Technische Universität Berlin
WiSe 2012

Prof. Andreas Bley

Computational mixed-integer programming

4th set of programming exercises: Steiner Tree Problem – A cutting plane algorithm

In this exercise we will revisit the Steiner tree problem. In particular, we will work with the directed cut formulation:

$$\begin{aligned}
&\min c^T x && && \text{(DCUT)}\\
&\text{s.t. } x(\delta^+(S)) \geq 1 && \forall\, S \subset V:\ r \in S,\, S \cap (T \setminus \{r\}) \neq (T \setminus \{r\}) &&\\
&\quad\ x_{(i,j)} \geq 0 && \forall\, (i,j) \in A &&\\
&\quad\ \mathbf{x} \in \mathbb{Z}^A && &&
\end{aligned}$$

We already used ZIMPL to generate a model containing *all* the cut constraints and then solved this model. In this exercise we will improve this by iteratively adding only those cut constraints that are actually needed. The basic idea behind this approach is simple:
First, we do not add any cut constraints to our model and solve the resulting trivial LP relaxation. Then we check if, for the resulting LP solution, there exists a node set S with $r \in S$, $S \cap T' \neq T'$ for witch the directed Steiner cut constraint is violated. If this is the case, we add the Steiner cut constraint corresponding to S to the model and solve the new LP. We continue with these steps, until no such set S can be found. Then the optimal fractional solution is found and we proceed by enforcing integrality via branch-and-bound. However, at each node of the branch-and-bound tree the corresponding LP relaxation needs to be solved, so we again have to check for new violated cuts.

Getting started

a) Extract the package `exercise04.tgz`.

b) In the `Steiner` folder you will find the following files:

`src/Digraph.{h,cpp}` These files contain the digraph data structure that is used to store the underlying bidirected graph.

`src/STPData.{h,cpp}` These files contain the complete Steiner Tree Problem data structure, including the digraph, the terminal set, the arc lengths, functions to associate SCIP variables with the arcs of the digraph, and the function to read the problem data.

`src/MaxFlow.{h,cpp}` contains the interface of the max flow / min cut computation function, which is used to separate Steiner cuts. You need to implement this function in `src/MaxFlow.cpp` the first part of the exercise.

`src/main.cpp` is the main file which reads the data files from the command line, initializes SCIP, builds the above formulation and solves it

`src/SCutConstraintHandler.{h,cpp}` These files contain the handler class for the Steiner cut constraints. This class contains functions to check the feasibility of an integer candidate solution vector and to separate violated Steiner cut inequalities.

`scipstp.set` A SCIP settings file that is used to disable some preprocessing routines. The standard MIP preprocessing cannot be used for this problem since the model does not yet contain all constraints.

`data/*.stp` A folder with (relatively easy) test instances of different sizes.

`challenge/*.stp` A folder with test instances that are a little bit harder to solve ...

c) As usual, you can can compile the code with `make` in the `Steiner` folder. The executable file created is called `bin/stp`.

1. Implement a Max Flow Algorithm

Read the descriptions in the files `src/MaxFlow.{h,cpp}`. They tell you how your max flow implementation should behave to be useful as a separation subroutine.

Then recall your favorite max s-t flow (or min s-t cut) algorithm and implement it.

Keep in mind that, after your function returns, the node-indexed vector `predecessors` will be used to identify the minimum cut: if `predecessor[v]` ≥ 0, then `predecessor[v]` is the edge via which node v can be reached from the source on an flow-augmenting path. Otherwise, `predecessor[v]` must be equal to -1, indicating that v cannot be reached from the source on a flow-augmenting path. The corresponding directed s-t cut thus is given by the set of nodes with `predecessor[v]` ≥ 0.

To test your implementation, compute the max flows / min cuts between the first terminal (i.e., the root) and all other terminals for some of the given test instances, using the arc lengths as arc capacities.

Constraint Handler

Since SCIP is a solver for *Constraint Integer Programs* we can create our own special constraint type that handles the automatic generation of violated cut constraints. To implement such a constraint we have to write our own constraint handler.
In C++ a constraint handler can be implemented as a class that inherits form the base class `ObjConshdlr` and implements at least the following virtual methods:

`scip_check` gets a primal solution candidate in a `SCIP_SOL*` data structure and has to check this solution for global feasibility. In our case, we have to check, whether the solution is a Steiner Tree or not.

`scip_enfolp` Like the `scip_check` method, the `scip_enfolp` method should check the solution (in this case, the LP solution) for feasibility.

`scip_lock` tells SCIP, which variables are existing in the given constraint, and in which way modifications of these variables may affect the feasibility of the constraint. This is for example needed for the primal rounding heuristics.

`scip_sepalp` In this method we have to generate cutting planes for the constraints of the constraint handler in order to separate the current LP solution. The method is called in the LP solution loop, which means that a valid LP solution exists.

`scip_sepasol` is very similar to `scip_sepalp`. However, we now generate cutting planes for the constraints of the constraint handler in order to separate the given *primal* solution. So, here no LP solution is available.

The files `src/SCutConstraintHandler.{h,cpp}` already contain an implementation of such a constraint handler the only things left to do are the following functions:

- `findDirectedSteinerCut(...)`
- `sepaDirectedSteinerCut(...)`

2. Implement the Feasibility Check

To verify if the arcs in the given solution form a feasible Steiner arborescence, we check whether we can find a cut S in the graph with $r \in S$ and $(V \setminus S) \cap T' \neq \emptyset$ so that the capacity of S is less than 1. If that is the case, we know that the formulation (DCUT) is violated and that our solution must be infeasible.
Edit the function `findDirectedSteinerCut` and use the given function `max_flow` to check whether such a cut exists. If there is a cut with capacity less than 1 return `true` and `false` otherwise.

3. Implement the Separation

Now edit the function `sepaDirectedSteinerCut` to check if the (fractional) capacities satisfy all directed Steiner cuts constraints. If that is not the case then add the corresponding violated cut to the model. To add a cut in SCIP you first have to create the inequality row using `SCIPcreateEmptyRow` and then add this row as a cut (`SCIPaddCut`).
Warning: the given `max_flow` function only takes integers for the arc capacities. To transform the variable solution values of type `SCIP_Real` into integers, scale them first with the factor `SCALE` and then round up.
Think carefully about how you can use the results from `max_flow` to identify the arcs that are on a violated cut.

4. Test and Improve your Implementation

a) Now try to solve all the instances in `data` with your implementation. Probably, this is not going to work for the bigger instances!

b) Try to improve your implementation so that you can solve all the instances in `data`:

- Perform some test, to identify the quality of the cuts you find. You should avoid generating the same or very similar cuts more than once. Try to generate different cuts for the different terminals.
- Try to find more than one cut for every $t \in T'$ in one single call of the separation function. You can do this by changing arc capacities and then calling `max_flow` again. Test some different approaches on how to generate these additional cuts.
  Try to change the arc capacities only in such a way that flow values calculated so far are still feasible. Then you don't have to restart the flow algorithm form zero every time. Finding a good limit on the number of cuts for every t is again up to you and will also need some testing.

Challenge*

So far, the instances were not too difficult to solve, but now we are going to attack the instances in the `challenge` folder!

> There will be a price (a box of chocolates or 20 bottles of beer) for the group whose implementation minimizes the sum of all three optimality gaps, running each instance for at most 3600 sec on my workstation (6 core AMD with 8GB RAM)!

You could for example try to further improve your cut generation or to find better primal bounds by implementing specialized primal heuristic or adapting the problem formulation so that the built-in

heuristics work more efficiently.
Implement anything else you can think of to improve your implementation. There is a lot more you could do…

5. Submit your code

This time, **each group must submit** their implementation (source code) as a tar, tgz or zip archive to `bley@math.tu-berlin.de` by **January 6, 2013** for verification and evaluation on the (easy) instances, independent of whether or not they want to participate in the Challenge. For those who want to participate in the challenge, January 6 is the submission deadline as well.

(Note that the 5th exercise will already start in December. You may use the extra time until January 6 to improve and tune your implementation.)