Technische Universität Berlin
Institut für Mathematik

Wintersemester 11/12

Computational Integer Programming

PD Dr. Ralf Borndörfer
Dr. Thorsten Koch

Exercise sheet 7

Deadline: Thu, 15 Dez. 2011, by email to `borndoerfer@zib.de`

The symmetric *Traveling Salesman Problem* (TSP) on a complete graph $G = (V, E)$ with edge lengths c_e can be formulated as the following integer program, which uses a binary variable x_e for each edge $e \in E$.

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e \, x_e & & \\ s.t. \quad & \sum_{e \in \delta(v)} x_e = 2 & \forall \, v \in V & \quad (1) \\ & \sum_{e \in \delta(S)} x_e \geq 2 & \forall \, S \subset V, S \neq \varnothing & \quad (2) \\ & x_e \in \{0,1\} & \forall \, e \in E. & \end{aligned}$$

Here, x_e is equal to 1 if the edge $e \in E$ is contained in a tour and equal to 0 otherwise. The *degree constraints* (1) ensure that each node is incident to exactly two edges and the *subtour elimination inequalities* (2) rule out "small cycles".

The symmetric TSP can also be stated as the constraint integer program (CIP)

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e \, x_e & & \\ s.t. \quad & \sum_{e \in \delta(v)} x_e = 2 & \forall \, v \in V & \\ & \text{nosubtour}(G, x) & & \quad (3) \\ & x_e \in \{0,1\} & \forall \, e \in E. & \end{aligned}$$

The *nosubtour constraint* (3) is defined as

$$\text{nosubtour}(G, x) \;\Leftrightarrow\; \nexists C \subseteq \{e \in E \mid x_e = 1\} : C \text{ is a cycle of length } |C| < |V|.$$

This constraint must be supported by a constraint handler, which – given an integral solution $x \in \{0,1\}^E$ – **has to check** whether the corresponding set of edges contains a subtour C (in `scip_check`, `scip_enfolp`, and `scip_enfops`).

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 strengthens the LP relaxation of the CIP. The linear relaxation of the

nosubtour constraint consists of exponentially many subtour elimination inequalities (2) which **can be separated** and added on demand to the LP relaxation (in `scip_sepalp` and `scip_sepasol`).

Implement a constraint handler that supports nosubtour constraints. It should contain algorithms for feasibility checks and cutting plane separation.

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.

Getting started

(a) You must install the ZIBOptSuite 1.2.0 from `zibopt.zib.de` in order to do this exercise. Download `ziboptsuite-1.2.0.tgz` from the webpage and execute the following commands:

- `tar xvf ziboptsuite-1.2.0.tgz`
- `cd ziboptsuite-1.2.0`
- `make`
- `make test`

(b) Extract the TSP project `COatWork-TSP.tgz` (file `tar xzf COatWork-TSP.tgz` posted on the webpage). Amongst others, it contains:

`src/cppmain.cpp` Main file, which initializes SCIP, includes the default plugins of SCIP, includes the user defined plugins, e.g., a reader for TSP instances in TSP format and the constraint handler for nosubtour constraints to be implemented, and invokes the solving process of SCIP.

`src/ProbDataTSP.cpp` **and** `src/ReaderTSP.cpp` Provide methods for reading a TSP instance from a file and for creating and storing the corresponding CIP model.

`src/ConsNosubtour.cpp` Constraint handler to be implemented/completed. Note, that the following steps of the "How to add constraint handlers" instruction have already been implemented: Properties of a Constraint Handler, Constraint Data and Constraint Handler Data, Interface Methods. Furthermore, the fundamental callback methods `scip_lock` (C++ for `CONSLOCK`) and the additional callback methods `scip_delete` (C++ for `CONSDELETE`) and `scip_trans` (C++ for `CONSTRANS`) have already been implemented.

`src/GomoryHuTree.cpp` Gomory-Hu-tree algorithm. See also see next point.

`src/GomoryHuTree.h` Structure for a complete directed graph. The `GRAPH` consists of `GRAPHNODE`s, each with a unique number `id` $\in \{0, \ldots, |V| - 1\}$ and a `GRAPHEDGE` pointer to `first_edge`, which is the first element of an adjacency list. Each `GRAPHEDGE` stores its target node `adjac`, the next element `next` in the adjacency list of its start node, its reverse edge `back`, its corresponding problem variable `var`, and weights `cap` and `rcap` for the Gomory-Hu algorithm.

`Makefile` Makefile for the TSP project. In particular, it links the necessary files of SCIP and the constraint handler to be implemented to the TSP project.

`tspdata/*.tsp` Some TSP instances in TSP format.

(c) In the folder `COatWork-TSP/lib`, create a softlink to your SCIP directory, e.g. `ln -s /home/coatwork/software/ziboptsuite-1.2.0/scip-1.2.0 scip`

(d) In the folder `COatWork-TSP`, try to compile the provided code: `make depend` and `make`.

(e) Open the file `COatWork-TSP/src/ConsNosubtour.cpp` and implement all missing methods as indicated below. For each section, please read all given instructions, before you begin the implementation.

Implementing fundamental callback methods: Feasibility check

(a) The `scip_check`, `scip_enfolp`, and `scip_enfops` methods all call the local method `consCheck()`, which currently only loops through all nosubtour constraints in the array `conss`. For each of these constraints, you should implement an algorithm that checks whether the given solution `sol` satisfies the nosubtour constraint (3). Note that in the TSP, there is only one nosubtour constraint, but in general, several constraints of this type could be present.

(b) In `ConsNosubtour.h`, the parameters `scip_checkpriority_` and `scip_enfopriority_` (C++ for `CONSHDLR_CHECKPRIORITY` and `CONSHDLR_ENFOPRIORITY`) have been set such that the nosubtour constraint handler is called after the integrality and the linear constraint handler. This ensures that all feasibility checks are only called for solutions that are already integral and satisfy the degree

constraints (1).

(c) For example, a feasibility check could follow a path of edges whose corresponding problem variables have an LP value (which you get by `SCIPgetSolVal()`) numerically equal to one (to be checked by calling `SCIPisFeasEQ()`). Since `consCheck()` only gets called, when the degree constraints (1) are fulfilled, it can conclude feasibility, iff this path contains all nodes of the graph G.

(d) The constraint data `SCIP_CONSDATA` of a nosubtour constraint contains a pointer to the `GRAPH` on which it is defined.

(e) For detailed information on SCIP methods, see the "List of callable functions" in the doxygen documentation on the SCIP homepage `http://scip.zib.de`.

Intermediate test

(a) Compile your project and test it on the provided TSP instances, e.g., `ulysses16.tsp` (optimal value 6859). Since all fundamental callbacks are implemented now, the resulting code should be correct and find an optimal solution to a given problem instance. However, it might be very slow, because additional features like cutting plane separation are missing.

Implementing additional callback methods: Cutting plane separation

(a) The methods `scip_sepalp` and `scip_sepasol` also call a common method `sepaCons()`, which you should implement. It is supposed to separate subtour elimination inequalities (2) that are violated by the given LP solution and the given arbitrary primal solution passed by `scip_sepalp` and `scip_sepasol`, respectively.

(b) Use `SCIPallocBufferArray()` to allocate memory for the arrays which you pass to `ghc_tree()`, see next point. Use `SCIPfreeBufferArray()` to free the allocated memory afterwards. Note that the `cuts` array is two-dimensional and you have to allocate all entries.

(c) The separation problem can be solved by the `ghc_tree()` algorithm, which is defined in `GomoryHuTree.cpp`. It should operate on the `GRAPH` stored in the constraint's data. Before calling this method, you have to install the LP values (which you get by `SCIPgetSolVal()`) of each edge variable to the capacities `cap` and `rcap` of each arc and its backwards arc in the `GRAPH`. The Gomory-Hu algorithm fills the array `cuts`. A cut $\delta(S)$ is defined by a bipartition S and $V \setminus S$ of the nodes of the graph. Each `cuts[i]` corresponds to the incidence vector of S. For each such cut $\delta(S)$, you should construct an inequality (LP row) of type (2), and add it to the separation storage of SCIP.

(d) As an example, the following lines of code create an LP row corresponding to the cutting plane $1x + 2y \leq 3$ and add it to the separation storage:

```
SCIP_ROW *row;
char rowname[SCIP_MAXSTRLEN];
(void) SCIPsnprintf(rowname, SCIP_MAXSTRLEN, "cut_%d", ncuts);

SCIP_CALL( SCIPcreateEmptyRow(scip, &row, 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( SCIPaddCut(scip, NULL, row, FALSE) );
SCIP_CALL( SCIPreleaseRow(scip, &row));
```

(e) By the way: In `ConsNosubtour.h`, `scip_sepapriority_` and `scip_sepafreq_` (C++ for parameters `CONSHDLR_SEPAPRIORITY` and `CONSHDLR_SEPAFREQ`) have been set such that your separation algorithm will be performed at each branch-and-bound node and will always be called as the first cutting plane routine.

Final test

(a) Compile your project and test it on the provided TSP instances.

Good luck!