Mathematical Programming
Programming Exercise
Part 2: Exponentially-Sized Formulations

Helmuth Breitenfellner

Abstract

This report describes the exponentially-sized formulations of the k-nodes Minimal Spanning Tree (k-MST). The formulations are based on *Cycle Elimination Constraints* and *Directed Cutset Constraints*. These formulations have then been implemented using `CPLEX` in C++.

The implementation can be found on GitHub:
`https://github.com/helmuthb/mathprog/tree/master/ProgrammingExercise`

Cycle Elimination Constraints (*CEC*)

The following variables are introduced:

$z_i \in \{0,1\}$ ... Node i is selected
$x_{ij} \in \{0,1\}$ ... Edge i,j is selected
$y_{ij} \in \{0,1\}$ ... Arc i,j is taken in the tree, going from i to j

With this variables one can define the basic constraints for the *Cycle Elimination Constraints* formulation of the k-MST as follows:

$$x_{ij} \leq z_i, \;\; x_{ij} \leq z_j \tag{1}$$
$$y_{ij} + y_{ji} = x_{ij} \tag{2}$$
$$\sum_{i,j} x_{ij} = k \tag{3}$$
$$\sum_{j} z_j = k + 1 \tag{4}$$
$$\sum_{i} y_{0i} = 1 \tag{5}$$

These are the basic constraints.
Then, in both the *lazy cut* and the *user cut* situation, cycles in proposed solutions are identified and corresponding constraints are added to the model dynamically.
For adding the constraints I used the following approach:

- Create a weighted directed graph, giving each arc the weight of 1 if the edge has *not* been selected and a weight of 0 if the edge has been selected (binarize the weights)
- shuffle the arcs of the graph

- loop through the arcs, and search for cycles which contain the specified arc, by using Dijkstra algorithm to find the shortest path complementing the arc
- stop when the first additional constraint is found

This approach was tested against alternatives:

- Do not binarize the weights
- Only add constraints as *lazy cuts*
- Add more than one constraint

However, the presented approach was the fastest.

Directed Cutset Constraints (*DCC*)

The following variables are introduced:

$$\begin{aligned} z_i &\in \{0,1\} \ \ldots \text{ Node } i \text{ is selected} \\ x_{ij} &\in \{0,1\} \ \ldots \text{ Arc } i,j \text{ is taken in the tree, going from } i \text{ to } j \\ x^0_{ij} &\in \{0,1\} \ \ldots \text{ Edge } i,j \text{ is selected} \end{aligned}$$

Note that in contrast to the other implementations here the variable x denotes the directed arcs, not the undirected edges.
With this variables one can define the basic constraints for the *Directed Cutset Constraints* formulation of the k-MST as follows:

$$x^0_{ij} \leq z_i \tag{6}$$
$$x^0_{ij} \leq z_j \tag{7}$$
$$x_{ij} + x_{ji} = x^0_{ij} \tag{8}$$
$$\sum_{i,j} x_{ij} = k \tag{9}$$
$$\sum_{j} z_j = k + 1 \tag{10}$$
$$\sum_{i} x_{0i} = 1 \tag{11}$$

These are the basic constraints.
Then, in both the *lazy cut* and the *user cut* situation, using MaxFlow algorithm the minimal cuts are searched, and they are added to the model dynamically.
For adding the cuts I used the following approach:

- Create a weighted directed graph, giving each arc the weight of 1 if the arc has been selected and a weight of 0 if the arc has not been selected (binarized weights)
- shuffle the vertices of the graph
- loop through the vertices, and for each vertice search for minimal cuts between the root and the selected vertice, using the provided Max-Flow algorithm

- stop when the first additional constraint is found

This approach was tested against alternatives:

- Do not binarize the weights
- Only add constraints as *lazy cuts*
- Add more than one constraint

However, the presented approach was the fastest.

Implementation Details

The implementation used the framework provided in the lecture.
Some adaptations include:

- Use subclasses of the `kMST_ILP` class for the algorithms
- Add runtime options whether cuts shall be added earlier (`-c`) and whether multiple cuts shall be added (`-n <number>`)
- Quiet option for output of only minimal results (`-q`)
- Allow output of results with a `-v` option

Other than that the framework was (hopefully) used as supposed.

Output Table

Runtime of the algorithms was quite high, especially in cases with higher k. Only the first test cases could be executed, the larger graphs did not deliver results after more than 24 hours.
The runtimes refer to the output from the program and indicate CPU time for the solver - the actual time was of course larger.
For reference, the runtime of the MTZ-formulation is added.
The system used for the evaluation has an Intel® Xeon® CPU E31245 @ 3.30 GHz with 32 GB RAM.

			BnB-Nodes			CPU Time		
file	k	Objective	*CEC*	*DCC*	*MTZ*	*CEC*	*DCC*	*MTZ*
`g01.dat`	2	46	0	0	0	0.01	0.01	0.02
`g01.dat`	5	477	0	0	0	0.02	0.02	0.02
`g02.dat`	4	373	0	0	0	0.07	0.02	0.03
`g02.dat`	10	1390	0	26	0	0.04	0.09	0.05
`g03.dat`	10	725	0	0	0	0.18	0.06	0.10
`g03.dat`	25	3074	60	1151	807	11.53	8.03	0.56
`g04.dat`	14	909	0	9	3	3.63	0.44	0.17
`g04.dat`	35	3292	0	1416	355	1.24	15.68	0.43
`g05.dat`	20	1235	0	36	186	6.98	0.79	0.52
`g05.dat`	50	4898	732		2390	650.47		3.63

All higher instances did not produce results even after hours with the exponentially-sized formulations.

Interpretation

With the exponentially-size formulations the problem can no longer be handled. While the MTZ is capable to deal with the largest graphs in the test set, the formulations CEC and DCC just cannot produce results in a reasonable time frame.

Branch-and-Bound Nodes

The number of Branch-and-Bound Nodes can be seen as a measurement of the strength of the Polyhedron of the LP-relaxation. The smaller the Polyhedron is, the higher the probability that no branch-and-bound is needed and that the best (LP) solution is already integer.
In a test run it was visible that not adding the constraints for the LP-relaxation is increasing the number of BnB-nodes, consistent with this explanation.