-artois.fr/PB09/
ZIP Optimization Suite = SCIP + SoPlex + ZIMPL

Tool for **generating** and **solving** constraint integer programs

**ZIMPL**
- A mixed integer programming modeling language
- Easily generating linear programs and mixed integer programs

**SCIP**
- A mixed integer programming solver and constraint programming
- ZIMPL models can directly be loaded into SCIP and solved

**SoPlex**
- A linear programming (LP) solver
- Solution process SCIP may use SoPlex as underlying LP solver

All three tools are available in source code and free for academic use
Important Aspects

- SCIP is constraint based
  - Advantage: flexibility
  - Disadvantage: limited global view

- A constraint knows its variables, but a variable does not know the constraints it appears in.

- A single constraint may represent a whole set of inequalities, not only a single one.

- From the constraint programming perspective:
  - LP-relaxation is only an add-on!
  - Many constraints do not separate inequalities at all.
Operational Stages

Init

Problem → Transforming → Presolving → Solving

Init Solve

Free Transform → Free Solve

Basic data structures are allocated and initialized.
User includes required plugins (or just takes default plugins).
Basic data structures are allocated and initialized.
User includes required plugins (or just takes default plugins).
Main SCIP interface: plugins.

- Perform all problem specific actions.
- Each step calls user defined plugins.
- SCIP knows of plugins through “include” functions.
- Plugins may have private data.
- User defined callback functions (virtual functions in C++-interface).
- Yields modular structure.
- The MIP solver is realized through plugins.

Everything SCIP knows, it knows through plugins.
Types of Plugins

- **Constraint handler:** assures feasibility, strengthens formulation
- **Separator:** adds cuts, improves dual bound
- **Pricer:** allows dynamic generation of variables
- **Heuristic:** searches solutions, improves primal bound
- **Branching rule:** how to divide the problem?
- **Node selection:** which subproblem should be regarded next?
- **Presolver:** simplifies the problem in advance, strengthens structure
- **Propagator:** simplifies problem, improves dual bound locally
- **Reader:** reads problems from different formats
- **Event handler:** catches events (e.g., bound changes, new solutions)
- **Display:** allows modification of output
User creates and modifies the original problem instance.

Problem creation is usually done in file readers (SCIPreadProb()).
Transforming

- Creates a working copy of the original problem.
- data are copied into separate memory area
- presolving and solving operate on transformed problem
- original data can only be modified in problem modification stage
Presolving

Problem \rightarrow Transforming \rightarrow Presolving \rightarrow Solving

Init \rightarrow Init Solve

Free Transform \rightarrow Free Solve
Presolvers provide global presolving (e.g. dual fixing)

Constraint Handlers provide constraint specific presolving, e.g.:
- Domain tightening,
- Coefficient modification,
- Deletion of redundant constraints,
- Constraint upgrading.
- ...
SCIP Structures

Ambros M. Gleixner (ZIB)
Flow Chart SCIP

- Presolving
- Stop
- Node selection
  - Conflict analysis
  - Processing
    - Primal heuristics
    - Branching
  - Solve LP
    - Pricing
    - Cuts
    - Enforce constraints
    - Domain propagation
Flow Chart SCIP

- Presolving
- Node selection
- Processing
- Branching
- Conflict analysis
- Primal heuristics
- Solve LP
- Pricing
- Cuts
- Enforce constraints
- Domain propagation
- Stop
Propagatore solvers provide local presolving

Constraint Handlers provide constraint specific propagation, e.g.:
- Domain tightening,
- Coefficient modification,
- Deletion of redundant constraints,
- Constraint upgrading.
- ...
Flow Chart SCIP

Presolving

Stop

Node selection

Conflict analysis

Processing

Primal heuristics

Branching

Solve LP

Pricing

Cuts

Domain propagation

Enforce constraints
Flow Chart SCIP

Presolving

Stop

Node selection

Conflict analysis

Processing

Primal heuristics

Branching

Solve LP

Domain propagation

Pricing

Cuts

Enforce constraints

Enforce constraints
Pricing: allows dynamic generation of variables
Flow Chart SCIP

- **Presolving**
- **Node selection**
- **Processing**
- **Branching**
- **Conflict analysis**
- **Primal heuristics**
- **Solve LP**
- **Domain propagation**
- **Pricing**
- **Cuts**
- **Enforce constraints**
- **Stop**

- Creates a branching with the infeasible solution no longer being feasible in the relaxations of the child nodes.
Primal heuristics try to find feasible solutions (in addition to feasible LP solutions).
Flow Chart SCIP

- Presolving
- Stop
- Node selection
  - Conflict analysis
  - Primal heuristics
  - Branching
- Processing
- Solve LP
  - Domain propagation
  - Pricing
  - Cuts
- Enforce constraints

 Conflict analysis learns from infeasible subproblems
Default Plugins

Constraint Handler

xor
bound
sos2
sos1
setppc
or
logicor
linear

and
bound
disjunc.
count
sols
indicator
integral

knap
sack
Special Linear Constraints

\[ \begin{align*} 
\text{min} & \quad x_1 + x_2 + x_3 + x_4 + x_5 + y_1 + y_2 \\
\text{s.t.} & \quad y_1 \leq 3 + 4x_1 \\
& \quad y_2 \leq 4 + 5x_2 \\
& \quad 3x_1 + 2x_2 + 4x_3 \leq 8 \\
& \quad x_2 + x_3 + x_4 = 1 \\
& \quad 3x_1 + 4.5x_2 + 3.2x_5 + 0.8y_1 + 1.2y_2 \geq 4 \\
\end{align*} \]

\[ \begin{align*} 
& x_1, x_2, x_3, x_4 \in \{0, 1\} \\
& x_5 \in \mathbb{Z}_+ \\
& y_1, y_2 \in \mathbb{R}_+ \\
\end{align*} \]

presolved problem has 7 variables (4 bin, 1 int, 0 impl, 2 cont) and 5

2 constraints of type <varbound>
1 constraints of type <knapsack>
1 constraints of type <setppc>
1 constraints of type <linear>
\[ \begin{align*}
\text{min} & \quad x_1 + x_2 + x_3 + x_4 + x_5 + y_1 + y_2 \\
\text{s.t.} & \quad y_1 \leq 3 + 4x_1 \\
& \quad y_2 \leq 4 + 5x_2 \\
& \quad 3x_1 + 2x_2 + 4x_3 \leq 8 \\
& \quad x_2 + x_3 + x_4 = 1 \\
& \quad 3x_1 + 4.5x_2 + 3.2x_5 + 0.8y_1 + 1.2y_2 \geq 4 \\
\end{align*} \]

\[ x_1, x_2, x_3, x_4 \in \{0, 1\} \]
\[ x_5 \in \mathbb{Z}^+ \]
\[ y_1, y_2 \in \mathbb{R}^+ \]

The presolved problem has 7 variables (4 bin, 1 int, 0 impl, 2 cont) and 5 constraints of type \(<\text{varbound}>\), 2 constraints of type \(<\text{knapsack}>\), 1 constraints of type \(<\text{setppc}>\), and 1 constraint of type \(<\text{linear}>\).
min \quad x_1 + x_2 + x_3 + x_4 + x_5 + y_1 + y_2

s.t. \quad y_1 \leq 3 + 4x_1
\quad y_2 \leq 4 + 5x_2
\quad 3x_1 + 2x_2 + 4x_3 \leq 8
\quad x_2 + x_3 + x_4 = 1
\quad 3x_1 + 4.5x_2 + 3.2x_5 + 0.8y_1 + 1.2y_2 \geq 4

x_1, x_2, x_3, x_4 \in \{0, 1\}
\quad x_5 \in \mathbb{Z}_+
\quad y_1, y_2 \in \mathbb{R}_+

presolved problem has 7 variables (4 bin, 1 int, 0 impl, 2 cont) and 5
2 constraints of type <varbound>
1 constraints of type <knapsack>
1 constraints of type <setppc>
1 constraints of type <linear>
Special Linear Constraints

\[ \begin{array}{l}
\text{min} & x_1 + x_2 + x_3 + x_4 + x_5 + y_1 + y_2 \\
\text{s.t.} & y_1 \leq 3 + 4 x_1 \\
& y_2 \leq 4 + 5 x_2 \\
& 3 x_1 + 2 x_2 + 4 x_3 \leq 8 \\
& x_2 + x_3 + x_4 = 1 \\
& 3 x_1 + 4.5 x_2 + 3.2 x_5 + 0.8 y_1 + 1.2 y_2 \geq 4 \\
\end{array} \]

\[ x_1, x_2, x_3, x_4 \in \{0, 1\} \]
\[ x_5 \in \mathbb{Z}_+ \]
\[ y_1, y_2 \in \mathbb{R}_+ \]

presolved problem has 7 variables (4 bin, 1 int, 0 impl, 2 cont) and 5

2 constraints of type <varbound>
1 constraints of type <knapsack>
1 constraints of type <setppc>
1 constraints of type <linear>
\[
\begin{align*}
\text{min} \quad & x_1 + x_2 + x_3 + x_4 + x_5 + y_1 + y_2 \\
\text{s.t.} \quad & y_1 \leq 3 + 4 x_1 \\
& y_2 \leq 4 + 5 x_2 \\
& 3 x_1 + 2 x_2 + 4 x_3 \leq 8 \\
& x_2 + x_3 + x_4 = 1 \\
& 3 x_1 + 4.5 x_2 + 3.2 x_5 + 0.8 y_1 + 1.2 y_2 \geq 4 \\
\end{align*}
\]

\[x_1, x_2, x_3, x_4 \in \{0, 1\} \]
\[x_5 \in \mathbb{Z}_+ \]
\[y_1, y_2 \in \mathbb{R}_+ \]

The presolved problem has 7 variables (4 bin, 1 int, 0 impl, 2 cont) and 5 constraints of type <varbound>, 1 constraints of type <knapsack>, 1 constraints of type <setppc>, 1 constraints of type <linear>.
Knapsack Constraints

Feasible region of 0-1 knapsack problem:

\[ \{ x \in \{0, 1\}^{|N|} : \sum_{j \in N} a_j x_j \leq b \} \]

- weights of the variables: \( a_j \in \mathbb{Z}_+ \) for all \( j \in N \)
- capacity of the knapsack: \( b \in \mathbb{Z}_+ \)
Ingredients of a Constraint Handler

Callback Methods

- **Fundamental:**
  - CONSLOCK
  - CONSCHECK
  - CONSENFOLP, CONSENFOPS

- **Additional:**
  - CONSINIT..., CONSEXIT...
  - CONSSEPALP, CONSSEPASOL
  - CONSPROP, CONSRESPROP, CONSPRESOL
  - CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE, CONSTRANS, CONSDELETE, CONSFREE
  - CONSPRINT, CONSCOPY, CONSPARSE

Further Ingredients

- Private data
- Interface methods
- Properties/Parameters
Ingredients of a Constraint Handler

Callback Methods

▷ Fundamental:
  ◦ CONSLOCK
  ◦ CONSCHECK
  ◦ CONSENFOLP, CONSENFOPS

▷ Additional:
  ◦ CONSINIT..., CONSEXIT...
  ◦ CONSSEPALP, CONSSEPASOL
  ◦ CONSPROP, CONSRESPROP, CONSPRESOL
  ◦ CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE, CONSTRANS, CONDELETE, CONSFREE
  ◦ CONSPRINT, CONSCOPY, CONSPARSE

Further Ingredients

▷ Private data
▷ Interface methods
▷ Properties/Parameters
Private Data

Constraint data: information needed to define single constraint

```c
struct SCIP_ConsData
{
    SCIP_VAR** vars;    // variables in knapsack
    int nvars;          // number of variables
    SCIP_Longint* weights; // weights of variables
    SCIP_Longint capacity; // capacity of knapsack
};
```

Constraint handler data: information belonging to constraint handler itself

```c
struct SCIP_ConshdlrData
{
    int maxrounds;    // max nr. of sepa rounds per node
    int maxsepacuts;  // max nr. of cuts per sepa round
};
```
Creating a single knapsack constraint.

```c
SCIP_RETCODE SCIPcreateConsKnapsack(
    SCIP* scip,          // SCIP data structure
    SCIP_CONS** cons,    // pointer to hold created cons
    const char* name,    // name of constraint
    SCIP_VAR** vars,     // array with variables
    int nvars,           // number of variables in knapsack
    SCIP_Longint* weights, // array with weights
    SCIP_Longint capacity, // capacity of knapsack
    SCIP_Bool separate,  // should constraint be separated?
    ...
)
{
    SCIP_CONSDATA* consdata;
    SCIP_CALL( consdataCreate(scip, &consdata, nvars, vars,
                                weights, capacity) );
    SCIP_CALL( SCIPcreateCons(scip, cons, name, conshdlr,
                                consdata, separate, ...) );
    ...
}
```
Including the knapsack constraint handler.

```c
SCIP_RETCODE SCIPincludeConshdlrKnapsack(
    SCIP*       scip  // SCIP data structure
)
{
    SCIP_CONSHDLRDATA* conshdlrdata;

    SCIP_CALL( conshdlrdataCreate(scip, &conshdlrdata) );

    SCIP_CALL( SCIPincludeConshdlr(scip, CONSHDLR_CHECKPRIORITY,
                                     consCheckKnapsack, ..., conshdlrdata) );
    ...
}
```
Ingredients of a Constraint Handler

Callback Methods

- **Fundamental:**
  - CONSLOCK
  - CONSCHECK
  - CONSENFOLP, CONSENFOPS

- **Additional:**
  - CONSINIT..., CONSEXIT...
  - CONSSEPalp, CONSSEPasol
  - CONSPROP, CONSRESPROP, CONSPRESOL
  - CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSENDISABLE, CONSTRANS, CONSENDELETE, CONSFREE
  - CONSPRINT, CONSCOPY, CONSPARSE

Further Ingredients

- Private data
- Interface methods
- Properties/Parameters
Provides dual information for single constraints
(useful for presolving, primal heuristics, ...)

For each variable of a constraint, returns whether...
  - increasing its value, and/or
  - decreasing its value

may lead to a violation of the constraint

\[ 3x_1 - 5x_2 + 2x_3 \leq 7 \]

increasing: \( x_1 \) and \( x_3 \)

decreasing: \( x_2 \)
Most important callback ...  

- usually called by **primal heuristics**  
- checks given solution for **feasibility** wrt all constraints of its type  
- possible **result values**  
  - SCIP_FEASIBLE  
  - SCIP_INFEASIBLE  

Given solution:  
\[(x_1, x_2, x_3) = (1, 0, 1)\]  

Knapsack constraints:  
\[3x_1 + 6x_2 + 4x_3 \leq 8\]  
\[2x_1 + 2x_3 \leq 3\]  

Result: SCIP_INFEASIBLE
CONSENFOLP and CONSENFOPS

CONSENFOLP: checks LP solution for feasibility
CONSENFOPS: checks Pseudo solution for feasibility

LP solution

▷ solution of LP relaxation

Pseudo Solution

▷ solution of LP relaxation with only bound constraints
▷ used if LP solving disabled, or
▷ numerical difficulties occurred

\[
\begin{align*}
\text{min} & \quad x_1 - x_2 + x_3 \\
\text{s.t.} & \quad 3x_1 + 8x_2 + 4x_3 \leq 4 \\
& \quad x_1, x_2, x_3 \in \{0, 1\}
\end{align*}
\]

LP solution: \((0, \frac{1}{2}, 0)\)

\[
\begin{align*}
\text{min} & \quad x_1 - x_2 + x_3 \\
\text{s.t.} & \quad x_1, x_2, x_3 \in \{0, 1\}
\end{align*}
\]

Pseudo Solution: \((0, 1, 0)\)
LP solution may violate a constraint not contained in the relaxation.

Enforcement callbacks are necessary for a correct implementation!

In addition, they can resolve an infeasibility by . . .

- reducing a variable’s domain,
- separating a cutting plane (may use integrality),
- adding a (local) constraint,
- creating a branching,
- concluding that the subproblem is infeasible and can be cut off, or
- just saying “solution infeasible”.

Ambros M. Gleixner (ZIB)
Enforcement result of constraint handler:

- reduced domain
- added cut
- cutoff
- added constraint
- branched
- infeasible
- feasible
Enforcement result of constraint handler:

- reduced domain
- added cut
- cutoff
- infeasible
- reduced domain
- added cut
- branched
- feasible
Enforcement result of constraint handler:

- reduced domain
- added constraint
- added cut
- branched
- cutoff
- infeasible
- feasible
Enforcement result of constraint handler:

- reduced domain
- added cut
- cutoff
- infeasible
- added constraint
- branched
- feasible
Enforcement result of constraint handler:

- reduced domain
- added cut
- cutoff
- infeasible
- added constraint
- branched
- feasible
CONSENFOLP and CONSENFOPS

Enforcement result of constraint handler:
- reduced domain
- added cut
- added constraint
- branched
- cutoff
- infeasible
- feasible
Enforcement result of constraint handler:

- reduced domain
- added cut
- cutoff
- infeasible
- added constraint
- branched
- feasible
Enforcement result of constraint handler:

- reduced domain
- added cut
- cutoff
- infeasible
- added constraint
- branched
- feasible
Ingredients of a Constraint Handler

Callback Methods

▷ **Fundamental:**
  - CONSLOCK
  - CONSCHECK
  - CONSENFOLP, CONSENFOPS

▷ **Additional:**
  - CONSINIT..., CONSEXIT...
  - CONSSEPALP, CONSSEPASOL
  - CONSPROP, CONSRESPROP, CONSPRESOL
  - CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE, CONSTRANS, CONSDELETE, CONSFREE
  - CONSPRINT, CONSCOPIY, CONSPARSE

Further Ingredients

▷ Private data
▷ Interface methods
▷ Properties/Parameters
CONSINIT... , CONSEXIT...

CONSINIT and CONSEXIT:

- called after problem was transformed / before transformed problem is freed
- initialize and free statistics in SCIP_ConshdlrData
CONSINIT..., CONSEXIT...

CONSINITPRE and CONSEXITPRE:
▷ called before presolving starts / after presolving is finished
▷ initialize and free presolving data
CONSINIT..., CONSEXIT...

CONSINITSOL and CONSEXITSOL:
- called before branch-and-bound process starts / before branch-and-bound process is freed
- initialize and release branch-and-bound specific data
CONSINIT..., CONSEXIT...

CONSINITLP:
▷ called before first LP relaxation is solved
▷ add linear relaxation of all "initial" constraints to the LP relaxation
Feasible region of 0-1 knapsack problem:

\[
\{ x \in \{0, 1\}^{|N|} : \sum_{j \in N} a_j x_j \leq b \}
\]

**Minimal Cover: \( C \subseteq N \)**

- \( \sum_{j \in C} a_j > b \)
- \( \sum_{j \in C \setminus \{i\}} a_j \leq b \quad \forall i \in C \)

**Minimal Cover Inequality**

\[
\sum_{j \in C} x_j \leq |C| - 1
\]

Minimal cover: \( C = \{2, 3, 4\} \)

Minimal cover inequality: \( x_2 + x_3 + x_4 \leq 2 \)

5x_1 + 6x_2 + 2x_3 + 2x_4 \leq 8

separated in knapsack constraint handler
Separation is implemented in separators and constraint handlers.

1. Separators with \texttt{SEPA\_PRIORITY} \( \geq 0 \) (decreasing order)
2. Constraint handlers (decreasing order of \texttt{CONSHDLR\_SEPA\_PRIORITY})
3. Separators with \texttt{SEPA\_PRIORITY} < 0 (decreasing order)
Ingredients of a Constraint Handler

Callback Methods

▸ Fundamental:
  ▪ CONSLOCK
  ▪ CONSCHECK
  ▪ CONSENFOLP, CONSENFOPS

▸ Additional:
  ▪ CONSINIT..., CONSEXIT...
  ▪ CONSSEPALP, CONSSEPASOL
  ▪ CONSPROP, CONSRESPROP, CONSPRESOL
  ▪ CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE
  ▪ CONSTRANS, CONSENDELETE, CONSFREE
  ▪ CONSPRINT, CONSPARSE, CONSCOPY

▸ Domain propagation
  (during subproblem processing)

▸ Reason for domain reductions
  (for conflict analysis)

▸ Presolving
  (before processing root node)
Ingredients of a Constraint Handler

Callback Methods

▶ Fundamental:
- CONSLOCK
- CONSCHECK
- CONSENFOLP, CONSENFOPS

▶ Additional:
- CONSINIT..., CONSEXIT...
- CONSSEPALP, CONSESEPASOL
- CONSPROP, CONSPRESPROP, CONSPRESOL
- CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE
- CONSTRANS, CONSDELETE, CONSFREE
- CONSPRINT, CONSPARSE, CONSCOPY

Called whenever ...

▷ SCIP enters/leaves subtree where local constraint exists
▷ constraint is enabled/disabled (no propagation, no separation)
Ingredients of a Constraint Handler

Callback Methods

- **Fundamental:**
  - CONSLOCK
  - CONSCHECK
  - CONSENFOLP, CONSENFOPS

- **Additional:**
  - CONSINIT..., CONSEXIT...
  - CONSSEPALP, CONSSEPASOL
  - CONSPROP, CONSRESPROP, CONSPRESOL
  - CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE
  - CONSTRANS, CONSDELETE, CONSFREE
  - CONSPRINT, CONSPARSE, CONSCOPY

- **Displaying, parsing** problems
- **Copying** problems between different SCIP environments
Ingredients of a Constraint Handler

Callback Methods

▷ Fundamental:
  ▶ CONSLOCK
  ▶ CONSCHECK
  ▶ CONSENFOLP, CONSENFOPS

▷ Additional:
  ▶ CONSINIT..., CONSEXIT...
  ▶ CONSSEPALP, CONSESEPASOL
  ▶ CONSPROP, CONSRESPROP, CONSPRESOL
  ▶ CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE
  ▶ CONSTRANS, CONSDELETE, CONSFREE
  ▶ CONSPRINT, CONSPARSE, CONSCOPY

Further Ingredients

▷ Private data
▷ Interface methods
▷ Properties/Parameters
Exercise: Traveling Salesman Problem (TSP)

Definition

Given a complete graph $G = (V, E)$ with edge lengths $c_e$.

Find a Hamiltonian cycle (cycle containing all nodes, tour) of minimum length.

$K_8$
Exercise: Traveling Salesman Problem (TSP)

Definition

Given a complete graph $G = (V, E)$ with edge lengths $c_e$.

Find a Hamiltonian cycle (cycle containing all nodes, tour) of minimum length.

$K_8$
**Exercise: TSP**

### MIP Formulation

<table>
<thead>
<tr>
<th>Objective</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \min \sum_{e \in E} c_e x_e ]</td>
<td>[ \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V ]</td>
</tr>
<tr>
<td></td>
<td>[ \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, S \neq \emptyset ]</td>
</tr>
<tr>
<td></td>
<td>[ x_e \in {0, 1} \quad \forall e \in E. ]</td>
</tr>
</tbody>
</table>

### CIP Formulation

<table>
<thead>
<tr>
<th>Objective</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \min \sum_{e \in E} c_e x_e ]</td>
<td>[ \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V ]</td>
</tr>
<tr>
<td></td>
<td>[ \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, S \neq \emptyset ]</td>
</tr>
<tr>
<td></td>
<td>[ \text{nosubtour}(G, x) ]</td>
</tr>
<tr>
<td></td>
<td>[ x_e \in {0, 1} \quad \forall e \in E. ]</td>
</tr>
</tbody>
</table>

**nosubtour(G, x) \iff \nexists C \subseteq \{e \in E \mid x_e = 1\} : C \text{ is a cycle of length } |C| < |V|
Exercise: TSP

- Search Tree
- LP Relaxation
- Presolve Management
- Implication Graph
- Solution Pool
- Cut Pool
- Conflict Analysis
- MIP Default Plugins
Exercise: TSP

- Search Tree
- LP Relaxation
- Presolve Management
  - Implication Graph
  - Solution Pool
  - Cut Pool
  - Conflict Analysis
- MIP Default Plugins
- 2-Opt Heuristic
- TSP File Reader
- New Solution Event Handler
Exercise: TSP

- Search Tree
- LP Relaxation
- Presolve Management
  - Implication Graph
  - Solution Pool
  - Cut Pool
  - Conflict Analysis
- MIP Default Plugins
- 2-Opt Heuristic
- Nosubtour Constraint Hdlr
- LP Greedy Heuristic
- TSP File Reader
- New Solution Event Handler
Exercise: TSP

Callback Methods

- **Fundamental:**
  - CONSLOCK
  - CONSCHECK
  - CONSENFOLP, CONSENFOPS

- **Additional:**
  - CONSINIT..., CONSEXIT...
  - CONSSEPALP, CONSSEPEASOL
  - CONSPROP, CONSRESPROP, CONSPRESOL
  - CONSACTIVE, CONSDEACTIVE, CONSENABLE, CONSDISABLE
  - CONSTRANS, CONSDELETE, CONSFREE
  - CONSPRINT, CONSPARSE, CONSCOPY

Further Ingredients

- Private data
- Interface methods
- Properties
Exercise: TSP

Callback Methods

▶ Fundamental:
  ▶ CONSLCK
  ▶ CONSCHECK
  ▶ CONSENFOLP, CONSENFOPS

▶ Additional:
  ▶ CONSSEPALP, CONSSEPASOL

▶ CONSTRANS, CONSDELETE

Further Ingredients

▶ Private data
▶ Interface methods
▶ Properties
**Exercise: TSP**

### Callback Methods

#### Fundamental:
- CONSLOCK
- CONSCHECK
- CONSENFOLP, CONSENFOPS

#### Additional:
- CONSSEPALP, CONSSEPASOL
- CONSTRANS, CONSDELETE

### Further Ingredients

- Private data
- Interface methods
- Properties
### Separation Problem

Given complete graph $G = (V, E)$ and $x^* \in [0, 1]^{|E|}$.

- Decide whether $x^*$ satisfies all **subtour elimination constraints**.
- If not, find violated subtour elimination constraint.
Separation Problem

Given complete graph $G = (V, E)$ and $x^* \in [0, 1]^{|E|}$.

- Decide whether $x^*$ satisfies all subtour elimination constraints.
- If not, find violated subtour elimination constraint.

Trivial observation:

- Consider $G = (V, E)$ with edge capacities $x^*_e$.
- $x^*$ violates at least one subtour elimination constraint
  $\Leftrightarrow \exists$ cut $\delta(S)$ with capacity $x^*(\delta(S)) < 2$. 
Separation Problem

Given complete graph $G = (V, E)$ and $x^* \in [0, 1]^{|E|}$.

- Decide whether $x^*$ satisfies all subtour elimination constraints.
- If not, find violated subtour elimination constraint.

Idea of separation algorithm:

- $\forall s, t \in V$: Find $(s, t)$-cut $\delta(S)$ of minimum capacity
- If all cut capacities $\geq 2$, all subtour elimination constraints satisfied

$\rightarrow \binom{|V|}{2}$ times MaxFlow-MinCut algo
Separation Problem

Given complete graph \( G = (V, E) \) and \( x^* \in [0, 1]^{|E|} \).

- Decide whether \( x^* \) satisfies all **subtour elimination constraints**.
- If not, find violated subtour elimination constraint.

Idea of separation algorithm:

- \( \forall s, t \in V \): Find \((s, t)\)-cut \( \delta(S) \) of minimum capacity
- If all cut capacities \( \geq 2 \), all subtour elimination constraints satisfied

\[ |V| - 1 \text{ times MaxFlow-MinCut algo within Gomory-Hu-Algorithm} \]
Result of Gomory-Hu-Algorithm

Gomory-Hu-Tree $T$

For all $s, t \in V$:

- capacity of minimum $(s, t)$-cut in $G$:
  minimum label $f_e$ of all edges $e$ in unique $(s, t)$-path in $T$

- minimum $(s, t)$-cut in $G$:
  bipartition of $V$ obtained by deleting this edge from $T$
Result of Gomory-Hu-Algorithm

Gomory-Hu-Tree $T$

For all $s, t \in V$:

- capacity of minimum $(s, t)$-cut in $G$:
  - minimum label $f_e$ of all edges $e$ in unique $(s, t)$-path in $T$
- minimum $(s, t)$-cut in $G$:
  - bipartition of $V$ obtained by deleting this edge from $T$

```
ghc_tree() returns all minimum $(s, t)$-cuts with capacity $\leq 2 - \text{minviol}$
```
Constraint Integer Programming

Primal Heuristics for Mixed Integer Programs

Implementation of Cutting Plane Separators for Mixed Integer Programs

http://scip.zib.de
Doxygen documentation, HowTos, FAQ

source code
scip.h, pub_*.h, type_*.h