Technische Universität Berlin Prof. Andreas Bley
WiSe 2012

Computational mixed-integer programming

3rd set of programming exercises: Open Pit Mine Production Scheduling – IP formulation and preprocessing

In this exercise, we will implement an IP formulation for the *Open Pit Mine Production Scheduling Problem* (OPMPSP) using the SCIP callable library. We are given the following data:

- a block model with block indices in $\mathcal{N} = \{0, \ldots, N-1\}$,
- for each block $i \in \mathcal{N}$:
  - a set $\mathcal{P}(i)$ of immediate predecessors,
  - the amount of rock R_i and of ore O_i contained in block i,
  - the cost m_i of mining block i,
  - the profit p_i from processing block i,
- mining and processing capacities M and P per period,
- and a number of time periods T with uniform discount factor δ.

We use the binary decision variables

$$x_{i,t} \in \{0,1\} \qquad \text{equal to 1 iff block } i \text{ has been mined within periods } 0, \ldots, t-1,$$

i.e. $x_{i,0}, \ldots, x_{i,T-1}$ is a sequence like $0, \ldots, 0, 1, \ldots, 1$ and $x_{i,t} - x_{i,t-1}$ is 1 if and only if block i is mined in period t.

Then a simple IP formulation for the OPMPSP reads

$$\max \sum_{t=0}^{T-1} \delta^t \sum_{i=0}^{N-1} (p_i - m_i)(x_{i,t} - x_{i,t-1})$$

$$\begin{aligned}
s.t. \quad & x_{i,t-1} \leqslant x_{i,t} && \text{for } i \in \mathcal{N}, t = 1, \ldots, T-1, && (1)\\
& \sum_{i=0}^{N-1} R_i x_{i,t} \leqslant (t+1)M && \text{for } t = 0, \ldots, T-1, && (2)\\
& \sum_{i=0}^{N-1} O_i x_{i,t} \leqslant (t+1)P && \text{for } t = 0, \ldots, T-1. && (3)\\
& x_{i,t} \leqslant x_{k,t} && \text{for } i \in \mathcal{N}, k \in \mathcal{P}(i), t = 0, \ldots, T-1, && (4)\\
& x_{i,t} \in \{0,1\} && \text{for } i \in \mathcal{N}, t = 0, \ldots, T-1, && (5)\\
& x_{i,-1} = 0 && \text{for } i \in \mathcal{N}. && (6)
\end{aligned}$$

(The variables $x_{i,-1}$ are used only to simplify the notation here, do not implement these!)

Getting started

(a) Extract the OPMPSP project (`tar xzfv exercise03.tgz`). Amongst others, it contains:

`src/main.cpp` Main file, which reads the data files from the command line, initializes SCIP, calls a function to build the above IP formulation, calls SCIP to solve it, and displays the solution. *(nothing to do here)*

`src/mine.h/cpp` **and** `src/digraph.h/cpp` Provides data structures for the digraph and the mine and methods for reading the data. *(nothing to do here, but look at the data structures* `Digraph` *and* `Mine`*)*

`src/model.h/cpp` Contains the method `create_mip`, which needs to be extended to build the above IP formulation within SCIP.

`src/presolver.h/cpp` Contains an OPMPSP-specific presolver which has to be extended to apply single block variable fixing and extend the clique table of SCIP.

`opmpsp.set` A SCIP settings file, which is automatically read by the command `SCIPreadParams(scip, "opmpsp.set")` in file `main.cpp`. This may be extended or changed, e.g. to set a time limit when dealing with larger instances.

`data/` A folder with small, randomly generated block model data of different sizes. Each instance is specified by two files: `randXXX.blocks` (containing the amount of rock and ore of each block) and `randXXX.arcs` (containing the immediate predecessors of each block). `XXX` indicates the number of blocks.

(b) In the folder `OPMPSP/lib`, create a softlink to your SCIP 3.0 directory, e.g.
`ln -s /usr/site-local/scipoptsuite-3.0.0/scip-3.0.0 scip`

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

Task 1: Build and solve the OPMPSP formulation

(a) Go to `scip.zib.de/doc/html/index.shtml` and check out how to search the online documentation for interface methods.

- All public interface methods are declared either in files `scip.h` or `pub_<...>.h`. These header files are listed on `scip.zib.de/doc/html/group__PUBLICMETHODS.html`. You can either search them via the online documentation or directly open the files from your SCIP installation and browse the source code.
  Attention: Be aware that the documentation mostly relates to the C language programming interface. In our exercises, we use the C++ interface. The methods of the C++ classes have (almost) the same signatures as the corresponding C functions. You will find the classes that provide the interfaces for the scip plugins in the subdirectory `src/objscip/} of \verb`/usr/site-local/scipoptsuite-3.0.0.
- If you are locking for information about a particular object of SCIP, such as a variable or a constraint, you should first search the corresponding `pub_<...>.h` header. E.g., in case of a constraint, `pub_cons.h`.
- If you need some information about the overall problem, you should start searching in `scip.h`.
- Last, but not least: If you already know the name of the method and are only looking for the parameters, you can search it directly using the search box in the top right corner of the online documentation.

(b) Open the file `src/model.cpp`: The function `create_mip` is already partially provided. You need to fill in the gaps marked by `BEGIN_TODO...END_TODO`: The variables are already propperly created with their upper and lower bounds and objective coefficients. Also the first class of constraints ia already implemented. You only need to create and add the constraints (2)–(4) analogously.

(c) In the folder `OPMPSP`, try to compile the code and run it on the smallest example: `bin/opmpsp data/rand035.blocks data/rand035.arcs`. The solving process can be interrupted by pressing `CTRL-C`. The optimal value is 12.756506.

Task 2: Implementing a problem-specific presolver – Variable fixing

The precedence graph $(\mathcal{N}, \{(i,k) \mid k \in \mathcal{P}(i)\})$ is usually given transitively reduced to save constraints of type (4). Suppose we additionally compute $\mathcal{C}(i)$, the full predecessor cone, i.e. the set containing i itself *and* all predecessors of block i in the transitive closure of the precedence graph. If the amount of rock in $\mathcal{C}(i)$ exceeds the mining capacity M, then block i cannot feasibly be excavated in the first time period, i.e. $x_{i,1}$ may be fixed to 0. In general, we may fix

$$x_{i,0}, \ldots, x_{i,t-1} = 0 \tag{7}$$

if $\sum_{k \in \mathcal{C}(i)} R_k > tM$ holds. The same holds for ore content exceeding the processing capacity P.

(a) Open the file `src/presolver.cpp`, which already contains a partial implementation of the presolver. You need to extend the execution method of the presolver `Presolver::exec`.

First, the transitive closure of the precedence graph has to be computed. This can be done using a dynamic programming approach. A function stub for this is already available in beginning of the file.

(b) Fill in the gaps marked by `BEGIN_TODO_1...END_TODO_1`: Complete the function `compute_closure`, which recusively computes the closure of a block by aggregating the closures of its predecessor blocks (which are the successors in the digraph).

(c) Fill in the gaps marked by `BEGIN_TODO_2...END_TODO_2`: compute the value of $\sum_{k \in \mathcal{C}(i)} R_k$ and $\sum_{k \in \mathcal{C}(i)} O_k$ and store it in `induced_rock[i]` resp. `induced_ore[i]`.

(d) Fill in the gap marked by `BEGIN_TODO_3...END_TODO_3`: implement the single block variable fixing according to (7).

Task 3: Implementing a problem-specific presolver – Building the clique graph

The idea of single block variable fixing can be extended to multiple blocks. As a special case, consider two blocks i, $j \in \mathcal{N}$, which are incomparable, i.e. $i \notin \mathcal{C}(j)$ and $j \notin \mathcal{C}(i)$. Again, if the amount of rock in $\mathcal{C}(i) \cup \mathcal{C}(j)$ exceeds the mining capacity M, then at most one of i and j can feasibly be excavated during the first time period. In general, if

$$\sum_{k \in \mathcal{C}(i) \cup \mathcal{C}(j)} R_k > tM \quad \text{or} \quad \sum_{k \in \mathcal{C}(i) \cup \mathcal{C}(j)} O_k > tP \tag{8}$$

holds for time period t, then the inequality

$$x_{i,t-1} + x_{j,t-1} \leqslant 1 \tag{9}$$

is valid. One way of using this information in SCIP is to add the edge $(x_{i,t-1}, x_{j,t-1})$ to the so-called *clique graph* (or conflict graph). The clique separator will then dynamically try to separate these inequalities if violated.

(d) Fill in all the remaining gaps marked by `BEGIN_TODO_4...END_TODO_4` to `BEGIN_TODO_6...END_TODO_6` by generating the two block conflicts (9): For each pair of blocks $i < j$, compute the union and the intersection of the two predecessor sets, compute $\sum_{k \in \mathcal{C}(i) \cup \mathcal{C}(j)} R_k$ and $\sum_{k \in \mathcal{C}(i) \cup \mathcal{C}(j)} O_k$, and for all t with (8), add the edge $(x_{i,t-1}, x_{j,t-1})$ to the clique graph. Use the command `SCIPaddClique` as explained in the code.

If the two blocks i and j have a common predecessor block, add the triple conflict $(x_{i,t-1}, x_{j,t-1}, k, \bar{t-1})$ for all common predecessors k to the conflict graph (as explained in th code).

4. Experiments

(a) Compile your project and test it on the provided OPMPSP instances. In the file opmpsp.set you can turn your presolving techniques off by adjusting the lines `presolving/opmpsp/varfix = ...` and `presolving/opmpsp/pckcliques = ...`. You may also have to choose the time and node limits suitably, because the primal-dual gap can sometimes not be closed completely.

(b) Pick one of the test instances in folder `data/` and analyse the computational behaviour with and without your presolving techniques:

- Do they help to increase the overall number of fixings and implications which SCIP is able to generate in presolving?
- How many clique cuts are generated? (Search the SCIP statistics.)
- How do they affect the dual bound at the root node and during solving?
- Experiment with the clique separator settings in `opmpsp.set`: By default, it is only called at the root node, but using `separating/clique/freq = ...` you can call it additionally during the branch-and-bound tree. Does this help to decrease the dual bound faster? Does it affect the primal bound?
- In general, which primal heuristics seem to be especially effective in finding or improving feasible solutions? (to decypher the abbreviations on the left of the SCIP output, type `disp heur` in a separate SCIP shell.)

A note of caution: Results on randomly generated data are not always conclusive for real-world instances.

Good luck!