Exercise: Subtour elimination Constraint Handler for the TSP

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 $e \in E$.

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

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* (**??**) ensure that each node is incident to exactly two edges and the *subtour elimination inequalities* (**??**) rule out "small cycles".

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

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

The *nosubtour constraint* (**??**) 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 `CONSCHECK`, `CONSENFOLP`, and `CONSENFOPS`).

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 (**??**) which **can be separated** and added on demand to the LP relaxation (in `CONSSEPALP` and `CONSSEPASOL`).

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 instructions will guide you through this documentation.

Getting started

(a) Download and extract the TSP project `COatWork-TSP.tgz` from the course web page. (`tar xzf COatWork-TSP.tgz`). Amongst others, it contains:

  `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.

  `src/cmain.c` 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/probdata_tsp.c` **and** `src/reader_tsp.c` providing methods for reading a TSP instance from a file and for creating and storing the corresponding CIP model.

  `src/GomoryHuTree.c` 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` $\in \{0, \dots, |V|-1\}$ 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.

  `src/cons_nosubtour.c` constraint handler to be implemented/completed. Note, that the following steps of the "How to add constraint handlers" instructions have already been implemented: Properties of a Constraint Handler, Constraint Data and Constraint Handler Data, Interface Methods. Furthermore, the fundamental callback methods `CONSLOCK` and the additional callback methods `CONSDELETE` and `CONSTRANS` have already been implemented.

(b) In the folder `COatWork-TSP/lib`, create a softlink to your SCIP directory, e.g.

```
> ln -s /home/coatwork/projects/scipoptsuite-3.2.0/scip-3.2.0 scip
```

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

(d) Open the file `COatWork-TSP/src/cons_nosubtour.c` 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 `CONSCHECK`, `CONSENFOLP`, and `CONSENFOPS` methods all call the local method `findSubtour()`. You should implement an algorithm that checks whether the given solution `sol` satisfies the nosubtour constraint (**??**). Currently, this method returns `TRUE` meaning that there is a subtour, ie. the solution is infeasible.

(b) At the beginning of `cons_nosubtour.c`, the parameters `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 (**??**).

(c) For example, a feasibility check could construct a cycle by following 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 `findSubtour()` is only called when the degree constraints (**??**) are fulfilled, it can conclude that no subtour exists (ie. the solution is feasible), if and only if, the constructed cycle contains all nodes of the graph G.

(d) The constraint data `SCIP_CONSDATA` of a nosubtour constraint contains a pointer to the `GRAPH` G on which the constraint 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 `CONSSEPALP` and `CONSSEPASOL` also call a common method `sepaSubtour()`, which you should implement. It is supposed to separate subtour elimination inequalities (**??**) that are violated by the given LP solution and the given arbitrary primal solution passed by `CONSSEPALP` and `CONSSEPASOL`, 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.c`. 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`. Recall that 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, containing an entry of `1` (or `TRUE`, since the array should be of type `SCIP_Bool`) for every node $j \in S$ and `0`, otherwise. For each such violated cut `cuts[i]`, you should construct an inequality (LP row) of type (**??**), and add it to the separation storage of SCIP.

(d) As an example, suppose that a given constraint, whose constraint handler is `conshdlr`, generates the cutting plane $1x + 2y \leq 3$. 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];
SCIP_Bool infeasible;

(void) SCIPsnprintf(rowname, SCIP_MAXSTRLEN, "cut_%d", ncuts);
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( SCIPaddCut(scip, NULL, row, FALSE, &infeasible) );

if( infeasible )
     *result = SCIP_CUTOFF;
else
     *result = SCIP_SEPARATED;

SCIP_CALL( SCIPreleaseRow(scip, &row));
```

(e) By the way: at the beginning of `cons_nosubtour.c`, `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!

BONUS: Primal Heuristic for TSP

Implement a primal heuristic for the symmetric traveling salesman problem (TSP) on a complete graph $G = (V, E)$. This heuristic should be a greedy start heuristic, which is based on the current LP solution. It starts with an arbitrary node $s \in V$ and iteratively constructs an $[s, u]$-path T, as long as there are nodes left not contained in T. It then closes the path to a tour. In each iteration, the $[s, u]$-path T is extended to an $[s, v]$-path where node $v \in V \setminus V(T)$ is selected in a greedy fashion such that

$$x^*_{uv} = \max \{x^*_{uw} : w \in V \setminus V(T),\ x_{uw} \text{ not fixed to zero}\}.$$

Here, x^*_{uv} denotes the value of the variable x_{uv} in the current LP solution. A binary variable is fixed to zero if its global upper bound has changed to zero.

In the doxygen documentation of SCIP, you will find the entry "How to add primal heuristics", which explains all steps of implementing a primal heuristic in detail. Since not all of these steps are needed for this exercise, again, the following instructions will guide you through this documentation.

Getting started

(a) Go to the directory `COatWork-TSP/src` and copy the template files `heur_xyz.c` and `heur_xyz.h` to files named `heur_lpgreedy.c` and `heur_lpgreedy.h`, respectively. Open the new files with a text editor and replace all occurrences of `xyz`, `Xyz`, and `XYZ` by `lpgreedy`, `Lpgreedy`, and `LPGREEDY`, respectively.

(b) Adjust the name, description, and display character parameters at the beginning of `heur_lpgreedy.c`.

(c) Include your header file `heur_lpgreedy.h` into the main file `cmain.c`. Then, add `SCIPincludeHeurLpgreedy(scip)` to the `runSCIP()` method of `cmain.c` and include the line "`heur_lpgreedy.o`" into the `MAINOBJ` list of the Makefile. Compile and test.

(d) `[src/heur_lpgreedy.c:157] ERROR: method of lpgreedy primal heuristic not implemented yet`
in the SCIP output means that everything works (for now).

(e) To have access to all necessary methods, you need to include the headers`GomoryHuTree.h` and `probdata_tsp.h`.

Setting up the primal heuristic data

(a) Fill the `struct SCIP_HeurData` with one member for the heuristic working solution (`SCIP_SOL* sol`).

(b) Allocate memory for the heuristic data in `SCIPincludeHeurLpgreedy()` with `SCIP_CALL( SCIPallocMemory(scip, &heurdata) )`

(c) All data allocated in `SCIPincludeHeurLpgreedy()` should be freed in `HEURFREE`. By default, this callback is commented out. Your task here is to uncomment it and free `heurdata` using `SCIPfreeMemory(&heurdata)`.
Note that the callback method receives a pointer to the heuristic `SCIP_HEUR* heur` from which you need to obtain its heuristic data pointer by `SCIPheurGetData(heur)`.

Implementing two additional callback methods

(a) Implement the `HEURINIT` method. It should initialize the member of the heuristic data. Therefore, you have to create the solution `sol`, using `SCIPcreateSol()`.

(b) Implement the `HEUREXIT` method. It should free all of the memory that has been allocated in the `HEURINIT` method, hence call `SCIPfreeSol()`.

Implementing the fundamental callback method

(a) Implement the `HEUREXEC` callback. It should realize the actual heuristic outlined above. Here are some implementation hints:

- The heuristic should be called only when the current LP has been solved to optimality. You can check this with

  ```
  if( SCIPgetLPSolstat(scip) != SCIP_LPSOLSTAT_OPTIMAL )
  ```

- The graph pointer can be accessed via the problem data structure, which you obtain by `SCIP_PROBDATA* probdata = SCIPgetProbData(scip);`
  Then access the graph from the `probdata` (see `probdata_tsp.h`).
- At the beginning you need to clear the solution to contain 0 for every variable. For this use:

  ```
  SCIP_CALL( SCIPclearSol(scip, heurdata->sol) );
  ```

- For accessing the LP solution value of variable `var` use:

  ```
  SCIPgetSolVal(scip, NULL, var);
  ```

- To set values to variables in the working solution, use `SCIPsetSolVal()`. All variables for which no value is set explicitly, are treated as zero. Once you constructed the greedy solution, you can pass it to SCIP by `SCIPtrySol()`.
- `SCIPvarGetUbGlobal()` (defined in `pub_var.h`) returns the global upper bound of a variable, which you need for checking that a variable is not globally fixed to 0.
- Use the methods `SCIPallocBufferArray()` and `SCIPfreeBufferArray()` for allocating and freeing memory for arrays in the `HEUREXEC` method, e.g., for a `SCIP_Bool` array that stores (at position `node->id`) whether `node` is already contained in $V(T)$.
- The `result` pointer should be set to `SCIP_DIDNOTFIND` or `SCIP_FOUNDSOL`.

(b) You can adjust the priority, frequency, frequency offset, maximal depth, and timing parameters at the beginning of `heur_lpgreedy.c` to change the performance.

Final test

(a) Compile your project and test it on the provided TSP instances. You should see your chosen display character as first character in an output line, whenever your heuristic found a solution.