Technische Universität Berlin
WiSe 2012

Prof. Andreas Bley

Computational mixed-integer programming

6th set of programming exercises:
Vehicle routing problem – A column generation algorithm

This exercise shows how to implement a solver for the vehicle routing problem using a formulation with exponentially many variables, each representing a "tour" (in fact, we use closed walks that generalize tours). Therefore, we use a pricer which dynamically generates new variables with negative reduced costs and adds them to the master problem. The pricer has to solve a combinatorial optimization problem to generate a new variable ("tour") with negative reduced cost.

Problem description

The (asymmetric) vehicle routing problem was introduced in the lecture. Note that we assume that the underlying digraph D is complete and the arc lengths c satisfy the triangle inequality.

One possible formulation is the following binary integer program:

$$\min \sum_{T \in \mathcal{T}} \lambda_T \tag{1}$$

$$\text{s.t.} \quad \sum_{t \in \mathcal{T}: v \in T} a_{T,v} \cdot \lambda_T \geq 1 \quad \forall\, v \in V \setminus \{r\} \tag{2}$$

$$\sum_{t \in \mathcal{T}: e} a_{T,e} \cdot \lambda_T = x_e \quad \forall\, e \in A \tag{3}$$

$$\lambda_T \geq 0 \quad \forall\, T \in \mathcal{T} \tag{4}$$

$$x_e \geq 0 \quad \forall\, e \in A \tag{5}$$

$$\lambda_T \in \{0,1\} \quad \forall\, T \in \mathcal{T} \tag{6}$$

$$x_e \in \{0,1\} \quad \forall\, e \in A \tag{7}$$

where

- $r \in V$ is the depot (in our program, this is the node with number 0)
- $\mathcal{T}$ is the set of all closed walks T starting and ending at r that do not exceed the given vehicle capacity U, i.e., with $\sum_{v \in V} a_{T,v} d_v \leq U$.
  (Optionally, you can also enforce that T contains at most $n = |V|$ arcs or, even more, that T is an elementary tour without node repetitions. This however, will change the pricing problem that you have to solve.)
- $a_{T,v}$ denotes the number of occurrences of node v in walk T
- $a_{T,e}$ denotes the number of occurrences of arc e in walk T

Since $\mathcal{T}$ can be of exponential size, we will use a column generation approach to solve this problem. In this approach, we iteratively search for variables representing better walks, i.e., walks with negative reduced cost. For a given solution λ^* of the (restricted) linear program, we denote the dual variable corresponding to constraint (2) for node $v \in V$ by π^*_v and the dual variable corresponding

to constraint (3) for arc $e \in A$ by π_e^*. In the pricing subproblem, we have to find a variable λ_T for a walk $T \in \mathcal{T}$ for which the reduced costs are negative, i.e.

$$\overline{c}(T) = \sum_{e \in T} a_{T,e} c_e - \sum_{e \in T} a_{T,e} \pi_e^* - \sum_{v \in T,\ v \neq r} a_{T,v} \pi_v^* < 0\,,$$

or prove that all walks have nonnegative reduced cost. If such a walk exists, we have to create and add its corresponding variable to the formulation.

Note that, assuming strictly positive arc lengths satisfying the triangle inequality and strictly positive node demands, we can drop either the constraints (6) or the constraints (7) from the model.

The exercise consists of two main parts. In the first part, we will work with the model presented above and drop the integrality constraints (6) for the tour variables. Thus, we can rely on the standard variable branching mechanism to enforce integrality for the arc variables x_e during branch-and-bound.

In the second part of the exercise, we remove the arc variables x_e and the corresponding constraints (3), (5), and (7) from the model. To enforce the integrality constraint (6) on the tour variables, we then have to implement our own branching mechanism.

Getting started

a) Extract the package `exercise06.tgz`.

b) In the `VRP` 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/VRPData.{h,cpp}` These files contain the complete VRP problem data, a function to read the problem data, and vectors storing (pointers to) all arc- and node-indexed variables and constraints of the model.

`src/VarData.h` An object of the class defined in this files is attached as data to each tour-(walk) variable created during column generation and stores the tour corresponding to this variable.

`src/Pricer.{h,cpp}` This class implements the pricer that generates new tour-variables. In the first part of the exercise, you have to implement the main functionalities of this class.

`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/BranchConstraintHandler.{h,cpp}` These files contain the handler class and helper functions to manage the constraints created by branching.

Branching constraints are constructed to be valid only locally at some node (in fact: subtree) of the branch and bound tree. For variables that are already in the model, the constraints are enforced via the constraint propagation mechanisms of scip at each node of the branch and bound tree.

Whenever some constraint becomes active (because we switch to a part of the branch and bound tree where it is active), all variables affected by the constraint must be locally fixed to 0.

If the model contains arc variables, fixing is applied also for the arc variables. Note that it is even sufficient to apply the fixing to only the arc variables in this case.

For variables that are not yet in the model, the initialization method of the pricer must enforce all locally active branching constraints by marking all fixed arcs as currently not available and ensuring that only available arcs are considered in the tour computation.

`src/BranchRule.{h,cpp}` This class implements the split branching discussed in the lecture, where two different sets of arcs are fixed to 0 in the two child nodes of the current branch-and-bound node.

`pqueue.h` A simple priority queue class you can (but need not) use in the implementation of the pricing algorithm.

`data/*.vrp` A folder with test instances (both easy and hard ones).

c) As usual, you can can compile the code with `make`, `make OPT=dbg`, or `make LPS=cpx` in the `VRP` folder. The executable file created is called `bin/vrp`.

Part 1

In the first part, we will work with the model including arc variables and without integrality constraints (6) for the tour variables. Thus, can rely on the default branching in `scip` and focus on the implementation of the pricing.

1.1 Make the basic framework run

First, we have to make sure that the overall framework works (with an incomplete version of the pricing).

1. Implement the addition of the node covering constraints (2) in the file `main.cpp`.
   (TODO 1 in `main.cpp`. Also read the rest of the file!)
2. Implement the remaining parts of the function `add_tour_variable` in file `Pricer.cpp`.
   (TODO 1 and 2 in `Pricer.cpp`)
3. Implement the computation of the reduced-cost arc lengths and the call to the pricing and add-variable functions in the function `pricing` in file `Pricer.cpp`.
   (TODO 3 in 4 in `Pricer.cpp`)

After these steps, the code should run and produce a valid solution with one tour per customer node (constructed via the dummy implementation in the function `find_shortest_tour` in `Pricer.cpp`.

1.2 Implement the pricing problem

This step consists of only one part

1. Replace the dummy implementation in function `find_shortest_tour` in `Pricer.cpp` by a correct one.
   (TODO 5 in `Pricer.cpp`)

Essentially, you have to implement an algorithm the correctly solves the problem of finding the shortest walk (or tour) T whose total demand does not exceed the vehicle capacity.

One option is to implement a (pseudo-polynomial) dynamic programming algorithm similar to the traditional shortest path algorithms. You might, for example, program an algorithm similar to Dijkstra's algorithm that uses tuples of the last node, the current length, the current demand, and maybe even the cardinality of the walk-prefixes (together with a properly defined order on these tuples) instead of the (node,walk length) pairs used in the traditional Dijkstra algorithm. Yet, any other algorithmic variant is welcome, too.

You also may implement an algorithm that solves the pricing problem over the set of elementary tours (i.e. without cycles that do not contain the depot) if you wish to do so.

Please test your shortest walk/tour implementation for some small examples!

After this task, your code should solve the problems.

Part 2

In the second part of the exercise, we will eventually remove the arc variables x_e and the corresponding constraints (3), (5), and (7) from the model. For this, we have to implement our own branching mechanism.

2.1 Implement the brancher still using arc variables

As a first step, we will implement the principle branching scheme using the arc variables. The goal is to implement a brancher that only fixes arc variables to 0, fixings to 1 are not allowed!

1. Set the value of `use_vrp_brancher` to `true` in `main.cpp`. This will turn on and activate our branching mechanisms. (TODO 2 in `main.cpp`)
2. Implement the construction (and selection) of the branching in `scip_execlp` in `BranchRule.cpp`. (TODO 1 in `BranchRule.cpp`)

   The two arc sets you construct here will be fixed to 0 in the two child nodes that are added to the branch and bound tree. You must make sure that these fixing do cut off the current fractional solution and that they reduce the solution space. Ideally, you construct branches that reduce the solution space in both child node substantially and also increase to lower bound obtained in both child nodes substantially.

   If the current solution is integer, you should not create a branching, of course!

After this part, your code should solve the problems using your brancher. You can check this via a look at the statistics printed when the code completes: If only your brancher and none of the scip default branchers created child nodes, this part is completed.

2.2 Remove the arc variables

Finally, we will remove the arc variables and also implement the branching using tour variables only.

1. Set the value of `data.use_arc_vars` to `false` in `main.cpp`. This will turn off the arc variables. (TODO 2 in `main.cpp`)
2. Implement the computation of the current arc usage values x_e^* and all other attributes that you need in your brancher based on the tour variables only. (TODO 2 in `BranchRule.cpp`)
3. Implement the fixing of existing tour variables that is implied by the branching constraints that are active at the current branch and bound node. For this, you have to complete function `fix_implied_variables` in file `BranchConstraintHandler.cpp`, which is called by the propagation method of the branch constraint handler for each active constraint.
   (TODO 2 in `BranchConstraintHandler.cpp`)

4. Implement the correct consideration of the locally active branching constraints in the pricer. For this, you have to complete the function `getBranchingRestrictions` in `Pricer.cpp` for the case that the model contains no arc variables. For each arc, you have to set the available/unavailable attribute correctly by explicitly checking if the arc has been blocked by one of the locally active branching constraints. (TODO 6 in `Pricer.cpp`)

That it folks!

After this subtask, you code should solve the problems also without using arc variables in the model.

Results

eil7*	99
eil13*	247
eil22	375
eil23	569
eil30	543
eil31*	300

Note: The instances marked with * do not satisfy the triangle inequality. The reported values correspond to the walk model, where (optimal) solutions may contain walks of the form (0-1-2-1-0) visiting some nodes multiple times. Note that these tours are not valid in a model that permits only elementary tours. In the elementary tour model, the optimal values for eil7 and eil31 are larger!