1 Overview: Solving the ATSP using SCIP

SCIP is a framework for *Constrained Integer Programs*, which are a generalization of MIPs. The generalization comes from the ability to deal with arbitrary constraints, provided some basic operations are provided for them.

In order to solve the ATSP problem, one could use the ATSP model based on (all) subtour elimination constraints. However, if one wants to employ dynamic generation of violated subtour elimination constraints (i. e., cutting planes methods), the "correct" way to represent the ATSP problem as a model to be solved by SCIP is

$$\begin{aligned}
\min \quad & \sum_{a \in A_n} c_a x_a && (1)\\
\text{s. t.} \quad & x(d^+(v)) = 1 \quad \forall v \in V && (2)\\
& x(d^-(v)) = 1 \quad \forall v \in V && (3)\\
& \mathrm{NoSubtour}(G, x) && (4)\\
& x \in \{0,1\}^{|A_n|} && (5)
\end{aligned}$$

where

$$\mathrm{NoSubtour}(G, x) :\Longleftrightarrow \text{ there exists only one cycle in the set } \{a \mid x_a = 1\}.$$

The constraint $\mathrm{NoSubtour}(G, x)$ is an example of a complex constraint. In order that SCIP can handle this type of constraint, we need to supply a *constraint handler*, which is one of the many types of plugins SCIP offers. The job of a constraint handler is to ensure that each overall feasible solution is feasible with respect to this kind of constraint.

In principle, there are many ways to establish the feasibility of the $\mathrm{NoSubtour}(G, x)$-constraint during the solution process of SCIP. The main purpose of these exercises is to get experience with using cutting planes, so we will use separation of subtour elimination constraints to ensure feasibility. In other words, any solution will only fulfill the $\mathrm{NoSubtour}(G, x)$-constraint if it does not violate any subtour elimination constraint. In this light, the above model is equivalent to that featuring all subtour elimination constraints.

Before we provide some more background on constraint handlers, here is an overview of the files in the exercise distribution:

`Problem_Data.{hh,cc}` These files define a data structure that stores additional problem data (a graph in our case). Moreover, they implement a *file reader plugin*, that reads our file format, sets up the graph and creates the model above in SCIP.

`Graph.{hh,cc}` These files implement the graph class to be used and some algorithms available for operating on them. For details, see Section 3.

`Subtour_Handler.hh` This file defines the interface of our constraint handler implementation for the $\mathrm{NoSubtour}(G, x)$-constraint.

`Subtour_Handler_framework.cc` This file implements some auxiliary framework stuff which is not important for the exercises.

`Subtour_Handler_stubs.cc` This file is the only file to be changed by you. It contains interfaces and comments for the methods to be implemented during this exercise.

`main.cc` This file contains the main function which sets up our constraint handler and starts SCIP.

2 The constraint handler concept of SCIP

The constraint handler concept of SCIP allows SCIP to deal with nearly arbitrary constraints without knowing anything specific about them. This flexibility is achieved by providing so-called callbacks which are methods with specified semantics that a constraint handler needs to provide.

We will implement the following callbacks (some more are implemented in the framework part):

`scip_check()` The purpose of this callback is to check an arbitrary solution for feasibility with respect to this type of constraints. It may either claim that the solution is feasible or infeasible.

`scip_enfolp()` The purpose of this callback is to guide the solution process towards final feasibility. It is called at the very end of processing a node in the branch and bound tree. If the solution is not yet feasible, the constraint handler should take an action to make progress towards feasibility. In our case, this means adding violated subtour elimination constraints, i. e., separation. Other possibilities are for instance branching or trying to cut the node off based on bound computations.

`scip_sepalp()` This callback is called if during node processing the current solution is still fractional. The constraint handler can then improve the LP relaxation by adding valid violated linear inequalities.

While the first two callbacks are needed for a correct implementation, implementing the `scip_sepalp()` callback is optional, but may improve performance.

The SCIP documentation of all callbacks was copied to the file `Subtour_Handler_stubs.cc` for reference.

3 The graph classes

3.1 Preliminaries

We provide two C++ classes representing graphs: `Weighted_Graph` and `Weighted_Digraph` for undirected and directed graphs, respectively. Both classes provide types `Node` and `Edge`, which describe a node or an edge. This type is given by the expression `Weighted_Graph::Node` or `Weighted_Digraph::Edge` and so on.

A value of these types is only valid for the graph instance it corresponds to. This is incovenient since we deal with more than one graph which need to be related in some way. For instance, the undirected graph corresponding to the current LP solution is a subgraph of the graph defining the ATSP instance and we need to express the correspondence between both node sets. To this end, we use values of type `ID_Type` to provide numerical IDs for each node and edge. So the node with ID 5 in the solution graph corresponds to the node with ID 5 in the original ATSP graph.

This is particularly useful when dealing with sets of nodes represented as instances of `Node_ID_Set`, which is just a set of node IDs. We can construct such a node set for the solution graph and –based on the node IDs– find the edge set in the original ATSP graph for generating subtour elimination constraints. The class `Node_ID_Set` supports the following operations.

`N.insert( ID_Type node_ID )`
inserts ID `node_ID` in node set `N`

`N.erase( ID_Type node_ID )`
deletes ID `node_ID` from node set `N`

`Node_ID_Set::iterator N.begin()`
returns an iterator pointing to the beginning of the node set

`Node_ID_Set::iterator N.end()`
returns an iterator pointing past the last element of the node set

`Node_ID_Set::iterator N.find( ID_Type node_ID )`
: returns an iterator to the element in the set and `N.end()` if `node_ID` is not in the set

Example: Using iterators The following C++ code shows how to deal with a `Node_ID_Set`, in particular how to use iterators for iterating through it.

```
Node_ID_Set S;

// S={1,...,10}.
for( ID_Type i = 1; i <= 10; ++i )
  S.insert( i );

// Remove 3, 7, 8.
S.erase( 3 );
S.erase( 7 );
S.erase( 8 );

// Print the set using iterators.
// ++it advances the iterator to the next element of the set
//  *it dereferences the iterator, ie returns the element the iterator points to
for( Node_ID_Set::iterator it = S.begin(); it != S.end(); ++it )
{
  std::cout << *it << std::endl;
}
```

3.2 Common operations of `Weighted_Graph` and `Weighted_Digraph`

This sections lists methods of the graph classes.

Modifying a graph

`Node add_node( const ID_Type node_ID )`
: adds a node with `node_ID` and returns corresponding `Node` instance for accessing this node (however, nodes are usually accessed using their IDs)

`void delete_node( const ID_Type node_ID )`
: deletes node with ID `node_ID` and all incident edges

`Edge add_edge( const ID_Type source_ID, const ID_Type target_ID, const double weight )`
: adds an edge between the nodes with IDs `source_ID` and `target_ID` with weight `weight`

`void set_weight( const Edge edge, const double new_weight )`
: set a new weight for the edge

Accessing nodes and edges and related information

`int nr_nodes()`

`int nr_edges()`
: number of nodes (edges) in the graph

`ID_Type max_node_ID()`

`ID_Type max_edge_ID()`
: Maximum node (edge) ID that has been in the graph. Useful if node or edge related information should be stored in a `std::vector`.

`Node node( const ID_Type node_ID )`
: access a node through its ID

`ID_Type node_ID( const Node node )`
: retrieve ID of a node

`std::pair< Node_Iterator, Node_Iterator > nodes()`

`std::pair< Edge_Iterator, Edge_Iterator > edges()`
: Support for iterating over nodes (edges). If `p` is the resulting `std::pair`, `p.first` is an iterator to the first node (edge) and `p.second` is an iterator pointing past the last node (edge).

`std::pair< In_Edge_Iterator, In_Edge_Iterator > in_edges( const Node node )`

`std::pair< Out_Edge_Iterator, Out_Edge_Iterator > out_edges( const Node node )` support
: for iterating over all edges incident to a node, semantics as above

`Node source( const Edge edge )`

`Node target( const Edge edge )`

`ID_Type source_ID( const Edge edge )`

`ID_Type target_ID( const Edge edge )`

`double weight( const Edge edge )`
: retrieve information related to an edge: source node, target node, ID of source node, ID of target node, edge weight

Example: Building a graph and iterating over edges incident to node 2 The following C++ code sets up a small graph on four vertices and prints all edges incident to the node with ID 2.

```
Weighted_Graph G;

// Setup G.
G.add_node( 1 );
G.add_node( 2 );
G.add_node( 3 );
G.add_node( 4 );

G.add_edge( 1, 2, 3.14 );
G.add_edge( 4, 2, 1.41 );
G.add_edge( 1, 3, 2.78 );

Weighted_Graph::Out_Edge_Iterator out_edge_i, out_edge_end;
for( boost::tie( out_edge_i, out_edge_end ) = G.out_edges( G.node( 2 ) );
     out_edge_i != out_edge_end; ++out_edge_i )
{
  std::cout << "(" << G.source_ID( *out_edge_i ) << ","
            << G.target_ID( *out_edge_i ) << ")" << std::endl;
}
```

In the `for`-loop, we employ the `boost::tie()`-function from the Boost library. It takes the iterator pair returned by `G.out_edges()` and assigns the first component to `out_edge_i` and the second to `out_edge_end`.

3.3 Operations for `Weighted_Graph`

Further methods of `Weighted_Graph`:

`std::pair< bool, Edge > find_edge( ID_Type source_ID, ID_Type target_ID )` Checks whether an edge between the two nodes is in the graph. If this is the case, the `first` component of the `std::pair` is true and the `second` component contains the corresponding `Edge` instance. If the edge is not present, `first` is false.

Further functions dealing with `Weighted_Graph`:

`size_t connected_components( const Weighted_Graph& graph, std::vector< int >& component )` Returns the number of connected components in `graph`. Moreover, the vector `component` stores for each node ID the number of the component it belongs to (i. e., all node IDs with the same number belong to the same connected component).

3.4 Operations for `Weighted_Digraph`

Further methods of `Weighted_Graph`:

`bool has_reverse_edge( const Edge e )`
checks whether there is also an edge in the other direction in the graph

`Edge reverse_edge( const Edge e )`
return edge in the other direction

Further functions dealing with `Weighted_Graph`:

`double min_cut( Weighted_Digraph& graph,`
`const ID_Type source_ID,`
`const ID_Type sink_ID,`
`std::vector< Color_Type >& node_color )` Computes a minimum cut separating nodes with IDs `source_ID` and `sink_ID`. The vector `node_color` encodes the two node sets corresponding to the cut by mapping each node ID to colors `BLACK` or `WHITE`.

4 Implementation tasks

All the implementation has to be done in the file `Subtour_Handler_stubs.cc` in the `src` directory. The program can be compiled by just doing `make`.

1. Test the program on the instance `xwin10.dat` from the ATSP example files already used for our ZIMPL models by calling

   ```
   ./ATSPcuts -f <path-to-dir>/xwin10.dat
   ```

   The program should print the initial LP solution and wait for a key to be pressed.

2. Implement the method `setup_solution_graph()` which constructs the undirected graph corresponding to the solution. In the current setup it is called by `scip_enfolp()` and outputs the current LP value for each edge if it is nonzero. You need to adjust this code to build the graph. The result can be printed to the screen using `empty_graph.pretty_print( std::cerr )`.

3. Implement the `scip_enfolp()` callback to work correctly.

   (a) Adjust the `scip_enfolp()` callback to call `separate_connected_components()` with the solution graph.

   (b) Adjust this method to print the components of the graph and to build node ID sets corresponding to the vertices of each component.

(c) Implement the method `generate_subtour_elimination()` which adds a violated subtour elimination constraint based on a set of nodes to the model.

(d) In `scip_enfolp()`, call `separate_connected_components()` for each component if there is more than one.

4. Implement the `scip_check()` callback to work correctly as described in the comment there.

5. Test the code with some instances. It should now correctly solve them.

6. Implement the `scip_sepalp()` callback to get better performance.

   (a) Try to separate connected components of the solution graph.

   (b) If the solution graph has only one component, try to separate subtour elimination constraints using minimum cuts. To this end, implement `separate_min_cut()` as described in the comment and use it in `scip_sepalp()`.

   (c) Test the implementation.