Exercise: Linear ordering constraint handler

C. Puchert
Original version by T. Achterberg, T. Berthold, M. Pfetsch, K. Wolter

A *tournament* is a digraph that for each pair of nodes i, j with $i \neq j$ contains exactly one of the arcs (i, j) and (j, i). Consider a complete digraph $D = (V, A)$ on n nodes (containing arcs (i, j) and (j, i) for each pair of nodes i, j) with arc weights c_{ij} for all $(i, j) \in A$. The task of the *linear ordering problem* is to find an *acyclic tournament* (V, T) in D such that

$$\sum_{(i,j) \in T} c_{ij}$$

is maximized. Note that any acyclic tournament in D induces a *linear ordering* of V (order the nodes of V by non-increasing value of $|\delta^-_T(v)|$, $v \in V$.

The linear ordering problem can be formulated as the integer program

$$\begin{aligned} \max \quad & \sum_{(i,j) \in A} c_{ij} x_{ij} & & \\ \text{s.t.} \quad & x_{ij} + x_{ji} = 1 & i, j \in V,\ i \neq j \quad & (1) \\ & x_{ij} + x_{jk} + x_{ki} \leq 2 & i, j, k \in V,\ i \neq j \neq k \quad & (2) \\ & x_{ij} \in \{0, 1\} & (i, j) \in A. & \end{aligned}$$

Here, x_{ij} is equal to 1 if the arc (i, j) is contained in T and equal to 0 otherwise. Furthermore, the inequalities of type (1) ensure that (V, T) is a tournament in D and the inequalities of type (2) are needed to eliminate dicycles.

The linear ordering problem can also be stated as the constraint integer program (CIP)

$$\begin{aligned} \max \quad & \sum_{(i,j) \in A} c_{ij} x_{ij} & \\ \text{s.t.} \quad & \mathrm{LO}(D, x) & \\ & x_{ij} \in \{0, 1\} & (i, j) \in A, \end{aligned}$$

where the *linear ordering constraint* is defined as

$$\mathrm{LO}(D, x) :\Longleftrightarrow x \in \{0, 1\}^{|A|} \text{ represents an acyclic tournament in } D.$$

In order to solve an instance of the linear ordering problem with SCIP, linear ordering constraints must be supported by a constraint handler. For a given solution $x \in \{0, 1\}^{|A|}$, it has to check whether x represents an acyclic tournament in D, i. e., whether x satisfies all constraints of type (1) and (2). To improve the performance of the solving process of SCIP, the constraint handler may provide additional information about its constraints to the framework: for example, a linear relaxation that strenghtens the LP relaxation of the CIP. This relaxation should consist of constraints of type (1) and (2) and can be generated in advance and on the fly (cutting plane method), respectively.

Implement a constraint handler that supports linear ordering constraints. It should contain algorithms for feasibility checks, provide linear relaxations, and generate cutting planes.

In the doxygen documentation of SCIP, you will find the entry "How to add constraint handlers" which explains all steps of implementing a constraint handler in detail. Since not all of these steps are needed for this exercise, the following instruction will guide you through this documentation. All callback methods of a constraint handler are defined in `scip/src/scip/type_cons.h`. In particular, you will find the input data and possible return values of the callback methods there.

Getting Started

(a) Extract the linear ordering project `LOP.tgz`. Amongst other files, it contains:

- `src/cmain.c` Main file which initializes SCIP, includes the default plugins of SCIP and the linear ordering plugins, and starts the SCIP interactive shell.
- `CMakeLists.txt` `cmake` build file for the linear ordering project.
- `data/*.lop` Some linear ordering instances which you can use to test your code.
- `src/reader_lop.c` A file reader plugin which reads a linear ordering instance in `*.lop` format and creates the CIP model.
- `src/cons_lop.c` The (yet empty) constraint handler which you have to implement.

(b) Open the source file `src/cons_lop.c` and the header file `src/cons_lop.h` with a text editor and replace all occurrences of `xyz` and `Xyz` by `lop` and `Lop`, respectively.

(c) Adjust the properties `CONSHDLR_NAME` and `CONSHDLR_DESC`.

Defining the constraint data

(a) Define two fields in `struct SCIP_ConsData`: one for the number of elements (vertices) in the set V (`int nelems`) and one for the variables in the linear ordering constraint (`SCIP_VAR*** vars`, a quadratic matrix of `SCIP_VAR` pointers).

Implementing the interface methods and two additional callback methods

(a) Implement the interface method `SCIPcreateConsLop()` (to be found at the bottom of `cons_lop.c`).

It should allocate the memory for the constraint data by calling

```
SCIP_CALL( SCIPallocBlockMemory(scip, &consdata) );
```

and fill the fields of the constraint data.

The necessary problem information is passed by the file reader which calls this method. Therefore, you need to modify the input of this method and of `SCIPcreateConsBasicLop()`. Change them such that they receive `int nelems` and `SCIP_VAR*** vars` as problem information.

(b) Implement the `CONSDELETE` callback. For the given linear ordering constraint, it should free all of the memory that has been allocated in `SCIPcreateConsLop()`.

(c) Implement the `CONSTRANS` callback. It is needed since SCIP maintains two problem instances, the original problem and the transformed problem.

Hints:

- In `SCIPcreateConsLop()`, use `SCIPallocBlockMemoryArray()` to allocate memory for arrays.
- Use the methods `SCIPfreeBlockMemory()` and `SCIPfreeBlockMemoryArray()` if you want to free memory which has been allocated by calling `SCIPallocBlockMemory()` and `SCIPallocBlockMemoryArray()`, respectively.
- In order to get the constraint data `consdata` of a constraint `cons`, you can use:

```
SCIP_CONSDATA* consdata;
consdata = SCIPconsGetData(cons);
```

- When you copy a constraint to the transformed problem, the data of the new constraint must also hold the transformed variables. To get the transformed variable `transvar` for an original variable `var`, use

```
SCIP_CALL( SCIPgetTransformedVar(scip, var, &transvar) );
```

Implementing the fundamental callback methods

(a) Implement the `CONSCHECK` callback. It should loop through all linear ordering constraints given in the array `conss`. For each of these constraints, it should check whether the given solution satisfies all corresponding constraints of type (1) and of type (2). Note that usually there is only one linear ordering constraint, but it might happen that several constraints of this type are present, e. g. if more than one linear orderings should be coupled together.

(b) Implement the `CONSENFOLP` and `CONSENFOPS` callbacks. They should be similar to the `CONSCHECK`, but `CONSENFOLP` should try to resolve an infeasibility by adding a constraints of type (1) and (2) as cutting planes to the LP relaxation of the CIP.

(c) Adjust the properties `CONSHDLR_CHECKPRIORITY` and `CONSHDLR_ENFOPRIORITY`.

(d) Implement the `CONSLOCK` callback.

Hints:

- To compare `SCIP_Real`s, use `SCIPisFeasLE()` rather than `<=`, etc.
- By setting `CONSHDLR_CHECKPRIORITY` and `CONSHDLR_ENFOPRIORITY` to negative values, you can ensure that the feasibility check methods are only called for solutions that are already integral.

- The following lines of code create an LP row corresponding to the cut $1x + 2y \leq 3$ and add it to the separation storage:

```
SCIP_ROW* row;
SCIP_Bool* cutoff;
char rowname[SCIP_MAXSTRLEN];
(void) SCIPsnprintf(rowname, SCIP_MAXSTRLEN, "cut_%d", SCIPgetNCuts(scip));

SCIP_CALL( SCIPcreateEmptyRowCons(scip, &row, conshdlr, rowname,
     -SCIPinfinity(scip), 3.0, FALSE, FALSE, TRUE) );
SCIP_CALL( SCIPcacheRowExtensions(scip, row) );
SCIP_CALL( SCIPaddVarToRow(scip, row, varx, 1.0) );
SCIP_CALL( SCIPaddVarToRow(scip, row, vary, 2.0) );
SCIP_CALL( SCIPflushRowExtensions(scip, row) );
SCIP_CALL( SCIPaddRow(scip, row, TRUE, &cutoff) );
SCIP_CALL( SCIPreleaseRow(scip, &row) );
```

  where `SCIP_Bool cutoff` becomes `TRUE` if adding the row has lead to an infeasible LP.

Intermediate test

(a) Compile your project as follows:

```
mkdir build
cd build
cmake .. [-DSCIP_DIR=/non/systemwide/installation/path/of/scip]
make
```

(b) Test it on the provided linear ordering instances. Since you have implemented all fundamental callback methods, the resulting code should be correct and find an optimal solution to a given problem instance. However, it might be very slow because the additional features like linear relaxation and cut separation are missing.

Implementing the additional callback methods

(a) Implement the `CONSINITLP` callback. It should add constraints of type (1) as linear relaxations of linear ordering constraints to the initial LP relaxation of the CIP.

(b) Implement the `CONSSEPALP` and the `CONSSEPASOL` callback. They should separate inequalities of type (2).

(c) Adjust the properties of `CONSHDLR_SEPAPRIORITY`, `CONSHDLR_SEPAFREQ`, and that of `CONSHDLR_DELAYSEPA`.

Hints:

- For creating an LP row and adding it to the initial LP relaxation, see the similar hint for implementing the fundamental callback methods. You can use the same lines of code, but you have to use `FALSE` as the last argument of the method `SCIPcreateEmptyRowCons()`.